Concurrent HOL

Peter Gammie

September 1, 2025

Abstract

This is a simple framework for expressing linear-time properties. It supports the usual programming constructs (including interleaving parallel composition), equational and inequational reasoning about these, compositional assume/guarantee specifications and refinement, and the mixing of specifications and programs, all shallowly embedded in Isabelle/HOL.

Contents

1 Introduction **4**
1.1 Road map . . . 4

2 Terminated Aczel sequences **4**
2.1 Traces . . . 5
2.2 Combinators on traces . . . 5
2.3 Behaviors . . . 15
2.4 Combinators on behaviors . . . 16

3 Point-free notation **23**

4 More lattice **25**
4.1 Boolean lattices and implication . . . 28
4.2 Compactness and algebraicity . . . 30

5 Closure operators **34**
5.1 Complete lattices and algebraic closures . . . 36
5.2 Closures over powersets . . . 40
5.3 Matroids and antimatroids . . . 42
5.4 Composition . . . 43
5.5 Path independence . . . 44
5.6 Some closures . . . 44
5.6.1 Reflexive, symmetric and transitive closures . . . 44
5.6.2 Relation image . . . 45
5.6.3 Kleene closure . . . 46

6 Galois connections **49**
6.1 Some Galois connections . . . 57

7 Heyting algebras **59**
7.1 Downwards closure of preorders (downsets) . . . 65

8 Safety logic **68**
8.1 Stuttering . . . 68
8.2 The *('a, 's, 'v) spec* lattice . . . 76
8.3 Irreducible elements . . . 78
8.4 Maps . . . 81
8.5 The idle process . . . 86
8.6 Actions . . . 87
8.7 Operations on return values . . . 94
8.8 Bind . . . 100

8.9 Kleene star . . . 112
8.10 Transition relations . . . 113
8.11 Sequential assertions . . . 124
8.11.1 Preconditions . . . 125
8.11.2 Postconditions . . . 128
8.11.3 Strongest postconditions . . . 132
8.12 Initial steps . . . 134
8.13 Heyting implication . . . 136
8.14 Miscellaneous algebra . . . 139

9 Constructions in the *('a, 's, 'v) spec* lattice 146
9.1 Constrains-at-most . . . 146
9.2 Abadi and Plotkin's composition principle . . . 152
9.3 Interference closure . . . 154
9.4 The *'a agent* datatype . . . 162
9.5 Parallel composition . . . 164
9.6 Specification Inhabitation . . . 180

10 "Next step" implication ala Abadi and Merz (and Lamport) 184
10.1 Compositionality ala Abadi and Merz (and Lamport) . . . 191

11 Stability 192

12 Refinement 195
12.1 Geenral rules for the *('a, 's, 'v) spec* lattice . . . 199
12.1.1 Actions . . . 200
12.1.2 Bind . . . 200
12.1.3 Interference . . . 205
12.1.4 Parallel . . . 205
12.2 A relational assume/guarantee program logic for the *(sequential, 's, 'v) spec* lattice . . . 207
12.2.1 Stability rules . . . 213

13 A programming language 215
13.1 The *('s, 'v) prog* lattice . . . 215
13.2 Morphisms to and from the *(sequential, 's, 'v) spec* lattice . . . 216
13.3 Programming language constructs . . . 219
13.3.1 Laws of programming . . . 220
13.4 Refinement for *('s, 'v) prog* . . . 234
13.4.1 Introduction rules . . . 234
13.4.2 Galois considerations . . . 235
13.4.3 Rules . . . 235
13.5 A relational assume/guarantee program logic for the *('s, 'v) prog* lattice . . . 237
13.5.1 Galois considerations . . . 237
13.5.2 A proof of the parallel rule using Abadi and Plotkin's composition principle . . . 239
13.6 Specification inhabitation . . . 242

14 More combinators 244

15 Structural local state 251
15.1 *spec.local* . . . 251
15.2 Local state transformations . . . 256
15.2.1 Permuting local actions . . . 263
15.3 *spec.localize* . . . 268
15.4 *spec.local_init* . . . 276
15.5 Hoist to *('s, 'v) prog* . . . 282
15.6 Refinement rules . . . 286
15.6.1 Data refinement . . . 288
15.7 Assume/guarantee . . . 290
15.8 Specification inhabitation . . . 291

16 A Temporal Logic of Safety (TLS) 293
16.1 Stuttering . . . 294
16.2 The *('a, 's, 'v) tls* lattice . . . 304
16.3 Irreducible elements . . . 305
16.4 The idle process . . . 307
16.5 Temporal Logic for *('a, 's, 'v) tls* . . . 308
16.5.1 Leads-to and leads-to-via . . . 326
16.5.2 Fairness . . . 327
16.6 Safety Properties . . . 329
16.7 Maps . . . 340
16.8 Abadi's axioms for TLA . . . 343
16.9 Tweak syntax . . . 345

17 Atomic sections 345
17.1 Inhabitation . . . 349
17.2 Assume/guarantee . . . 350

18 Exceptions 350

19 Assume/Guarantee rule sets 354
19.1 Implicit stabilisation . . . 354
19.1.1 Assume/guarantee rules using implicit stability . . . 356
19.2 Refinement with relational assumes . . . 358

20 Wickerson, Dodds and Parkinson: explicit stabilisation 361
20.1 Assume/Guarantee rules . . . 364
20.2 Examples . . . 366

21 Example: inhabitation 367

22 Example: findP 367

23 Example: data refinement (search) 373

24 Observations about safety closure 381
24.1 Liveness . . . 384
24.2 A Haskell-like *Ix* class . . . 386

25 A polymorphic heap 390
25.1 References . . . 395
25.2 Arrays . . . 397
25.2.1 Code generation constants: one-dimensional arrays . . . 397
25.2.2 User-facing arrays . . . 398
25.2.3 Stability . . . 403

26 A concurrent variant of Imperative HOL 404
26.1 Code generator setup . . . 406
26.1.1 Haskell . . . 406
26.2 Value-returning parallel . . . 408

27 Total store order (TSO) 409
27.1 References . . . 428
27.2 Inhabitation . . . 428
27.3 Code generator setup for TSO . . . 432
27.3.1 Haskell . . . 432
27.4 A TSO litmus test . . . 434

28 Floyd-Warshall all-pairs shortest paths 435

References 447

1 Introduction

This is a simple framework for expressing linear-time properties. It supports the usual programming constructs (including interleaving parallel composition), equational and inequational reasoning about these, compositional assume/guarantee specifications and refinement, and the mixing of specifications and programs, all shallowly embedded in Isabelle/HOL. The closest extent works to ours are by Xu and He (1991, 1994) and Dingel (1996, 2000, 2002). It is heavily influenced by Lamport (1994).

1.1 Road map

Rather than begin with *a priori* "laws of programming" we take finite and infinite sequences as models of system executions (§16). Also, as transforming realistic concurrent systems while preserving total correctness is too difficult to be usable, we adopt Lamport's approach to separating liveness and safety properties (Abadi and Lamport 1991) and do most of our work on safety properties.

The safety model consists of a series of closures (§5) over the powerset lattice of finite, non-empty, terminated "Aczel" sequences (§2), where each transition is ascribed to an agent. The termination marker supports sequential composition. The model of system executions is built similarly.

The *spec* lattice. Firstly and fundamentally we close under prefixes (§7.1), which captures precisely the safety properties (i.e., we identify a safety property with the set of sequences that satisfies it). We also close under stuttering ala Lamport (§8.1) to support refinement and the "laws of programming" (§13.3.1). All properties we consider therefore need to be stuttering invariant which is a mild constraint. We call the set of sets closed in this way the *spec* lattice (§8.2); we can interpret its points as propositions as it is a Heyting algebra. Its chief novelty is that it supports a logical presentation of assume/guarantee reasoning due to Abadi and Plotkin (§13.5.2) where parallel composition (§9.5) is simple (infinitary) conjunction ala Lamport (1994).

This lattice is satisfactory as a logic but deficient as a programming language; see Zwiers (1989) for an extended discussion on this point, and a solution for synchronous message passing. In brief, parallel composition-as-conjunction and the monad laws (§8.8) fail to meet expectations. We therefore look for a stronger closure condition.

The *prog* lattice. We take the view that a concurrent process is a parallel composition of sequential processes where the parallel composition itself yields a sequential process. Abadi and Plotkin's constrains-at-most (§9.1) closure adds interference to the ends of traces – sufficient to support their circular composition principle (§9.2) – but not their beginnings. Our interference closure (§9.3) makes this symmetric, ensuring that parallel composition conforms to expectations: the monad laws hold as do many of the "laws of programming" (§13.3.1). We define the *prog* type (§13.1) to be the interference-closed specifications. We reason about programs in *prog* using propositions in *spec* via a pair of morphisms that form a Galois connection (§13.2).

Refinement. Abadi and Plotkin's approach does not support refinement in our setting. We therefore adopt a "next step" implication (§10) and develop a logical account of compositional program refinement (§12). Refinement here is trace inclusion (i.e., the preservation of all safety properties).

Relational assume/guarantee. The definition of relational assume/guarantee in this setting is pleasantly intuitive (§12.2). Its key strength is that program phrases can be abstracted to relational assume/guarantee quadruples that can then be used as program phrases (§13.5). This generalises Morgan's specification statement to a concurrent setting.

State spaces. As is traditional with shallow embeddings in HOL, we defer state space and value considerations using polymorphism. We develop a mechanism that partially encapsulates local state (§15).

Miscellany. Along the way we assemble some facts about Heyting algebras (§7), and sometimes construct our closures (§5) from Galois connections (§6). We explore the impact of using safety properties and this mix of finite and infinite sequences on TLA (§16).

2 Terminated Aczel sequences

We model a *behavior* of a system as a non-empty finite or infinite sequence of the form $s_0 -a_1\rightarrow s_1 -a_2\rightarrow \ldots (\rightarrow v)?$ where s_i is a state, a_i an agent and v a return value for finite sequences (see §16). A *trace* is a finite sequence $s_0 -a_1\rightarrow s_1 -a_2\rightarrow \ldots -a_n\rightarrow s_n \rightarrow v$ for $n \geq 0$ with optional return value v (see §8). States, agents and return values are of arbitrary type.

2.1 Traces

setup ‹*Sign.mandatory-path trace*›

datatype (*aset*: *'a*, *sset*: *'s*, *vset*: *'v*) *t* =
  T (*init*: *'s*) (*rest*: (*'a* × *'s*) *list*) (*term*: *'v option*)
for
  map: *map*
  pred: *pred*
  rel: *rel*

declare *trace.t.map-id0*[*simp*]
declare *trace.t.map-id0*[*unfolded id-def*, *simp*]
declare *trace.t.map-sel*[*simp*]
declare *trace.t.set-map*[*simp*]
declare *trace.t.map-comp*[*unfolded o-def*, *simp*]
declare *trace.t.set*[*simp del*]

instance *trace.t* :: (*countable*, *countable*, *countable*) *countable* **by** *countable-datatype*

lemma *split-all*[*no-atp*]: — imitate the setup for *'a* × *'b* without the automation
  shows (⋀*x*. *PROP P x*) ≡ (⋀*s xs v*. *PROP P* (*trace.T s xs v*))
proof
  show *PROP P* (*trace.T s xs v*) **if** ⋀*x*. *PROP P x* **for** *s xs v* **by** (*rule that*)
next
  fix *x*
  assume ⋀*s xs v*. *PROP P* (*trace.T s xs v*)
  from ‹*PROP P* (*trace.T* (*trace.init x*) (*trace.rest x*) (*trace.term x*))› **show** *PROP P x* **by** *simp*
qed

lemma *split-All*[*no-atp*]:
  shows (∀ *x*. *P x*) ⟷ (∀ *s xs v*. *P* (*trace.T s xs v*)) (**is** *?lhs* ⟷ *?rhs*)
proof (*intro iffI allI*)
  show *P x* **if** *?rhs* **for** *x* **using** *that* **by** (*cases x*) *simp-all*
qed *simp*

lemma *split-Ex*[*no-atp*]:
  shows (∃ *x*. *P x*) ⟷ (∃ *s xs v*. *P* (*trace.T s xs v*)) (**is** *?lhs* ⟷ *?rhs*)
proof (*intro iffI allI*; *elim exE*)
  show ∃ *s xs v*. *P* (*trace.T s xs v*) **if** *P x* **for** *x* **using** *that* **by** (*cases x*) *fast*
qed *auto*

2.2 Combinators on traces

definition *final'* :: *'s* ⇒ (*'a* × *'s*) *list* ⇒ *'s* **where**
  final' s xs = *last* (*s* # *map snd xs*)

abbreviation (*input*) *final* :: (*'a*, *'s*, *'v*) *trace.t* ⇒ *'s* **where**
  final σ ≡ *trace.final'* (*trace.init σ*) (*trace.rest σ*)

definition *continue* :: $('a, 's, 'v)$ *trace.t* $\Rightarrow$ $('a \times 's)$ *list* $\times$ $'v$ *option* $\Rightarrow$ $('a, 's, 'v)$ *trace.t* (**infixl** ‹@−$_S$› *64*)
where
  σ @−$_S$ *xsv* = (*case trace.term* σ *of None* $\Rightarrow$ *trace.T* (*trace.init* σ) (*trace.rest* σ @ *fst xsv*) (*snd xsv*) | *Some v* $\Rightarrow$ σ)

definition *tl* :: $('a, 's, 'v)$ *trace.t* $\rightharpoonup$ $('a, 's, 'v)$ *trace.t* **where**
  tl σ = (*case trace.rest* σ *of* [] $\Rightarrow$ *None* | *x* # *xs* $\Rightarrow$ *Some* (*trace.T* (*snd x*) *xs* (*trace.term* σ)))

definition *dropn* :: *nat* $\Rightarrow$ $('a, 's, 'v)$ *trace.t* $\rightharpoonup$ $('a, 's, 'v)$ *trace.t* **where**
  dropn = ($\frown\!\!\!\frown$) *trace.tl*

definition *take* :: *nat* $\Rightarrow$ $('a, 's, 'v)$ *trace.t* $\Rightarrow$ $('a, 's, 'v)$ *trace.t* **where**
  take i σ = (*if i* $\leq$ *length* (*trace.rest* σ) *then trace.T* (*trace.init* σ) (*List.take i* (*trace.rest* σ)) *None else* σ)

type-synonym $('a, 's)$ *transitions* = $('a \times 's \times 's)$ *list*

primrec *transitions′* :: $'s \Rightarrow ('a \times 's)$ *list* $\Rightarrow$ $('a, 's)$ *trace.transitions* **where**
  transitions′ s [] = []
| *transitions′ s* (*x* # *xs*) = (*fst x*, *s*, *snd x*) # *transitions′* (*snd x*) *xs*

abbreviation (*input*) *transitions* :: $('a, 's, 'v)$ *trace.t* $\Rightarrow$ $('a, 's)$ *trace.transitions* **where**
  transitions $\sigma \equiv$ *trace.transitions′* (*trace.init* σ) (*trace.rest* σ)

setup ‹*Sign.mandatory-path final′*›

lemma *simps*[*simp*]:
  shows *trace.final′ s* [] = *s*
    and *trace.final′ s* (*x* # *xs*) = *trace.final′* (*snd x*) *xs*
    and *trace.final′ s* (*xs* @ *ys*) = *trace.final′* (*trace.final′ s xs*) *ys*
    and *idle*: *snd* ‘ *set xs* $\subseteq$ {*s*} $\Longrightarrow$ *trace.final′ s xs* = *s*
    and *snd* ‘ *set xs* $\subseteq$ {*s*} $\Longrightarrow$ *trace.final′ s* (*xs* @ *ys*) = *trace.final′ s ys*
    and *snd* ‘ *set ys* $\subseteq$ {*trace.final′ s xs*} $\Longrightarrow$ *trace.final′ s* (*xs* @ *ys*) = *trace.final′ s xs*
by (*simp-all add*: *trace.final′-def last-map image-subset-iff split*: *if-split-asm*)

lemma *map*:
  shows *trace.final′* (*sf s*) (*map* (*map-prod af sf*) *xs*) = *sf* (*trace.final′ s xs*)
by (*simp add*: *trace.final′-def last-map*)

lemma *replicate*:
  shows *trace.final′ s* (*replicate i as*) = (*if i* = *0 then s else snd as*)
by (*simp add*: *trace.final′-def*)

lemma *map-idle*:
  assumes (λ*x*. *sf* (*snd x*)) ‘ *set xs* $\subseteq$ {*sf s*}
  shows *sf* (*trace.final′ s xs*) = *sf s*
using *assms* **by** (*induct xs arbitrary*: *s*) *simp-all*

setup ‹*Sign.parent-path*›

setup ‹*Sign.mandatory-path tl*›

lemma *simps*[*simp*]:
  shows *trace.tl* (*trace.T s* [] *v*) = *None*
    and *trace.tl* (*trace.T s* (*x* # *xs*) *v*) = *Some* (*trace.T* (*snd x*) *xs v*)
by (*simp-all add*: *trace.tl-def*)

setup ‹*Sign.parent-path*›

lemma *dropn-alt-def*:
  shows *trace.dropn i σ*
      = (*case drop i* ((*undefined*, *trace.init σ*) # *trace.rest σ*) *of*
          [] ⇒ *None*
        | *x* # *xs* ⇒ *Some* (*trace.T* (*snd x*) *xs* (*trace.term σ*)))
proof(*induct i arbitrary*: *σ*)
  case *0* **show** *?case*
    by (*simp add*: *trace.dropn-def*)
next
  case (*Suc i σ*) **then show** *?case*
    by (*cases σ*; *cases trace.rest σ*; *simp add*: *trace.dropn-def drop-Cons′ split*: *list.split*)
qed

setup ‹*Sign.mandatory-path dropn*›

lemma *simps*[*simp*]:
  shows *0*: *trace.dropn 0 = Some*
    and *Suc*: *trace.dropn* (*Suc i*) *σ* = *Option.bind* (*trace.tl σ*) (*trace.dropn i*)
    and *dropn*: *Option.bind* (*trace.dropn i σ*) (*trace.dropn j*) = *trace.dropn* (*i* + *j*) *σ*
by (*simp-all add*: *trace.dropn-def pfunpow-add*)

lemma *Suc-right*:
  shows *trace.dropn* (*Suc i*) *σ* = *Option.bind* (*trace.dropn i σ*) *trace.tl*
by (*simp add*: *trace.dropn-def pfunpow-Suc-right del*: *pfunpow.simps*)

lemma *eq-none-length-conv*:
  shows *trace.dropn i σ* = *None* ⟷ *length* (*trace.rest σ*) < *i*
by (*auto simp*: *trace.dropn-alt-def split*: *list.split*)

lemma *eq-Some-length-conv*:
  shows (∃*σ′*. *trace.dropn i σ* = *Some σ′*) ⟷ *i* ≤ *length* (*trace.rest σ*)
by (*auto simp*: *trace.dropn-alt-def dest*: *drop-eq-Cons-lengthD split*: *list.split*)

lemma *eq-Some-lengthD*:
  assumes *trace.dropn i σ* = *Some σ′*
  shows *i* ≤ *length* (*trace.rest σ*)
using *assms trace.dropn.eq-Some-length-conv* **by** *blast*

setup ‹*Sign.parent-path*›

setup ‹*Sign.mandatory-path take*›

lemma *sel*:
  shows *trace.init* (*trace.take i σ*) = *trace.init σ*
    and *trace.rest* (*trace.take i σ*) = *List.take i* (*trace.rest σ*)
    and *trace.term* (*trace.take i σ*) = (*if i* ≤ *length* (*trace.rest σ*) *then None else trace.term σ*)
by (*simp-all add*: *trace.take-def*)

lemma *0*:
  shows *trace.take 0 σ* = *trace.T* (*trace.init σ*) [] *None*
by (*simp add*: *trace.take-def*)

lemma *Nil*:
  shows *trace.take i* (*trace.T s* [] *None*) = *trace.T s* [] *None*
by (*simp add*: *trace.take-def*)

lemmas *simps*[*simp*] =
  trace.take.sel

trace.take.0
trace.take.Nil

lemma *map*:
shows *trace.take i* (*trace.map af sf vf σ*) = *trace.map af sf vf* (*trace.take i σ*)
by (*simp add*: *trace.take-def take-map*)

lemma *append*:
shows *trace.take i* (*trace.T s* (*xs* @ *ys*) *v*) = *trace.T s* (*List.take i* (*xs* @ *ys*)) (*if length* (*xs* @ *ys*) < *i then v else None*)
by (*simp add*: *trace.take-def*)

lemma *take*:
shows *trace.take i* (*trace.take j σ*) = *trace.take* (*min i j*) *σ*
by (*simp add*: *min-le-iff-disj trace.take-def*)

lemma *continue*:
shows *trace.take i* (*σ* @−$_S$ *xsv*)
= *trace.take i σ* @−$_S$ (*List.take* (*i* − *length* (*trace.rest σ*)) (*fst xsv*),
if i ≤ *length* (*trace.rest σ*) + *length* (*fst xsv*) *then None else snd xsv*)
by (*simp add*: *trace.continue-def trace.take-def split*: *option.split*)

lemma *all-iff*:
shows *trace.take i σ* = *σ* ⟷ (*case trace.term σ of None* ⇒ *length* (*trace.rest σ*) | *Some* - ⇒ *Suc* (*length* (*trace.rest σ*))) ≤ *i* (**is** *?thesis1*)
and *σ* = *trace.take i σ* ⟷ (*case trace.term σ of None* ⇒ *length* (*trace.rest σ*) | *Some* - ⇒ *Suc* (*length* (*trace.rest σ*))) ≤ *i* (**is** *?thesis2*)
proof −
show *?thesis1* **by** (*cases σ*) (*simp add*: *trace.take-def split*: *option.split*)
then show *?thesis2*
by (*rule eq-commute-conv*)
qed

lemmas *all* = *iffD2*[*OF trace.take.all-iff*(*1*)]

lemma *Ex-all*:
shows *σ* = *trace.take* (*Suc* (*length* (*trace.rest σ*))) *σ*
by (*simp add*: *trace.take-def*)

lemma *replicate*:
shows *trace.take i* (*trace.T s* (*replicate j as*) *v*)
= *trace.T s* (*replicate* (*min i j*) *as*) (*if i* ≤ *j then None else v*)
by (*simp add*: *trace.take-def*)

setup ‹*Sign.parent-path*›

setup ‹*Sign.mandatory-path continue*›

lemma *sel*[*simp*]:
shows *trace.init* (*σ* @−$_S$ *xs*) = *trace.init σ*
and *trace.rest* (*σ* @−$_S$ *xsv*) = (*case trace.term σ of None* ⇒ *trace.rest σ* @ *fst xsv* | *Some v* ⇒ *trace.rest σ*)
and *trace.term* (*σ* @−$_S$ *xsv*) = (*case trace.term σ of None* ⇒ *snd xsv* | *Some v* ⇒ *trace.term σ*)
by (*simp-all add*: *trace.continue-def split*: *option.split*)

lemma *simps*[*simp*]:
shows *trace.T s xs None* @−$_S$ *ysv* = *trace.T s* (*xs* @ *fst ysv*) (*snd ysv*)
and *trace.T s xs* (*Some v*) @−$_S$ *ysv* = *trace.T s xs* (*Some v*)
and *σ* @−$_S$ ([], *None*) = *σ*

by (*simp-all add*: *trace.continue-def trace.t.expand split*: *option.split*)

lemma *Nil*:
shows σ @$-_S$ ([], *trace.term* σ) = σ
and *trace.T* (*trace.init* σ) [] *None* @$-_S$ (*trace.rest* σ, *trace.term* σ) = σ
by (*cases* σ) (*simp-all add*: *trace.continue-def split*: *option.split*)

lemma *map*:
shows *trace.map af sf vf* (σ @$-_S$ *xsv*) = *trace.map af sf vf* σ @$-_S$ *map-prod* (*map* (*map-prod af sf*)) (*map-option vf*) *xsv*
by (*simp add*: *trace.continue-def split*: *option.split*)

lemma *eq-trace-conv*:
shows σ @$-_S$ *xsv* = *trace.T s xs v* $\longleftrightarrow$ *trace.init* σ = *s* $\wedge$ (*case trace.term* σ *of None* $\Rightarrow$ *trace.rest* σ @ *fst xsv* = *xs* $\wedge$ *v* = *snd xsv* | *Some v'* $\Rightarrow$ *trace.rest* σ = *xs* $\wedge$ *v* = *Some v'*)
and *trace.T s xs v* = σ @$-_S$ *xsv* $\longleftrightarrow$ *trace.init* σ = *s* $\wedge$ (*case trace.term* σ *of None* $\Rightarrow$ *trace.rest* σ @ *fst xsv* = *xs* $\wedge$ *v* = *snd xsv* | *Some v'* $\Rightarrow$ *trace.rest* σ = *xs* $\wedge$ *v* = *Some v'*)
by (*case-tac* [!] σ) (*auto simp*: *trace.continue-def split*: *option.split*)

lemma *self-conv*:
shows (σ = σ @$-_S$ *xsv*) $\longleftrightarrow$ (*case trace.term* σ *of None* $\Rightarrow$ *xsv* = ([], *None*) | *Some* - $\Rightarrow$ *True*)
and (σ @$-_S$ *xsv* = σ) $\longleftrightarrow$ (*case trace.term* σ *of None* $\Rightarrow$ *xsv* = ([], *None*) | *Some* - $\Rightarrow$ *True*)
by (*cases* σ; *cases xsv*; *fastforce split*: *option.splits*)+

lemma *same-eq*:
shows (σ @$-_S$ *xsv* = σ @$-_S$ *ysv*) $\longleftrightarrow$ (*case trace.term* σ *of None* $\Rightarrow$ *xsv* = *ysv* | *Some* - $\Rightarrow$ *True*)
by (*fastforce simp*: *trace.continue-def prod.expand split*: *option.split*)

lemma *continue*:
shows σ @$-_S$ *xsv* @$-_S$ *ysv* = σ @$-_S$ (*case snd xsv of None* $\Rightarrow$ (*fst xsv* @ *fst ysv*, *snd ysv*) | *Some* - $\Rightarrow$ *xsv*)
by (*simp add*: *trace.continue-def split*: *option.split*)

lemma *take-drop-id*:
shows *trace.take i* σ @$-_S$ *case-option* ([], *None*) ($\lambda\sigma'$. (*trace.rest* σ', *trace.term* σ')) (*trace.dropn i* σ) = σ
by (*cases* σ)
(*clarsimp simp*: *trace.take-def trace.dropn-alt-def split*: *list.split*;
metis append-take-drop-id list.sel(*3*) *tl-drop*)

setup ‹*Sign.parent-path*›

Prefix ordering **instantiation** *trace.t* :: (*type*, *type*, *type*) *order*
begin

definition *less-eq-t* :: ('*a*, '*s*, '*v*) *trace.t relp* **where**
less-eq-t σ_1 σ_2 $\longleftrightarrow$ ($\exists$ *xsv*. σ_2 = σ_1 @$-_S$ *xsv*)

definition *less-t* :: ('*a*, '*s*, '*v*) *trace.t relp* **where**
less-t σ_1 σ_2 $\longleftrightarrow$ σ_1 $\leq$ σ_2 $\wedge$ σ_1 $\neq$ σ_2

instance
by *standard*
(*auto simp*: *less-eq-t-def less-t-def trace.continue.self-conv trace.continue.continue trace.continue.same-eq*
split: *option.splits*)

end

lemma *less-eqE*[*consumes 1*, *case-names prefix maximal*]:
assumes σ_1 $\leq$ σ_2

assumes $[\![trace.term\ \sigma_1 = None;\ trace.init\ \sigma_1 = trace.init\ \sigma_2;\ prefix\ (trace.rest\ \sigma_1)\ (trace.rest\ \sigma_2)]\!] \Longrightarrow P$
assumes $\bigwedge v.\ [\![trace.term\ \sigma_1 = Some\ v;\ \sigma_1 = \sigma_2]\!] \Longrightarrow P$
shows P
using *assms* **by** (*cases trace.term* σ_1) (*auto simp*: *trace.less-eq-t-def trace.continue.self-conv*)

lemmas *less-eq-extE*[*consumes 1*, *case-names prefix maximal*]
= *trace.less-eqE*[*of trace.T* s_1 xs_1 v_1 *trace.T* s_2 xs_2 v_2, *simplified*, *simplified conj-explode*]
for s_1 xs_1 v_1 s_2 xs_2 v_2

lemma *less-eq-self-continue*:
shows $\sigma \leq \sigma\ @-_S\ xsv$
using *trace.less-eq-t-def* **by** *blast*

lemma *less-eq-same-append-conv*:
shows *trace.T s xs v* $\leq$ *trace.T s'* (*xs* @ *ys*) $v' \longleftrightarrow s = s' \wedge (\forall v''.\ v = Some\ v'' \longrightarrow ys = [] \wedge v = v')$
by (*auto simp*: *trace.less-eq-t-def trace.continue.eq-trace-conv split*: *option.split*)

lemma *less-same-append-conv*:
shows *trace.T s xs v* $<$ *trace.T s'* (*xs* @ *ys*) $v' \longleftrightarrow s = s' \wedge v = None \wedge (ys \neq [] \vee (\exists v''.\ v' = Some\ v''))$
by (*cases v*) (*auto simp*: *trace.less-t-def trace.less-eq-same-append-conv*)

lemma *less-eq-Some*[*simp*]:
shows *trace.T s xs* (*Some v*) $\leq \sigma \longleftrightarrow$ *trace.init* $\sigma = s \wedge$ *trace.rest* $\sigma = xs \wedge$ *trace.term* $\sigma =$ *Some v*
by (*cases* σ) (*simp add*: *trace.less-eq-t-def*)

lemma *less-eq-None*:
shows $\sigma \leq$ *trace.T s xs None* $\longleftrightarrow$ *trace.init* $\sigma = s \wedge$ *prefix* (*trace.rest* σ) *xs* $\wedge$ *trace.term* $\sigma =$ *None*
and *trace.T s xs None* $\leq \sigma \longleftrightarrow$ *trace.init* $\sigma = s \wedge$ *prefix xs* (*trace.rest* σ)
by (*case-tac* [!] σ) (*auto simp*: *trace.less-eq-same-append-conv elim*!: *trace.less-eqE prefixE*)

lemma *less*:
shows *trace.T s xs v* $<$ $\sigma \longleftrightarrow$ *trace.init* $\sigma = s \wedge (\exists ys.$ *trace.rest* $\sigma = xs$ @ $ys \wedge$ (*trace.term* $\sigma =$ *None* $\longrightarrow ys \neq [])) \wedge v =$ *None*
and $\sigma <$ *trace.T s xs v* $\longleftrightarrow$ *trace.init* $\sigma = s \wedge (\exists ys.\ xs =$ *trace.rest* σ @ $ys \wedge (v =$ *None* $\longrightarrow ys \neq [])) \wedge$ *trace.term* $\sigma =$ *None*
by (*case-tac* [!] σ)
(*auto simp*: *trace.less-t-def trace.less-eq-t-def trace.continue.eq-trace-conv split*: *option.split-asm*)

lemma *less-eq-take*[*iff*]:
shows *trace.take i* $\sigma \leq \sigma$
by (*simp add*: *trace.take-def take-is-prefix trace.less-eq-None*)

lemma *less-eq-takeE*:
assumes $\sigma_1 \leq \sigma_2$
obtains i **where** $\sigma_1 =$ *trace.take i* σ_2
using *assms*
by (*cases* σ_1)
(*auto simp*: *trace.take-def*
elim!: *trace.less-eqE prefixE*
dest: *meta-spec*[**where** *x*=*length* (*trace.rest* σ_1)]
meta-spec[**where** *x*=*Suc* (*length* (*trace.rest* σ_1))])

lemma *less-eq-take-def*:
shows $\sigma_1 \leq \sigma_2 \longleftrightarrow (\exists i.\ \sigma_1 =$ *trace.take i* $\sigma_2)$
by (*blast elim*: *trace.less-eq-takeE*)

lemma *less-take-less-eq*:
assumes $\sigma <$ *trace.take* (*Suc i*) σ'

shows $\sigma \leq$ *trace.take i* σ'
using *assms*
by (*clarsimp simp*: *trace.less-t-def trace.less-eq-take-def trace.take.take*) (*metis le-SucE min-def*)

lemma *wfP-less*:
shows *wfP* ((<) :: (-, -, -) *trace.t relp*)
unfolding *wfp-def*
proof(*rule wf-subset*[*rotated*])
let *?r* = *inv-image* ({(*None*, *Some v*) |*v*. *True*} <∗*lex*∗> {(*x*, *y*). *strict-prefix x y*}) (λσ. (*trace.term* σ, *trace.rest* σ))
show *wf ?r*
using *wfp-def wfP-strict-prefix wf-def* **by** *fastforce*
show {(*x*, *y*). *x* < *y*} ⊆ *?r*
by (*auto simp*: *trace.less-t-def trace.less-eq-take-def trace.take.all-iff split*: *option.splits*)
qed

lemma *less-eq-same-cases*:
fixes *ys* :: (-, -, -) *trace.t*
assumes $xs_1 \leq ys$
assumes $xs_2 \leq ys$
shows $xs_1 \leq xs_2 \vee xs_2 \leq xs_1$
using *assms*
by (*clarsimp simp*: *trace.less-eq-take-def trace.take.take*) (*metis min.absorb-iff1 nle-le*)

setup ‹*Sign.mandatory-path take*›

lemma *mono*:
assumes $\sigma_1 \leq \sigma_2$
assumes $i \leq j$
shows *trace.take i* $\sigma_1 \leq$ *trace.take j* σ_2
using *assms*
by (*clarsimp simp*: *trace.less-eq-take-def trace.take.take*) (*metis min.assoc min.commute min-def*)

setup ‹*Sign.parent-path*›

setup ‹*Sign.mandatory-path map*›

lemmas *map* = *trace.t.map-comp*[*unfolded comp-def*]

lemma *monotone*:
shows *mono* (*trace.map af sf vf*)
by (*rule monoI*) (*fastforce simp*: *trace.less-eq-take-def trace.take.map*)

lemmas *strengthen*[*strg*] = *st-monotone*[*OF trace.map.monotone*]
lemmas *mono* = *monoD*[*OF trace.map.monotone*]

lemma *monotone-less*:
shows *monotone* (<) (<) (*trace.map af sf vf*)
by (*rule monotoneI*)
(*auto simp*: *trace.less-t-def trace.map.mono*[*OF order.strict-implies-order*] *trace.split-all*
elim!: *trace.less-eqE prefixE*)

lemma *less-eqR*:
assumes $\sigma_1 \leq$ *trace.map af sf vf* σ_2
obtains σ_2' **where** $\sigma_2' \leq \sigma_2$ **and** σ_1 = *trace.map af sf vf* σ_2'
using *assms* **by** (*fastforce simp*: *trace.less-eq-take-def trace.take.map*)

setup ‹*Sign.parent-path*›

setup ‹*Sign.mandatory-path rel*›

lemmas *eq* = *trace.t.rel-eq*

lemmas *mono* = *trace.t.rel-mono-strong*[*of ar sr vr* σ_1 σ_2 *ar′ sr′ vr′*] **for** *ar sr vr* σ_1 σ_2 *ar′ sr′ vr′*

lemma *strengthen*[*strg*]:
  assumes *st-ord F ar ar′*
  assumes *st-ord F sr sr′*
  assumes *st-ord F vr vr′*
  shows *st-ord F* (*trace.rel ar sr vr* σ_1 σ_2) (*trace.rel ar′ sr′ vr′* σ_1 σ_2)
using *assms* **by** (*cases F*) (*auto intro*!: *le-boolI elim*: *trace.rel.mono*)

lemma *length-rest*:
  assumes *trace.rel ar sr vr* σ_1 σ_2
  shows *length* (*trace.rest* σ_1)
      = *length* (*trace.rest* σ_2) ∧ (∀ *i*<*length* (*trace.rest* σ_1). *rel-prod ar sr* (*trace.rest* σ_1 ! *i*) (*trace.rest* σ_2 ! *i*))
by (*rule rel-funE*[*OF trace.t.sel-transfer*(*2*) *assms*]) (*simp add*: *list-all2-conv-all-nth*)

setup ‹*Sign.parent-path*›

setup ‹*Sign.mandatory-path take*›

lemma *rel*:
  assumes *trace.rel ar sr vr* σ_1 σ_2
  shows *trace.rel ar sr vr* (*trace.take i* σ_1) (*trace.take i* σ_2)
using *assms*
by (*auto simp*: *trace.take-def trace.t.rel-sel trace.rel.length-rest*
      elim: *rel-funE*[*OF trace.t.sel-transfer*(*1*)])

setup ‹*Sign.parent-path*›

setup ‹*Sign.mandatory-path transitions′*›

lemma *prefix-conv*:
  shows *prefix* (*trace.transitions′ s xs*) (*trace.transitions′ s ys*) ⟷ *prefix xs ys*
proof(*induct xs arbitrary*: *s ys*)
  case (*Cons x xs s ys*) **then show** *?case* **by** (*cases ys*) *auto*
qed *simp*

lemma *monotone*:
  shows *monotone prefix prefix* (*trace.transitions′ s*)
by (*rule monotoneI*) (*simp add*: *trace.transitions′.prefix-conv*)

lemma *append*:
  shows *trace.transitions′ s* (*xs* @ *ys*) = *trace.transitions′ s xs* @ *trace.transitions′* (*trace.final′ s xs*) *ys*
by (*induct xs arbitrary*: *s ys*) *simp-all*

lemma *eq-Nil-conv*:
  shows *trace.transitions′ s xs* = [] ⟷ *xs* = []
    and [] = *trace.transitions′ s xs* ⟷ *xs* = []
by (*case-tac* [!] *xs*) *simp-all*

lemma *eq-Cons-conv*:
  shows *trace.transitions′ s xs* = *y* # *ys* ⟷ (∃ *a s′ xs′*. *xs* = (*a*, *s′*) # *xs′* ∧ *y* = (*a*, *s*, *s′*) ∧ *ys* = *trace.transitions′ s′ xs′*)

and $y \# ys = trace.transitions' \ s \ xs \longleftrightarrow (\exists \ a \ s' \ xs'. \ xs = (a, \ s') \# xs' \wedge y = (a, \ s, \ s') \wedge ys = trace.transitions' \ s' \ xs')$
by (*case-tac* [!] *xs*) *auto*

lemma *inj-conv*:
shows $trace.transitions' \ s \ xs = trace.transitions' \ s \ ys \longleftrightarrow xs = ys$
by (*induct xs arbitrary*: *s ys*) (*auto simp*: *trace.transitions'.eq-Nil-conv trace.transitions'.eq-Cons-conv*)

lemma *continue*:
shows $trace.transitions \ (\sigma \ @-_S \ xsv)$
$= trace.transitions \ \sigma \ @ \ (case \ trace.term \ \sigma \ of \ None \Rightarrow trace.transitions' \ (trace.final \ \sigma) \ (fst \ xsv) \mid Some \ v \Rightarrow [])$
by (*simp add*: *trace.transitions'.append last-map trace.final'-def split*: *option.splits*)

lemma *idle-conv*:
shows $set \ (trace.transitions' \ s \ xs) \subseteq UNIV \times Id \longleftrightarrow snd \ ` \ set \ xs \subseteq \{s\}$
by (*induct xs arbitrary*: *s*) (*simp*; *fast*)+

lemma *map*:
shows $trace.transitions' \ (sf \ s) \ (map \ (map\text{-}prod \ af \ sf) \ xs)$
$= map \ (map\text{-}prod \ af \ (map\text{-}prod \ sf \ sf)) \ (trace.transitions' \ s \ xs)$
by (*induct xs arbitrary*: *s*) *simp-all*

setup ‹*Sign.parent-path*›

setup ‹*Sign.mandatory-path transitions*›

lemma *monotone*:
shows *monotone* $(\leq)$ *prefix trace.transitions*
by (*rule monotoneI*) (*metis prefix-order.eq-iff trace.less-eqE trace.transitions'.prefix-conv*)

lemmas *mono* = *monotoneD*[*OF trace.transitions.monotone*]

lemma *subseq*:
assumes $\sigma \leq \sigma'$
shows *subseq* $(trace.transitions \ \sigma) \ (trace.transitions \ \sigma')$
by (*rule prefix-imp-subseq*[*OF trace.transitions.mono*[*OF assms*]])

setup ‹*Sign.parent-path*›

setup ‹*Sign.parent-path*›

type-synonym $('a, \ 's) \ steps = ('a \times 's \times 's) \ set$

setup ‹*Sign.mandatory-path trace*›

definition $steps' :: 's \Rightarrow ('a \times 's) \ list \Rightarrow ('a, \ 's) \ steps$ **where**
$steps' \ s \ xs = set \ (trace.transitions' \ s \ xs) - UNIV \times Id$

abbreviation (*input*) $steps :: ('a, \ 's, \ 'v) \ trace.t \Rightarrow ('a, \ 's) \ steps$ **where**
$steps \ \sigma \equiv trace.steps' \ (trace.init \ \sigma) \ (trace.rest \ \sigma)$

setup ‹*Sign.mandatory-path steps'*›

lemma *simps*[*simp*]:
shows $trace.steps' \ s \ [] = \{\}$
and $trace.steps' \ s \ ((a, \ s) \# xs) = trace.steps' \ s \ xs$
and $s \neq snd \ x \Longrightarrow trace.steps' \ s \ (x \# xs) = insert \ (fst \ x, \ s, \ snd \ x) \ (trace.steps' \ (snd \ x) \ xs)$

and $(a, s', s') \notin$ *trace.steps'* s *xs*
and *snd* ‘ *set xs* $\subseteq \{s\} \Longrightarrow$ *trace.steps'* s *xs* $= \{\}$
and *trace.steps'* s $[x] =$ (*if* $s =$ *snd* x *then* $\{\}$ *else* $\{(fst\ x, s, snd\ x)\}$)
by (*simp-all add*: *trace.steps'-def insert-Diff-if trace.transitions'.idle-conv*)

lemma *Cons-eq-if*:
shows *trace.steps'* s $(x \# xs)$
$=$ (*if* $s =$ *snd* x *then trace.steps'* s *xs else insert* (*fst* x, s, *snd* x) (*trace.steps'* (*snd* x) *xs*))
by (*auto simp*: *trace.steps'-def*)

lemma *stuttering*:
shows *trace.steps'* s *xs* $\subseteq r \cup A \times Id \longleftrightarrow$ *trace.steps'* s *xs* $\subseteq r$
and *trace.steps'* s *xs* $\subseteq A \times Id \cup r \longleftrightarrow$ *trace.steps'* s *xs* $\subseteq r$
by (*auto simp*: *trace.steps'-def*)

lemma *empty-conv*[*simp*]:
shows *trace.steps'* s *xs* $= \{\} \longleftrightarrow$ *snd* ‘ *set xs* $\subseteq \{s\}$ (**is** *?thesis1*)
and $\{\}$ = *trace.steps'* s *xs* $\longleftrightarrow$ *snd* ‘ *set xs* $\subseteq \{s\}$ (**is** *?thesis2*)
proof −
show *?thesis1* **by** (*simp add*: *trace.steps'-def trace.transitions'.idle-conv*)
then show *?thesis2*
by (*rule eq-commute-conv*)
qed

lemma *append*:
shows *trace.steps'* s $(xs$ @ $ys)$
$=$ *trace.steps'* s *xs* $\cup$ *trace.steps'* (*trace.final'* s *xs*) *ys*
by (*simp add*: *trace.steps'-def trace.transitions'.append Un-Diff*)

lemma *map*:
shows *trace.steps'* (*sf* s) (*map* (*map-prod af sf*) *xs*) = *map-prod af* (*map-prod sf sf*) ‘ *trace.steps'* s *xs* − *UNIV* × *Id*
and *trace.steps'* s (*map* (*map-prod af id*) *xs*) = *map-prod af id* ‘ *trace.steps'* s *xs* − *UNIV* × *Id*
by (*force simp*: *trace.steps'-def trace.transitions'.map trace.transitions'.map*[**where** *sf*=*id*, *simplified*])+

lemma *memberD*:
assumes $(a, s, s') \in$ *trace.steps'* s_0 *xs*
shows $(a, s') \in$ *set xs*
using *assms* **by** (*induct xs arbitrary*: s_0) (*auto simp*: *trace.steps'.Cons-eq-if split*: *if-split-asm*)

setup ‹*Sign.parent-path*›

setup ‹*Sign.mandatory-path steps*›

lemma *monotone*:
shows *mono trace.steps*
by (*simp add*: *monoI trace.steps'-def Diff-mono set-subseq*[*OF trace.transitions.subseq*])

lemmas *mono* = *monoD*[*OF trace.steps.monotone*]
lemmas *strengthen*[*strg*] = *st-monotone*[*OF trace.steps.monotone*]

setup ‹*Sign.parent-path*›

setup ‹*Sign.mandatory-path aset*›

lemma *simps*:
shows *trace.aset* (*trace.T* s *xs* v) = *fst* ‘ *set xs*
by (*force simp*: *trace.t.set*)

setup ‹*Sign.parent-path*›

setup ‹*Sign.mandatory-path sset*›

lemma *simps*:
  shows *trace.sset* (*trace.T s xs v*) = *insert s* (*snd* ‘ *set xs*)
by (*fastforce simp*: *trace.t.set image-iff*)

lemma *dropn-le*:
  assumes *trace.dropn i σ* = *Some σ′*
  shows *trace.sset σ′* ⊆ *trace.sset σ*
using *assms*
by (*cases σ*; *cases σ′*)
   (*fastforce simp*: *trace.dropn-alt-def trace.sset.simps image-iff*
            split: *list.split-asm*
             dest: *arg-cong*[**where** *f*=*set*] *in-set-dropD*)

lemma *take-le*:
  shows *trace.sset* (*trace.take i σ*) ⊆ *trace.sset σ*
by (*cases σ*) (*auto simp*: *trace.take-def trace.sset.simps dest*: *in-set-takeD*)

lemma *mono*:
  shows *mono trace.sset*
by (*rule monoI*, *unfold trace.less-eq-take-def*)
   (*blast dest*: *subsetD*[*OF trace.sset.take-le*])

setup ‹*Sign.parent-path*›

setup ‹*Sign.parent-path*›

2.3 Behaviors

setup ‹*Sign.mandatory-path behavior*›

datatype (*aset*: *′a*, *sset*: *′s*, *vset*: *′v*) *t* =
  B (*init*: *′s*) (*rest*: (*′a* × *′s*, *′v*) *tllist*)
for
  map: *map*

definition *term* :: (*′a*, *′s*, *′v*) *behavior.t* ⇒ *′v option* **where**
  term ω = (*if tfinite* (*behavior.rest ω*) *then Some* (*terminal* (*behavior.rest ω*)) *else None*)

declare *behavior.t.map-id0*[*simp*]
declare *behavior.t.map-id0*[*unfolded id-def*, *simp*]
declare *behavior.t.map-sel*[*simp*]
declare *behavior.t.set-map*[*simp*]
declare *behavior.t.map-comp*[*unfolded o-def*, *simp*]
declare *behavior.t.set*[*simp del*]

lemma *split-all*[*no-atp*]: — imitate the setup for *′a* × *′b* without the automation
  shows (⋀*x*. *PROP P x*) ≡ (⋀*s xs*. *PROP P* (*behavior.B s xs*))
proof
  show *PROP P* (*behavior.B s xs*) **if** ⋀*x*. *PROP P x* **for** *s xs* **by** (*rule that*)
next
  fix *x*
  assume ⋀*s xs*. *PROP P* (*behavior.B s xs*)
  from ‹*PROP P* (*behavior.B* (*behavior.init x*) (*behavior.rest x*))› **show** *PROP P x* **by** *simp*

qed

lemma *split-All*[*no-atp*]:
  shows $(\forall x.\ P\ x) \longleftrightarrow (\forall s\ xs.\ P\ (behavior.B\ s\ xs))$ (**is** *?lhs* $\longleftrightarrow$ *?rhs*)
proof (*intro iffI allI*)
  fix *x* **assume** *?rhs* **then show** *P x* **by** (*cases x*) *simp-all*
qed *simp*

lemma *split-Ex*[*no-atp*]:
  shows $(\exists x.\ P\ x) \longleftrightarrow (\exists s\ xs.\ P\ (behavior.B\ s\ xs))$ (**is** *?lhs* $\longleftrightarrow$ *?rhs*)
proof (*intro iffI allI*; *elim exE*)
  fix *x* **assume** *P x* **then show** $\exists s\ xs.\ P\ (behavior.B\ s\ xs)$ **by** (*cases x*) *fast*
qed *auto*

2.4 Combinators on behaviors

definition *continue* :: $('a, 's, 'v)$ *trace.t* $\Rightarrow$ $('a \times 's, 'v)$ *tllist* $\Rightarrow$ $('a, 's, 'v)$ *behavior.t* (**infix** ‹$@-_B$› *64*) **where**
  $\sigma\ @-_B\ xs = behavior.B\ (trace.init\ \sigma)\ (tshift2\ (trace.rest\ \sigma,\ trace.term\ \sigma)\ xs)$

definition *tl* :: $('a, 's, 'v)$ *behavior.t* $\rightharpoonup$ $('a, 's, 'v)$ *behavior.t* **where**
  $tl\ \omega = (case\ behavior.rest\ \omega\ of\ TNil\ v \Rightarrow None \mid TCons\ x\ xs \Rightarrow Some\ (behavior.B\ (snd\ x)\ xs))$

definition *dropn* :: *nat* $\Rightarrow$ $('a, 's, 'v)$ *behavior.t* $\rightharpoonup$ $('a, 's, 'v)$ *behavior.t* **where**
  $dropn = (\frown\frown)\ behavior.tl$

definition *take* :: *nat* $\Rightarrow$ $('a, 's, 'v)$ *behavior.t* $\Rightarrow$ $('a, 's, 'v)$ *trace.t* **where**
  $take\ i\ \omega = uncurry\ (trace.T\ (behavior.init\ \omega))\ (ttake\ i\ (behavior.rest\ \omega))$

setup ‹*Sign.mandatory-path continue*›

lemma *simps*:
  shows $trace.T\ s\ xs\ None\ @-_B\ ys = behavior.B\ s\ (tshift\ xs\ ys)$
    and $trace.T\ s\ xs\ (Some\ v)\ @-_B\ ys = behavior.B\ s\ (tshift\ xs\ (TNil\ v))$
    and $trace.T\ s\ (x\ \#\ xs)\ w\ @-_B\ ys = behavior.B\ s\ (TCons\ x\ (tshift2\ (xs,\ w)\ ys))$
by (*simp-all add*: *behavior.continue-def*)

lemma *sel*[*simp*]:
  shows *init*: $behavior.init\ (\sigma\ @-_B\ xs) = trace.init\ \sigma$
    and *rest*: $behavior.rest\ (\sigma\ @-_B\ xs) = tshift2\ (trace.rest\ \sigma,\ trace.term\ \sigma)\ xs$
by (*simp-all add*: *behavior.continue-def*)

lemma *term-None*:
  assumes $trace.term\ \sigma = None$
  shows $\sigma\ @-_B\ xs = behavior.B\ (trace.init\ \sigma)\ (tshift\ (trace.rest\ \sigma)\ xs)$
by (*simp add*: *assms behavior.continue-def*)

lemma *term-Some*:
  assumes $trace.term\ \sigma = Some\ v$
  shows $\sigma\ @-_B\ xs = behavior.B\ (trace.init\ \sigma)\ (tshift\ (trace.rest\ \sigma)\ (TNil\ v))$
by (*simp add*: *assms behavior.continue-def*)

lemma *tshift2*:
  shows $\sigma\ @-_B\ tshift2\ xsv\ ys = (\sigma\ @-_S\ xsv)\ @-_B\ ys$
by (*simp add*: *behavior.continue-def tshift2-def tshift-append split*: *option.split*)

setup ‹*Sign.parent-path*›

setup ‹*Sign.mandatory-path tl*›

lemma *TNil*:
shows *behavior.tl* (*behavior.B s* (*TNil v*)) = *None*
by (*simp add*: *behavior.tl-def*)

lemma *TCons*:
shows *behavior.tl* (*behavior.B s* (*TCons x xs*)) = *Some* (*behavior.B* (*snd x*) *xs*)
by (*simp add*: *behavior.tl-def*)

lemma *eq-None-conv*:
shows *behavior.tl ω* = *None* ⟷ *is-TNil* (*behavior.rest ω*)
by (*simp add*: *behavior.tl-def split*: *tllist.split*)

lemma *continue-Cons*:
shows *behavior.tl* (*trace.T s* (*x* # *xs*) *v* @−$_B$ *ys*) = *Some* (*trace.T* (*snd x*) *xs v* @−$_B$ *ys*)
by (*simp add*: *behavior.tl-def behavior.continue-def*)

lemmas *simps*[*simp*] =
behavior.tl.TNil
behavior.tl.TCons
behavior.tl.eq-None-conv
behavior.tl.continue-Cons

lemma *tfiniteD*:
assumes *behavior.tl ω* = *Some ω′*
shows *tfinite* (*behavior.rest ω′*) ⟷ *tfinite* (*behavior.rest ω*)
using *assms* **by** (*auto simp*: *behavior.tl-def split*: *tllist.splits*)

setup ‹*Sign.parent-path*›

lemma *dropn-alt-def*:
shows *behavior.dropn i ω*
= (*case tdropn i* (*TCons* (*undefined*, *behavior.init ω*) (*behavior.rest ω*)) *of*
TNil - ⇒ *None*
| *TCons x xs* ⇒ *Some* (*behavior.B* (*snd x*) *xs*))
proof(*induct i arbitrary*: *ω*)
case *0* **show** *?case* **by** (*simp add*: *behavior.dropn-def*)
next
case (*Suc i ω*) **then show** *?case*
by (*cases ω*; *cases behavior.rest ω*; *cases i*;
simp add: *behavior.dropn-def tdropn-eq-TNil-conv tdropn-tlength split*: *tllist.splits*)
qed

setup ‹*Sign.mandatory-path dropn*›

lemma *simps*[*simp*]:
shows *0*: *behavior.dropn 0 ω* = *Some ω*
and *TNil*: *behavior.dropn i* (*behavior.B s* (*TNil v*)) = (*case i of 0* ⇒ *Some* (*behavior.B s* (*TNil v*)) | - ⇒ *None*)
by (*simp-all add*: *behavior.dropn-def split*: *nat.splits*)

lemma *TCons*:
shows *behavior.dropn i* (*behavior.B s* (*TCons x xs*))
= (*case i of 0* ⇒ *Some* (*behavior.B s* (*TCons x xs*)) | *Suc j* ⇒ *behavior.dropn j* (*behavior.B* (*snd x*) *xs*))
by (*simp add*: *behavior.dropn-def split*: *nat.splits*)

lemma *Suc*:
shows *behavior.dropn* (*Suc i*) *ω* = *Option.bind* (*behavior.tl ω*) (*behavior.dropn i*)

by (*simp-all add*: *behavior.dropn-def*)

lemma *bind-tl-commute*:
  shows *behavior.tl* $\omega \gg\!=$ *behavior.dropn* i = *behavior.dropn* i $\omega \gg\!=$ *behavior.tl*
by (*simp add*: *behavior.dropn-def pfunpow-swap1*)

lemma *Suc-right*:
  shows *behavior.dropn* (*Suc* i) ω = *Option.bind* (*behavior.dropn* i ω) *behavior.tl*
by (*simp add*: *behavior.dropn-def pfunpow-Suc-right del*: *pfunpow.simps*)

lemma *dropn*:
  shows *Option.bind* (*behavior.dropn* i ω) (*behavior.dropn* j) = *behavior.dropn* (i + j) ω
by (*simp add*: *behavior.dropn-def pfunpow-add*)

lemma *add*:
  shows *behavior.dropn* (i + j) = ($\lambda\omega$. *Option.bind* (*behavior.dropn* i ω) (*behavior.dropn* j))
by (*simp add*: *fun-eq-iff behavior.dropn.dropn*)

lemma *tfiniteD*:
  assumes *behavior.dropn* i ω = *Some* ω'
  shows *tfinite* (*behavior.rest* ω') $\longleftrightarrow$ *tfinite* (*behavior.rest* ω)
using *assms*
by (*induct i arbitrary*: ω')
   (*auto simp*: *behavior.dropn.Suc-right behavior.tl.tfiniteD split*: *bind-split-asm*)

lemma *shorterD*:
  assumes *behavior.dropn* i ω = *Some* ω'
  assumes $j \leq i$
  shows $\exists\omega''$. *behavior.dropn* j ω = *Some* ω''
using *assms*(*1*) *le-Suc-ex*[*OF assms*(*2*)]
by (*clarsimp simp flip*: *behavior.dropn.dropn split*: *Option.bind-split-asm*)

lemma *eq-None-tlength-conv*:
  shows *behavior.dropn* i ω = *None* $\longleftrightarrow$ *tlength* (*behavior.rest* ω) < *enat* i
proof(*induct i arbitrary*: ω)
  case *0* **show** *?case* **by** (*simp add*: *enat-0*)
next
  case (*Suc i*) **then show** *?case*
    by (*cases* ω; *cases behavior.rest* ω; *simp add*: *behavior.dropn.Suc enat-0-iff flip*: *eSuc-enat*)
qed

lemma *eq-Some-tlength-conv*:
  shows ($\exists\omega'$. *behavior.dropn* i ω = *Some* ω') $\longleftrightarrow$ *enat* i $\leq$ *tlength* (*behavior.rest* ω)
by (*metis behavior.dropn.eq-None-tlength-conv leD leI not-None-eq2*)

lemma *eq-Some-tlengthD*:
  assumes *behavior.dropn* i ω = *Some* ω'
  shows *enat* i $\leq$ *tlength* (*behavior.rest* ω)
using *assms behavior.dropn.eq-Some-tlength-conv* **by** *blast*

lemma *tlength-eq-SomeD*:
  assumes *enat* i $\leq$ *tlength* (*behavior.rest* ω)
  shows $\exists\omega'$. *behavior.dropn* i ω = *Some* ω'
using *assms behavior.dropn.eq-Some-tlength-conv* **by** *blast*

lemma *eq-Some-tdropnD*:
  assumes *behavior.dropn* i ω = *Some* ω'
  shows *tdropn* i (*behavior.rest* ω) = *behavior.rest* ω'

using *assms*
proof(*induct i arbitrary*: ω)
  case (*Suc i*) **then show** *?case*
    by (*cases* ω; *cases behavior.rest* ω; *fastforce simp*: *behavior.dropn.Suc*)
qed *simp*

lemma *continue-shorter*:
  assumes $i \leq$ *length* (*trace.rest* σ)
  shows *behavior.dropn i* (σ @$-_B$ *xs*) = *Option.bind* (*trace.dropn i* σ) ($\lambda\sigma'$. *Some* (σ' @$-_B$ *xs*))
using *assms*
proof(*induct i arbitrary*: σ)
  case (*Suc i* σ) **from** *Suc.prems* **show** *?case*
    by (*cases* σ; *cases trace.rest* σ)
      (*simp-all add*: *behavior.dropn.Suc flip*: *Suc.hyps*)
qed *simp*

lemma *continue-Some*:
  assumes *length* (*trace.rest* σ) < *i*
  assumes *trace.term* σ = *Some v*
  shows *behavior.dropn i* (σ @$-_B$ *xs*) = *None*
using *assms* **by** (*simp add*: *behavior.dropn.eq-None-tlength-conv tlength-tshift2 tlength-tshift*)

lemma *continue-None*:
  assumes *length* (*trace.rest* σ) < *i*
  assumes *trace.term* σ = *None*
  shows *behavior.dropn i* (σ @$-_B$ *xs*)
      = (*case tdropn* (*i* − *Suc* (*length* (*trace.rest* σ))) *xs of*
          TNil - ⇒ *None*
        | *TCons y ys* ⇒ *Some* (*behavior.B* (*snd y*) *ys*))
using *assms* **by** (*cases i*) (*auto simp*: *behavior.dropn-alt-def tdropn-tshift*)

lemma *continue*:
  shows *behavior.dropn i* (σ @$-_B$ *xs*)
      = (*if i* ≤ *length* (*trace.rest* σ)
          then Option.bind (*trace.dropn i* σ) ($\lambda\sigma'$. *Some* (σ' @$-_B$ *xs*))
          else if trace.term σ = *None*
              then case tdropn (*i* − *Suc* (*length* (*trace.rest* σ))) *xs of*
                    TNil - ⇒ *None*
                  | *TCons y ys* ⇒ *Some* (*behavior.B* (*snd y*) *ys*)
              else None)
by (*clarsimp simp*: *behavior.dropn.continue-None behavior.dropn.continue-Some*
                behavior.dropn.continue-shorter)

setup ‹*Sign.parent-path*›

setup ‹*Sign.parent-path*›

setup ‹*Sign.mandatory-path trace.take.behavior*›

lemma *take*:
  shows *trace.take i* (*behavior.take j* ω) = *behavior.take* (*min i j*) ω
by (*simp add*: *behavior.take-def trace.take-def split-def*
          ttake-eq-None-conv ttake-flat length-ttake take-fst-ttake
      split: *enat.split split-min*)

setup ‹*Sign.parent-path*›

setup ‹*Sign.mandatory-path behavior*›

setup ‹*Sign.mandatory-path take*›

lemma *simps*[*simp*]:
  shows *0*: *behavior.take 0 ω = trace.T (behavior.init ω) [] None*
    and *Suc-TNil*: *behavior.take (Suc i) (behavior.B s (TNil v)) = trace.T s [] (Some v)*
by (*simp-all add*: *behavior.take-def*)

lemma *sel*[*simp*]:
  shows *trace.init (behavior.take i ω) = behavior.init ω*
    and *trace.rest (behavior.take i ω) = fst (ttake i (behavior.rest ω))*
    and *trace.term (behavior.take i ω) = snd (ttake i (behavior.rest ω))*
by (*simp-all add*: *behavior.take-def split-def*)

lemma *monotone*:
  shows *mono (λi. behavior.take i ω)*
by (*rule monoI*) (*fastforce simp*: *trace.less-eq-take-def trace.take.behavior.take min-def*)

lemmas *mono = monoD*[*OF behavior.take.monotone*]

lemma *map*:
  shows *behavior.take i (behavior.map af sf vf ω) = trace.map af sf vf (behavior.take i ω)*
by (*induct i arbitrary*: *ω*) (*simp-all add*: *behavior.take-def split-def ttake-tmap split*: *tllist.split*)

lemma *continue*:
  shows *behavior.take i (σ @−$_B$ ω) = trace.take i σ @−$_S$ ttake (i − length (trace.rest σ)) ω*
by (*cases σ*)
   (*auto simp*: *behavior.take-def split-def trace.take-def ttake-tshift2 ttake-TNil*
       split: *option.split nat.split*)

lemma *all-continue*:
  assumes *tlength (behavior.rest ω) < enat i*
  shows *behavior.take i ω @−$_S$ xsv = behavior.take i ω*
using *assms*
by (*auto simp*: *behavior.take-def split-def trace.continue-def ttake-eq-None-conv*
       split: *option.split*)

lemma *continue-same*:
  shows *behavior.take i (behavior.take i ω @−$_B$ xsv) = behavior.take i ω*
by (*auto simp*: *behavior.take-def split-def ttake-tshift2 length-ttake*
            ttake-eq-None-conv ttake-eq-Nil-conv ttake-eq-Some-conv ttake-TNil
       split: *enat.split nat.split option.split*)

lemma *treplicate*:
  shows *behavior.take i (behavior.B s (treplicate j as v))*
      = trace.T s (List.replicate (min i j) as) (if j < i then Some v else None)
by (*simp add*: *behavior.take-def*)

lemma *trepeat*:
  shows *behavior.take i (behavior.B s (trepeat as)) = trace.T s (List.replicate i as) None*
by (*simp add*: *behavior.take-def ttake-trepeat*)

lemma *tshift*:
  shows *behavior.take i (behavior.B s (tshift xs ys)) = trace.take i (trace.T s xs None) @−$_S$ ttake (i − length xs) ys*
by (*simp add*: *behavior.take-def trace.take-def ttake-tshift split-def*)

lemma *length*:

shows *length* (*trace.rest* (*behavior.take j* ω))
= (*case tlength* (*behavior.rest* ω) *of enat i* $\Rightarrow$ *min i j* | $\infty \Rightarrow$ *j*)
by (*auto simp*: *length-ttake split*: *enat.split*)

lemma *add*:
shows *behavior.take* (*i* + *j*) ω
= *behavior.take i* ω @$-_S$ (*case behavior.dropn i* ω *of Some* ω' $\Rightarrow$ *ttake j* (*behavior.rest* ω'))
by (*auto simp*: *behavior.take-def split-def ttake-add Let-def behavior.dropn.eq-Some-tdropnD*
behavior.dropn.eq-None-tlength-conv
dest: *iffD1*[*OF ttake-eq-None-conv*(*1*)]
split: *option.split*)

lemma *term-Some-conv*:
shows *trace.term* (*behavior.take j* ω) = *Some v*
$\longleftrightarrow$ (*tlength* (*behavior.rest* ω) < *enat j* $\wedge$ *Some v* = *behavior.term* ω)
by (*auto simp*: *behavior.term-def ttake-eq-Some-conv tfinite-tlength-conv*)

lemma *dropn*:
assumes *behavior.dropn i* ω = *Some* ω'
shows *behavior.take j* ω' = *the* (*trace.dropn i* (*behavior.take* (*i* + *j*) ω))
using *assms*
proof(*induct i arbitrary*: *j* ω ω')
case (*Suc i j* ω) **then show** *?case*
by (*cases* ω; *cases behavior.rest* ω; *cases i*)
(*auto simp*: *behavior.dropn.Suc behavior.take-def split-def*)
qed *simp*

lemma *continue-id*:
assumes *tlength* (*behavior.rest* ω) < *enat i*
shows *behavior.take i* ω @$-_B$ *xs* = ω
using *assms* **by** (*simp add*: *behavior.continue-def tshift2-ttake-shorter*)

lemma *flat*:
assumes *tlength* (*behavior.rest* ω) < *enat i*
assumes *i* $\leq$ *j*
shows *behavior.take i* ω = *behavior.take j* ω
using *assms* **by** (*simp add*: *behavior.take-def ttake-flat*)

lemma *eqI*:
assumes $\bigwedge$*i*. *behavior.take i* ω_1 = *behavior.take i* ω_2
shows ω_1 = ω_2
using *assms*
by (*cases* ω_1; *cases* ω_2; *simp add*: *behavior.take-def case-prod-beta prod-eq-iff ttake-eq-imp-eq*)

setup ‹*Sign.parent-path*›

setup ‹*Sign.mandatory-path continue*›

lemma *take-drop-shorter*:
assumes *i* $\leq$ *j*
shows *behavior.take i* ω @$-_S$ *apfst* (*drop i*) (*ttake j* (*behavior.rest* ω)) = *behavior.take j* ω
using *assms*
by (*simp add*: *trace.continue-def behavior.take.flat ttake-eq-None-conv ttake-eq-Some-conv trace.t.expand*
flip: *take-fst-ttake*[**where** *i*=*i* **and** *j*=*j*, *simplified min-absorb1*[*OF assms*]]
split: *option.split*)

lemma *take-drop-id*:
shows *behavior.take i* ω @$-_B$ *behavior.rest* (*the* (*behavior.dropn i* ω)) = ω

by (*cases tlength* (*behavior.rest* ω) < *enat i*)
(*simp add*: *behavior.take.continue-id*,
simp add: *behavior.continue-def tshift2-ttake-tdropn-id behavior.dropn.tlength-eq-SomeD*
flip: *behavior.dropn.eq-Some-tdropnD*)

setup ‹*Sign.parent-path*›

setup ‹*Sign.mandatory-path aset*›

lemma *simps*:
shows *behavior.aset* (*behavior.B s xs*) = *fst* ‘ *tset xs*
by (*force simp*: *behavior.t.set*)

setup ‹*Sign.parent-path*›

setup ‹*Sign.mandatory-path sset*›

lemma *simps*:
shows *behavior.sset* (*behavior.B s xs*) = *insert s* (*snd* ‘ *tset xs*)
by (*fastforce simp*: *behavior.t.set image-iff*)

lemma *dropn-le*:
assumes *behavior.dropn i* ω = *Some* ω'
shows *behavior.sset* ω' ⊆ *behavior.sset* ω
using *assms*
by (*cases* ω; *cases* ω')
(*fastforce simp*: *behavior.dropn-alt-def behavior.sset.simps image-iff*
split: *tllist.split-asm*
dest: *arg-cong*[**where** *f*=*tset*] *in-tset-tdropnD*)

lemma *take-le*:
shows *trace.sset* (*behavior.take i* ω) ⊆ *behavior.sset* ω
by (*cases* ω)
(*auto simp*: *behavior.take-def behavior.sset.simps trace.sset.simps split-def*
dest: *in-set-ttakeD*)

setup ‹*Sign.parent-path*›

setup ‹*Sign.parent-path*›

setup ‹*Sign.mandatory-path trace.dropn.behavior*›

lemma *take*:
shows *trace.dropn i* (*behavior.take j* ω)
= (*if i* ≤ *j then Option.bind* (*behavior.dropn i* ω) ($\lambda\omega'$. *Some* (*behavior.take* (*j* − *i*) ω'))
else None)
proof(*cases i* ≤ *j*)
case *False* **then show** *?thesis*
by (*clarsimp simp*: *trace.dropn.eq-none-length-conv length-ttake split*: *enat.split*) *linarith*
next
case *True* **then show** *?thesis*
proof(*induct j arbitrary*: *i* ω)
case (*Suc j i* ω) **then show** *?case*
by (*cases* ω; *cases i*)
(*auto simp*: *behavior.dropn.Suc behavior.take-def split-def behavior.split-all*
split: *tllist.splits*)
qed *simp*
qed

setup ‹*Sign.parent-path*›

3 Point-free notation

Typically we define predicates as functions of a state. The following provide a somewhat comfortable point-free imitation of Isabelle/HOL's operators.

type-synonym *'s pred = 's ⇒ bool*

abbreviation (*input*)
pred-K :: 'b ⇒ 'a ⇒ 'b (‹⟨-⟩›) **where**
⟨*f*⟩ ≡ λ*s*. *f*

abbreviation (*input*)
pred-not :: 'a pred ⇒ 'a pred (‹¬ -› [*40*] *40*) **where**
¬*a* ≡ λ*s*. ¬*a s*

abbreviation (*input*)
pred-conj :: 'a pred ⇒ 'a pred ⇒ 'a pred (**infixr** ‹∧› *35*) **where**
a ∧ *b* ≡ λ*s*. *a s* ∧ *b s*

abbreviation (*input*)
pred-disj :: 'a pred ⇒ 'a pred ⇒ 'a pred (**infixr** ‹∨› *30*) **where**
a ∨ *b* ≡ λ*s*. *a s* ∨ *b s*

abbreviation (*input*)
pred-implies :: 'a pred ⇒ 'a pred ⇒ 'a pred (**infixr** ‹⟶› *25*) **where**
a ⟶ *b* ≡ λ*s*. *a s* ⟶ *b s*

abbreviation (*input*)
pred-iff :: 'a pred ⇒ 'a pred ⇒ 'a pred (**infixr** ‹⟷› *25*) **where**
a ⟷ *b* ≡ λ*s*. *a s* ⟷ *b s*

abbreviation (*input*)
pred-eq :: ('a ⇒ 'b) ⇒ ('a ⇒ 'b) ⇒ 'a pred (**infix** ‹=› *40*) **where**
a = *b* ≡ λ*s*. *a s* = *b s*

abbreviation (*input*)
pred-neq :: ('a ⇒ 'b) ⇒ ('a ⇒ 'b) ⇒ 'a pred (**infix** ‹≠› *40*) **where**
a ≠ *b* ≡ λ*s*. *a s* ≠ *b s*

abbreviation (*input*)
pred-If :: 'a pred ⇒ ('a ⇒ 'b) ⇒ ('a ⇒ 'b) ⇒ 'a ⇒ 'b (‹(*If* (-)/ *Then* (-)/ *Else* (-))› [*0*, *0*, *10*] *10*)
where *If P Then x Else y* ≡ λ*s*. *if P s then x s else y s*

abbreviation (*input*)
pred-less :: ('a ⇒ 'b::ord) ⇒ ('a ⇒ 'b) ⇒ 'a pred (**infix** ‹<› *40*) **where**
a < *b* ≡ λ*s*. *a s* < *b s*

abbreviation (*input*)
pred-less-eq :: ('a ⇒ 'b::ord) ⇒ ('a ⇒ 'b) ⇒ 'a pred (**infix** ‹≤› *40*) **where**
a ≤ *b* ≡ λ*s*. *a s* ≤ *b s*

abbreviation (*input*)
pred-greater :: ('a ⇒ 'b::ord) ⇒ ('a ⇒ 'b) ⇒ 'a pred (**infix** ‹>› *40*) **where**
a > *b* ≡ λ*s*. *a s* > *b s*

abbreviation (*input*)
pred-greater-eq :: $('a \Rightarrow 'b{::}ord) \Rightarrow ('a \Rightarrow 'b) \Rightarrow 'a\ pred$ (**infix** ‹$\geq$› *40*) **where**
$a \geq b \equiv \lambda s.\ a\ s \geq b\ s$

abbreviation (*input*)
pred-plus :: $('a \Rightarrow 'b{::}plus) \Rightarrow ('a \Rightarrow 'b) \Rightarrow 'a \Rightarrow 'b$ (**infixl** ‹$+$› *65*) **where**
$a + b \equiv \lambda s.\ a\ s + b\ s$

abbreviation (*input*)
pred-minus :: $('a \Rightarrow 'b{::}minus) \Rightarrow ('a \Rightarrow 'b) \Rightarrow 'a \Rightarrow 'b$ (**infixl** ‹$-$› *65*) **where**
$a - b \equiv \lambda s.\ a\ s - b\ s$

abbreviation (*input*)
pred-times :: $('a \Rightarrow 'b{::}times) \Rightarrow ('a \Rightarrow 'b) \Rightarrow 'a \Rightarrow 'b$ (**infixl** ‹$*$› *65*) **where**
$a * b \equiv \lambda s.\ a\ s * b\ s$

abbreviation (*input*)
pred-all :: $('b \Rightarrow 'a\ pred) \Rightarrow 'a\ pred$ (**binder** ‹$\forall$› *10*) **where**
$\forall x.\ P\ x \equiv \lambda s.\ \forall x.\ P\ x\ s$

abbreviation (*input*)
pred-ex :: $('b \Rightarrow 'a\ pred) \Rightarrow 'a\ pred$ (**binder** ‹$\exists$› *10*) **where**
$\exists x.\ P\ x \equiv \lambda s.\ \exists x.\ P\ x\ s$

abbreviation (*input*)
pred-app :: $('a \Rightarrow 'b \Rightarrow 'c) \Rightarrow ('a \Rightarrow 'b) \Rightarrow 'a \Rightarrow 'c$ (**infixl** ‹**$**› *100*) **where**
f **$** $g \equiv \lambda s.\ f\ s\ (g\ s)$

abbreviation (*input*)
pred-app′ :: $('b \Rightarrow 'a \Rightarrow 'c) \Rightarrow ('a \Rightarrow 'b) \Rightarrow 'a \Rightarrow 'c$ (**infixl** ‹**$$**› *100*) **where**
f **$$** $g \equiv \lambda s.\ f\ (g\ s)\ s$

abbreviation (*input*)
pred-member :: $('a \Rightarrow 'b) \Rightarrow ('a \Rightarrow 'b\ set) \Rightarrow 'a\ pred$ (**infix** ‹$\in$› *40*) **where**
$a \in b \equiv \lambda s.\ a\ s \in b\ s$

abbreviation (*input*)
pred-subseteq :: $('a \Rightarrow 'b\ set) \Rightarrow ('a \Rightarrow 'b\ set) \Rightarrow 'a\ pred$ (**infix** ‹$\subseteq$› *50*) **where**
$A \subseteq B \equiv \lambda s.\ A\ s \subseteq B\ s$

abbreviation (*input*)
pred-union :: $('a \Rightarrow 'b\ set) \Rightarrow ('a \Rightarrow 'b\ set) \Rightarrow 'a \Rightarrow 'b\ set$ (**infixl** ‹$\cup$› *65*) **where**
$a \cup b \equiv \lambda s.\ a\ s \cup b\ s$

abbreviation (*input*)
pred-inter :: $('a \Rightarrow 'b\ set) \Rightarrow ('a \Rightarrow 'b\ set) \Rightarrow 'a \Rightarrow 'b\ set$ (**infixl** ‹$\cap$› *65*) **where**
$a \cap b \equiv \lambda s.\ a\ s \cap b\ s$

abbreviation (*input*)
pred-conjoin :: $'a\ pred\ list \Rightarrow 'a\ pred$ **where**
pred-conjoin xs $\equiv$ *foldr* $(\wedge)$ *xs* $\langle True \rangle$

abbreviation (*input*)
pred-disjoin :: $'a\ pred\ list \Rightarrow 'a\ pred$ **where**
pred-disjoin xs $\equiv$ *foldr* $(\vee)$ *xs* $\langle False \rangle$

abbreviation (*input*)
pred-min :: ('*a* ⇒ '*b*::*ord*) ⇒ ('*a* ⇒ '*b*) ⇒ '*a* ⇒ '*b* **where**
pred-min x y ≡ λ*s*. *min* (*x s*) (*y s*)

abbreviation (*input*)
pred-max :: ('*a* ⇒ '*b*::*ord*) ⇒ ('*a* ⇒ '*b*) ⇒ '*a* ⇒ '*b* **where**
pred-max x y ≡ λ*s*. *max* (*x s*) (*y s*)

abbreviation (*input*)
NULL :: ('*a* ⇒ '*b option*) ⇒ '*a pred* **where**
NULL a ≡ λ*s*. *a s* = *None*

abbreviation (*input*)
EMPTY :: ('*a* ⇒ '*b set*) ⇒ '*a pred* **where**
EMPTY a ≡ λ*s*. *a s* = {}

abbreviation (*input*)
LIST-NULL :: ('*a* ⇒ '*b list*) ⇒ '*a pred* **where**
LIST-NULL a ≡ λ*s*. *a s* = []

abbreviation (*input*)
SIZE :: ('*a* ⇒ '*b*::*size*) ⇒ '*a* ⇒ *nat* **where**
SIZE a ≡ λ*s*. *size* (*a s*)

abbreviation (*input*)
SET :: ('*a* ⇒ '*b list*) ⇒ '*a* ⇒ '*b set* **where**
SET a ≡ λ*s*. *set* (*a s*)

abbreviation (*input*)
pred-singleton :: ('*a* ⇒ '*b*) ⇒ '*a* ⇒ '*b set* **where**
pred-singleton x ≡ λ*s*. {*x s*}

abbreviation (*input*)
pred-list-nth :: ('*a* ⇒ '*b list*) ⇒ ('*a* ⇒ *nat*) ⇒ '*a* ⇒ '*b* (**infixl** ‹**!**› *150*) **where**
xs **!** *i* ≡ λ*s*. *xs s* ! *i s*

abbreviation (*input*)
pred-list-append :: ('*a* ⇒ '*b list*) ⇒ ('*a* ⇒ '*b list*) ⇒ '*a* ⇒ '*b list* (**infixr** ‹**@**› *65*) **where**
xs **@** *ys* ≡ λ*s*. *xs s* @ *ys s*

abbreviation (*input*)
FST :: '*a pred* ⇒ ('*a* × '*b*) *pred* **where**
FST P ≡ λ*s*. *P* (*fst s*)

abbreviation (*input*)
SND :: '*b pred* ⇒ ('*a* × '*b*) *pred* **where**
SND P ≡ λ*s*. *P* (*snd s*)

abbreviation (*input*)
pred-pair :: ('*a* ⇒ '*b*) ⇒ ('*a* ⇒ '*c*) ⇒ '*a* ⇒ '*b* × '*c* (**infixr** ‹⊗› *60*) **where**
a ⊗ *b* ≡ λ*s*. (*a s*, *b s*)

4 More lattice

lemma (**in** *semilattice-sup*) *sup-iff-le*:
shows *x* ⊔ *y* = *y* ⟷ *x* ≤ *y*
and *y* ⊔ *x* = *y* ⟷ *x* ≤ *y*

by (*simp-all add*: *le-iff-sup ac-simps*)

lemma (**in** *semilattice-inf*) *inf-iff-le*:
  shows $x \sqcap y = x \longleftrightarrow x \leq y$
    and $y \sqcap x = x \longleftrightarrow x \leq y$
by (*simp-all add*: *le-iff-inf ac-simps*)

lemma *if-sup-distr*:
  fixes $t\ e$ :: -::*semilattice-sup*
  shows *if-sup-distrL*: (*if* b *then* $t_1 \sqcup t_2$ *else* e) = (*if* b *then* t_1 *else* e) $\sqcup$ (*if* b *then* t_2 *else* e)
    and *if-sup-distrR*: (*if* b *then* t *else* $e_1 \sqcup e_2$) = (*if* b *then* t *else* e_1) $\sqcup$ (*if* b *then* t *else* e_2)
by (*simp-all split*: *if-splits*)

lemma *INF-bot*:
  assumes $F\ i = (\bot$::-::*complete-lattice*$)$
  assumes $i \in X$
  shows $(\bigsqcap i{\in}X.\ F\ i) = \bot$
using *assms* **by** (*metis INF-lower bot.extremum-uniqueI*)

lemma *mcont-fun-app-const*[*cont-intro*]:
  shows *mcont Sup* ($\leq$) *Sup* ($\leq$) ($\lambda f.\ f\ c$)
by (*fastforce intro*!: *mcontI monotoneI contI simp*: *le-fun-def*)

declare *mcont-applyI*[*cont-intro*]

lemma *INF-rename-bij*:
  assumes *bij-betw* $\pi\ X\ Y$
  shows $(\bigsqcap y{\in}Y.\ F\ Y\ y) = (\bigsqcap x{\in}X.\ F\ (\pi\ `\ X)\ (\pi\ x))$
using *assms* **by** (*metis bij-betw-imp-surj-on image-image*)

lemma *Inf-rename-surj*:
  assumes *surj* π
  shows $(\bigsqcap x.\ F\ x) = (\bigsqcap x.\ F\ (\pi\ x))$
using *assms* **by** (*metis image-image*)

lemma *INF-unwind-index*:
  fixes A :: -⇒-:: *complete-lattice*
  assumes $i \in I$
  shows $(\bigsqcap x{\in}I.\ A\ x) = A\ i \sqcap (\bigsqcap x{\in}I - \{i\}.\ A\ x)$
by (*metis INF-insert assms insert-Diff*)

lemma *Sup-fst*:
  shows $(\bigsqcup x{\in}X.\ P\ (fst\ x)) = (\bigsqcup x{\in}fst\ `\ X.\ P\ x)$
by (*simp add*: *image-image*)

setup ‹*Sign.mandatory-path order*›

lemma *assms-cong*: — simplify assumptions only
  assumes $x = x'$
  shows $x \leq y \longleftrightarrow x' \leq y$
using *assms* **by** *simp*

lemma *concl-cong*: — simplify conclusions only
  assumes $y = y'$
  shows $x \leq y \longleftrightarrow x \leq y'$
using *assms* **by** *simp*

lemma *subgoal*: — cut for lattice logics

fixes P :: -::*semilattice-inf*
assumes $P \leq Q$
assumes $P \sqcap Q \leq R$
shows $P \leq R$
using *assms* **by** (*simp add*: *inf-absorb1*)

setup ‹*Sign.parent-path*›

Logical rules ala HOL **lemmas** *SupI* = *Sup-upper*
lemmas *rev-SUPI* = *SUP-upper2*[*of x A b f* **for** *x A b f*]
lemmas *SUPI* = *rev-SUPI*[*rotated*]

lemmas *SUPE* = *SUP-least*[**where** *u*=*z* **for** *z*]
lemmas *SupE* = *Sup-least*

lemmas *INFI* = *INF-greatest*
lemmas *InfI* = *Inf-greatest*
lemmas *infI* = *semilattice-inf-class.le-infI*

lemma *InfE*:
fixes *R*::-::*complete-lattice*
assumes $P\ x \leq R$
shows $(\bigsqcap x.\ P\ x) \leq R$
using *assms* **by** (*meson Inf-lower2 rangeI*)

lemma *INFE*:
fixes *R*::'*a*::*complete-lattice*
assumes $P\ x \leq R$
assumes $x \in A$
shows $\bigsqcap (P\ `\ A) \leq R$
using *assms* **by** (*metis INF-lower2*)

lemmas *rev-INFE* = *INFE*[*rotated*]

lemma *Inf-inf-distrib*:
fixes *P*::-::*complete-lattice*
shows $(\bigsqcap x.\ P\ x \sqcap Q\ x) = (\bigsqcap x.\ P\ x) \sqcap (\bigsqcap x.\ Q\ x)$
by (*simp add*: *INF-inf-distrib*)

lemma *Sup-sup-distrib*:
fixes *P*::-::*complete-lattice*
shows $(\bigsqcup x.\ P\ x \sqcup Q\ x) = (\bigsqcup x.\ P\ x) \sqcup (\bigsqcup x.\ Q\ x)$
by (*simp add*: *SUP-sup-distrib*)

lemma *Inf-inf*:
fixes Q :: -::*complete-lattice*
shows $(\bigsqcap x.\ P\ x \sqcap Q) = (\bigsqcap x.\ P\ x) \sqcap Q$
by (*simp add*: *INF-inf-const2*)

lemma *inf-Inf*:
fixes P :: -::*complete-lattice*
shows $(\bigsqcap x.\ P \sqcap Q\ x) = P \sqcap (\bigsqcap x.\ Q\ x)$
by (*simp add*: *INF-inf-const1*)

lemma *SUP-sup*:
fixes Q :: -::*complete-lattice*
assumes $X \neq \{\}$
shows $(\bigsqcup x{\in}X.\ P\ x \sqcup Q) = (\bigsqcup x{\in}X.\ P\ x) \sqcup Q$ (**is** *?lhs* = *?rhs*)

proof(*rule antisym*)
show *?lhs* $\leq$ *?rhs* **by** (*simp add*: *SUP-le-iff SupI le-supI1*)
from *assms* **show** *?rhs* $\leq$ *?lhs* **by** (*auto simp add*: *SUP-le-iff intro*: *SUPI le-supI1*)
qed

lemma *sup-SUP*:
fixes P :: -::*complete-lattice*
assumes $X \neq \{\}$
shows $(\bigsqcup x{\in}X.\ P \sqcup Q\ x) = P \sqcup (\bigsqcup x{\in}X.\ Q\ x)$
using *SUP-sup*[*OF assms*, **where** $P{=}Q$ **and** $Q{=}P$] **by** (*simp add*: *ac-simps*)

4.1 Boolean lattices and implication

lemma
shows *minus-Not*[*simp*]: $-\ Not = id$
and *minus-id*[*simp*]: $-\ id = Not$
by *fastforce+*

definition *boolean-implication* :: $'a$::*boolean-algebra* $\Rightarrow$ $'a \Rightarrow 'a$ (**infixr** ‹$\longrightarrow_B$› *60*) **where**
$x \longrightarrow_B y = -x \sqcup y$

definition *boolean-eq* :: $'a$::*boolean-algebra* $\Rightarrow$ $'a \Rightarrow 'a$ (**infixr** ‹$\longleftrightarrow_B$› *60*) **where**
$x \longleftrightarrow_B y = x \longrightarrow_B y \sqcap y \longrightarrow_B x$

setup ‹*Sign.mandatory-path boolean-implication*›

lemma *bool-alt-def*[*simp*]:
shows $P \longrightarrow_B Q = (P \longrightarrow Q)$
by (*auto simp*: *boolean-implication-def*)

lemma *pred--alt-def*[*simp*]:
shows $(P \longrightarrow_B Q)\ x = (P\ x \longrightarrow_B Q\ x)$
by (*auto simp*: *boolean-implication-def*)

lemma *set-alt-def*:
shows $P \longrightarrow_B Q = \{x.\ x \in P \longrightarrow x \in Q\}$
by (*auto simp*: *boolean-implication-def*)

lemma *member*:
shows $x \in P \longrightarrow_B Q \longleftrightarrow x \in P \longrightarrow x \in Q$
by (*simp add*: *boolean-implication.set-alt-def*)

lemmas *setI* = *iffD2*[*OF boolean-implication.member*, *rule-format*]

lemma *simps*[*simp*]:
shows
$\top \longrightarrow_B P = P$
$\bot \longrightarrow_B P = \top$
$P \longrightarrow_B \top = \top$
$P \longrightarrow_B P = \top$
$P \longrightarrow_B \bot = -P$
$P \longrightarrow_B -P = -P$
by (*simp-all add*: *boolean-implication-def shunt1*)

lemma *Inf-simps*[*simp*]: — Miniscoping: pushing in universal quantifiers.
shows
$\bigwedge P$ (Q::-::*complete-boolean-algebra*). $(\bigsqcap x.\ P\ x \longrightarrow_B Q) = ((\bigsqcup x.\ P\ x) \longrightarrow_B Q)$
$\bigwedge P$ (Q::-::*complete-boolean-algebra*). $(\bigsqcap x{\in}X.\ P\ x \longrightarrow_B Q) = ((\bigsqcup x{\in}X.\ P\ x) \longrightarrow_B Q)$

$\bigwedge P$ (Q::-⇒-::*complete-boolean-algebra*). ($\bigsqcap x.\ P \longrightarrow_B Q\ x$) = ($P \longrightarrow_B$ ($\bigsqcap x.\ Q\ x$))
$\bigwedge P$ (Q::-⇒-::*complete-boolean-algebra*). ($\bigsqcap x{\in}X.\ P \longrightarrow_B Q\ x$) = ($P \longrightarrow_B$ ($\bigsqcap x{\in}X.\ Q\ x$))
by (*simp-all add*: *boolean-implication-def INF-sup sup-INF uminus-SUP*)

lemma *mono*:
assumes $x' \le x$
assumes $y \le y'$
shows $x \longrightarrow_B y \le x' \longrightarrow_B y'$
using *assms* **by** (*simp add*: *boolean-implication-def sup.coboundedI1 sup.coboundedI2*)

lemma *strengthen*[*strg*]:
assumes *st-ord* ($\neg$ F) X X'
assumes *st-ord* F Y Y'
shows *st-ord* F ($X \longrightarrow_B Y$) ($X' \longrightarrow_B Y'$)
using *assms* **by** (*cases* F) (*use boolean-implication.mono* **in** *auto*)

lemma *eq-conv*:
shows ($P = Q$) $\longleftrightarrow$ ($P \longrightarrow_B Q$) $\sqcap$ ($Q \longrightarrow_B P$) = $\top$
unfolding *boolean-implication-def order.eq-iff* **by** (*simp add*: *sup-shunt top.extremum-unique*)

lemma *uminus-imp*[*simp*]:
shows $-(P \longrightarrow_B Q) = P \sqcap -Q$
by (*simp add*: *boolean-implication-def*)

lemma *cases-simp*[*simp*]:
shows ($P \longrightarrow_B Q$) $\sqcap$ ($-P \longrightarrow_B Q$) = Q
by (*simp add*: *boolean-implication-def order.eq-iff boolean-algebra.disj-conj-distrib2 shunt1*)

lemma *conv-sup*:
shows ($P \longrightarrow_B Q$) = $-P \sqcup Q$
by (*rule boolean-implication-def*)

lemma *infL*:
shows $P \sqcap Q \longrightarrow_B R = P \longrightarrow_B Q \longrightarrow_B R$
by (*simp add*: *boolean-implication-def sup-assoc*)

lemmas *uncurry* = *boolean-implication.infL*[*symmetric*]

lemma *shunt1*:
shows $x \sqcap y \le z \longleftrightarrow x \le y \longrightarrow_B z$
by (*simp add*: *boolean-implication-def shunt1*)

lemma *shunt2*:
shows $x \sqcap y \le z \longleftrightarrow y \le x \longrightarrow_B z$
by (*subst inf.commute*) (*rule boolean-implication.shunt1*)

lemma *mp*:
assumes $x \sqcap y \le z$
shows $x \le y \longrightarrow_B z$
using *assms* **by** (*simp add*: *boolean-implication.shunt1*)

lemma *imp-trivialI*:
assumes $P \sqcap -R \le -Q$
shows $P \le Q \longrightarrow_B R$
using *assms* **by** (*simp add*: *boolean-implication-def shunt2 sup.commute*)

lemma *shunt-top*:
shows $P \longrightarrow_B Q = \top \longleftrightarrow P \le Q$

by (*simp add*: *boolean-implication-def sup-shunt*)

lemma *detachment*:
shows $x \sqcap (x \longrightarrow_B y) = x \sqcap y$ (**is** *?thesis1*)
and $(x \longrightarrow_B y) \sqcap x = x \sqcap y$ (**is** *?thesis2*)
proof −
show *?thesis1* **by** (*simp add*: *boolean-algebra.conj-disj-distrib boolean-implication-def*)
then show *?thesis2* **by** (*simp add*: *ac-simps*)
qed

lemma *discharge*:
assumes $x' \leq x$
shows $x' \sqcap (x \longrightarrow_B y) = x' \sqcap y$ (**is** *?thesis1*)
and $(x \longrightarrow_B y) \sqcap x' = y \sqcap x'$ (**is** *?thesis2*)
proof −
from *assms* **show** *?thesis1*
by (*simp add*: *boolean-implication-def inf-sup-distrib sup.absorb2 le-supI1 flip*: *sup-neg-inf*)
then show *?thesis2*
by (*simp add*: *ac-simps*)
qed

lemma *trans*:
shows $(x \longrightarrow_B y) \sqcap (y \longrightarrow_B z) \leq (x \longrightarrow_B z)$
by (*simp add*: *boolean-implication-def inf-sup-distrib le-supI1 le-supI2*)

setup ‹*Sign.parent-path*›

4.2 Compactness and algebraicity

Fundamental lattice concepts drawn from Davey and Priestley (2002).

context *complete-lattice*
begin

definition *compact-points* :: *'a set* **where** — Davey and Priestley (2002, Definition 7.15(ii))
compact-points $= \{x.\ \forall S.\ x \leq \bigsqcup S \longrightarrow (\exists\, T{\subseteq}S.\ \mathit{finite}\ T \wedge x \leq \bigsqcup T)\}$

lemmas *compact-pointsI* = *subsetD*[*OF equalityD2*[*OF compact-points-def*], *simplified*, *rule-format*]
lemmas *compact-pointsD* = *subsetD*[*OF equalityD1*[*OF compact-points-def*], *simplified*, *rule-format*]

lemma *compact-point-bot*:
shows $\bot \in$ *compact-points*
by (*rule compact-pointsI*) *auto*

lemma *compact-points-sup*: — Davey and Priestley (2002, Lemma 7.16)
assumes $x \in$ *compact-points*
assumes $y \in$ *compact-points*
shows $x \sqcup y \in$ *compact-points*
proof(*rule compact-pointsI*)
fix S **assume** $x \sqcup y \leq \bigsqcup S$
with *compact-pointsD*[*OF assms*(*1*), *of S*] *compact-pointsD*[*OF assms*(*2*), *of S*]
obtain $X\ Y$
where $X \subseteq S \wedge \mathit{finite}\ X \wedge x \leq \bigsqcup X$
and $Y \subseteq S \wedge \mathit{finite}\ Y \wedge y \leq \bigsqcup Y$
by *auto*
then show $\exists\, T{\subseteq}S.\ \mathit{finite}\ T \wedge x \sqcup y \leq \bigsqcup T$
by (*simp add*: *exI*[**where** $x{=}X \cup Y$] *Sup-union-distrib le-supI1 le-supI2*)
qed

lemma *compact-points-Sup*: — Davey and Priestley (2002, Lemma 7.16)
assumes $X \subseteq$ *compact-points*
assumes *finite* X
shows $\bigsqcup X \in$ *compact-points*
using *assms*(2,1) **by** *induct* (*simp-all add*: *compact-point-bot compact-points-sup*)

lemma *compact-points-are-ccpo-compact*: — converse should hold
assumes $x \in$ *compact-points*
shows *ccpo.compact Sup* $(\leq)$ x
proof(*rule ccpo.compactI*[*OF complete-lattice-ccpo*], *rule ccpo.admissibleI*, *rule notI*)
fix X
assume *Complete-Partial-Order.chain* $(\leq)$ X **and** $x \leq \bigsqcup X$ **and** $*$: $X \neq \{\}$ $\forall\, y{\in}X.\ \neg x \leq y$
from *compact-pointsD*[*OF assms* ‹$x \leq \bigsqcup X$›]
obtain T **where** $T \subseteq X$ **and** *finite* T **and** $x \leq \bigsqcup T$ **by** *blast*
with $*$ *Complete-Partial-Order.chain-subset*[*OF* ‹*Complete-Partial-Order.chain* $(\leq)$ X› ‹$T \subseteq X$›]
show *False*
by (*auto simp*: *sup.absorb1 sup.absorb2 bot-unique dest*: *chainD dest*!: *finite-Sup-in*)
qed

definition *directed* :: *'a set* ⇒ *bool* **where** — Davey and Priestley (2002, Definition 7.7)
directed $X \longleftrightarrow X \neq \{\} \wedge (\forall\, x{\in}X.\ \forall\, y{\in}X.\ \exists\, z{\in}X.\ x \leq z \wedge y \leq z)$

lemmas *directedI* = *iffD2*[*OF directed-def*, *simplified conj-explode*, *rule-format*]
lemmas *directedD* = *iffD1*[*OF directed-def*]

lemma *directed-empty*:
assumes *directed* X
shows $X \neq \{\}$
using *assms* **by** (*simp add*: *directed-def*)

lemma *chain-directed*:
assumes *Complete-Partial-Order.chain* $(\leq)$ Y
assumes $Y \neq \{\}$
shows *directed* Y
using *assms* **by** (*metis chainD directedI*)

lemma *directed-alt-def*:
shows *directed* $X \longleftrightarrow (\forall\, Y{\subseteq}X.\ \mathit{finite}\ Y \longrightarrow (\exists\, x{\in}X.\ \forall\, y{\in}Y.\ y \leq x))$ (**is** *?lhs* $\longleftrightarrow$ *?rhs*)
proof(*rule iffI*)
have $\exists\, x{\in}X.\ \forall\, y{\in}Y.\ y \leq x$ **if** *finite* Y **and** *directed* X **and** $Y \subseteq X$ **and** $Y \neq \{\}$**for** Y
using *that* **by** *induct* (*force dest*: *directedD*)+
then show *?lhs* ⟹ *?rhs* **by** (*auto simp*: *directed-def*)
next
assume *?rhs* **show** *?lhs*
proof(*rule directedI*)
from ‹*?rhs*› **show** $X \neq \{\}$ **by** *blast*
fix x y **assume** $x \in X$ **and** $y \in X$ **with** ‹*?rhs*› **show** $\exists\, z{\in}X.\ x \leq z \wedge y \leq z$
by (*clarsimp dest*!: *spec*[**where** x={x, y}])
qed
qed

lemma *compact-points-alt-def*: — Davey and Priestley (2002, Definition 7.15(i) (finite points))
shows *compact-points* = {x::*'a*. $\forall\, D.$ *directed* $D \wedge x \leq \bigsqcup D \longrightarrow (\exists\, d{\in}D.\ x \leq d)$} (**is** *?lhs* = *?rhs*)
proof(*rule antisym*)
show *?lhs* ⊆ *?rhs*
by (*clarsimp simp*: *compact-points-def directed-alt-def*) (*metis Sup-least order.trans*)
next
have $*$: $\bigsqcup S = \bigsqcup \{\bigsqcup T \mid T.\ T \neq \{\} \wedge T \subseteq S \wedge \mathit{finite}\ T\}$ **for** S :: *'a set* — Davey and Priestley (2002, Exercise 7.5)

by (*fastforce intro*: *order.antisym Sup-subset-mono Sup-least exI*[**where** x=$\{x\}$ **for** x])
have $**$: *directed* $\{\bigsqcup T \mid T.\ T \neq \{\} \wedge T \subseteq S \wedge$ *finite* $T\}$ (**is** *directed ?D*) **if** $S \neq \{\}$ **for** $S :: 'a$ *set*
proof(*rule directedI*)
from ‹$S \neq \{\}$› **show** *?D* $\neq \{\}$ **by** *blast*
fix $x\ y$ **assume** $x \in$ *?D* $y \in$ *?D*
then obtain $X\ Y$
where $x = \bigsqcup X \wedge X \neq \{\} \wedge X \subseteq S \wedge$ *finite* X
and $y = \bigsqcup Y \wedge Y \neq \{\} \wedge Y \subseteq S \wedge$ *finite* Y
by *blast*
then show $\exists z{\in}$*?D*. $x \leq z \wedge y \leq z$
by − (*rule bexI*[**where** x=$\bigsqcup(X \cup Y)$]; *auto simp*: *Sup-subset-mono*)
qed
have $\exists T{\subseteq}X$. *finite* $T \wedge x \leq \bigsqcup T$ **if** $\forall D$. *directed* $D \wedge x \leq \bigsqcup D \longrightarrow (\exists d{\in}D.\ x \leq d)$ **and** $x \leq \bigsqcup X$ **for** $x\ X$
using *that* $*$[*of* X] $**$[*of* X] **by** *force*
then show *?rhs* $\subseteq$ *?lhs*
by (*fastforce intro*: *compact-pointsI*)
qed

lemmas *compact-points-directedD*
= *subsetD*[*OF equalityD1*[*OF compact-points-alt-def*], *simplified*, *rule-format*, *simplified conj-explode*, *rotated* −*1*]

end

class *algebraic-lattice* = *complete-lattice* + — Davey and Priestley (2002, Definition 7.18)
assumes *algebraic*: $(x :: 'a) = \bigsqcup(\{Y.\ Y \leq x\} \cap$ *compact-points*$)$
begin

lemma *le-compact*:
shows $x \leq y \longleftrightarrow (\forall z{\in}$*compact-points*. $z \leq x \longrightarrow z \leq y)$
by (*subst algebraic*) (*auto simp*: *Sup-le-iff*)

end

lemma (**in** *ccpo*) *compact-alt-def*:
shows *ccpo.compact Sup* $(\leq)\ x \longleftrightarrow (\forall Y.\ Y \neq \{\} \wedge$ *Complete-Partial-Order.chain* $(\leq)\ Y \wedge x \leq$ *Sup* $Y \longrightarrow (\exists y{\in}Y.\ x \leq y))$
by (*auto elim*!: *ccpo.compact.cases dest*: *ccpo.admissibleD intro*!: *compactI ccpo.admissibleI*)

lemma *compact-points-eq-finite-sets*: — Davey and Priestley (2002, Examples 7.17)
shows *compact-points* = *Collect finite* (**is** *?lhs* = *?rhs*)
proof(*rule antisym*)
have $*$: $X \subseteq \bigcup\{\{x\} \mid x.\ x \in X\}$ **for** $X :: 'a$ *set* **by** *blast*
show *?lhs* $\subseteq$ *?rhs* **by** (*force dest*: *compact-pointsD*[*OF* - $*$] *elim*: *finite-subset*)
next
show *?rhs* $\subseteq$ *?lhs* **by** (*metis CollectD compact-pointsI finite-subset-Union subsetI*)
qed

instance *set* :: (*type*) *algebraic-lattice*
by *standard* (*fastforce simp*: *compact-points-eq-finite-sets*)

context *semilattice-sup*
begin

definition *sup-irreducible-on* :: $'a$ *set* $\Rightarrow$ $'a \Rightarrow$ *bool* **where** — Davey and Priestley (2002, Definition 2.42)
sup-irreducible-on $A\ x \longleftrightarrow (\forall y{\in}A.\ \forall z{\in}A.\ x = y \sqcup z \longrightarrow x = y \vee x = z)$

abbreviation *sup-irreducible* :: $'a \Rightarrow$ *bool* **where**

sup-irreducible ≡ *sup-irreducible-on UNIV*

lemma *sup-irreducible-onI*:
assumes ⋀*y z*. ⟦*y* ∈ *A*; *z* ∈ *A*; *x* = *y* ⊔ *z*⟧ ⟹ *x* = *y* ∨ *x* = *z*
shows *sup-irreducible-on A x*
using *assms* **by** (*simp add*: *sup-irreducible-on-def*)

lemma *sup-irreducible-onD*:
assumes *sup-irreducible-on A x*
assumes *x* = *y* ⊔ *z*
assumes *y* ∈ *A*
assumes *z* ∈ *A*
shows *x* = *y* ∨ *x* = *z*
using *assms* **by** (*auto simp*: *sup-irreducible-on-def*)

lemma *sup-irreducible-on-less*: — Davey and Priestley (2002, Definition 2.42 (alt))
shows *sup-irreducible-on A x* ⟷ (∀ *y*∈*A*. ∀ *z*∈*A*. *y* < *x* ∧ *z* < *x* ⟶ *y* ⊔ *z* < *x*)
by (*simp add*: *sup-irreducible-on-def ac-simps sup.strict-order-iff*)
(*metis local.sup.commute local.sup.left-idem*)

end

lemma *sup-irreducible-bot*:
assumes ⊥ ∈ *A*
shows *sup-irreducible-on A* (⊥::-::*bounded-semilattice-sup-bot*)
using *assms* **by** (*auto intro*: *sup-irreducible-onI*)

lemma *sup-irreducible-le-conv*:
fixes *x*::-::*distrib-lattice*
assumes *sup-irreducible x*
shows *x* ≤ *y* ⊔ *z* ⟷ *x* ≤ *y* ∨ *x* ≤ *z*
by (*auto simp*: *inf.absorb-iff2 inf-sup-distrib1 ac-simps*
dest: *sup-irreducible-onD*[*OF assms sym*, *simplified*])

lemma *set-sup-irreducible*:
shows *sup-irreducible X* ⟷ (*X* = {} ∨ (∃ *y*. *X* = {*y*})) (**is** *?lhs* ⟷ *?rhs*)
proof(*rule iffI*)
show *?lhs* ⟹ *?rhs*
by (*auto simp*: *sup-irreducible-on-def*) (*metis insert-is-Un mk-disjoint-insert*)
show *?rhs* ⟹ *?lhs* **by** (*auto intro*: *sup-irreducible-onI*)
qed

definition *Sup-irreducible-on* :: *'a*::*complete-lattice set* ⇒ *'a* ⇒ *bool* **where** — Davey and Priestley (2002, Definition 10.26)
Sup-irreducible-on A x ⟷ (∀ *S*⊆*A*. *x* = ⨆ *S* ⟶ *x* ∈ *S*)

abbreviation *Sup-irreducible* :: *'a*::*complete-lattice* ⇒ *bool* **where**
Sup-irreducible ≡ *Sup-irreducible-on UNIV*

definition *Sup-prime-on* :: *'a*::*complete-lattice set* ⇒ *'a* ⇒ *bool* **where** — Davey and Priestley (2002, Definition 10.26)
Sup-prime-on A x ⟷ (∀ *S*⊆*A*. *x* ≤ ⨆ *S* ⟶ (∃ *s*∈*S*. *x* ≤ *s*))

abbreviation *Sup-prime* :: *'a*::*complete-lattice* ⇒ *bool* **where**
Sup-prime ≡ *Sup-prime-on UNIV*

lemma *Sup-irreducible-onI*:
assumes ⋀*S*. ⟦*S* ⊆ *A*; *x* = ⨆ *S*⟧ ⟹ *x* ∈ *S*

shows *Sup-irreducible-on A x*
using *assms* **by** (*simp add*: *Sup-irreducible-on-def*)

lemma *Sup-irreducible-onD*:
assumes $x = \bigsqcup S$
assumes $S \subseteq A$
assumes *Sup-irreducible-on A x*
shows $x \in S$
using *assms* **by** (*simp add*: *Sup-irreducible-on-def*)

lemma *Sup-prime-onI*:
assumes $\bigwedge S.\ [\![S \subseteq A;\ x \leq \bigsqcup S]\!] \Longrightarrow \exists\, s{\in}S.\ x \leq s$
shows *Sup-prime-on A x*
using *assms* **by** (*simp add*: *Sup-prime-on-def*)

lemma *Sup-prime-onE*:
assumes *Sup-prime-on A x*
assumes $x \leq \bigsqcup S$
assumes $S \subseteq A$
obtains s **where** $s \in S$ **and** $x \leq s$
using *assms Sup-prime-on-def* **by** *blast*

lemma *Sup-prime-on-conv*:
assumes *Sup-prime-on A x*
assumes $S \subseteq A$
shows $x \leq \bigsqcup S \longleftrightarrow (\exists\, s{\in}S.\ x \leq s)$
using *assms* **by** (*auto simp*: *Sup-prime-on-def intro*: *Sup-upper2*)

lemma *Sup-prime-not-bot*:
assumes *Sup-prime-on A x*
shows $x \neq \bot$
using *assms* **by** (*force simp*: *Sup-prime-on-def*)

lemma *Sup-prime-on-imp-Sup-irreducible-on*: — the converse holds in Heyting algebras
assumes *Sup-prime-on A x*
shows *Sup-irreducible-on A x*
using *Sup-upper*
by (*fastforce intro*!: *Sup-irreducible-onI intro*: *antisym elim*!: *Sup-prime-onE*[*OF assms*, *rotated*])

lemma *Sup-irreducible-on-imp-sup-irreducible-on*:
assumes *Sup-irreducible-on A x*
assumes $x \in A$
shows *sup-irreducible-on A x*
using *assms* **by** (*fastforce simp*: *Sup-irreducible-on-def sup-irreducible-on-def*
dest: *spec*[**where** $x{=}\{x,\ y\}$ **for** $x\ y$])

lemma *Sup-prime-is-compact*:
assumes *Sup-prime x*
shows $x \in$ *compact-points*
using *assms* **by** (*simp add*: *compact-points-alt-def Sup-prime-on-def*)

5 Closure operators

Our semantic spaces are modelled as lattices arising from the fixed points of various closure operators. We attempt to reduce our proof obligations by defining a locale for Kuratowski's closure axioms, where we do not requre strictness (i.e., it need not be the case that the closure maps $\bot$ to $\bot$). Davey and Priestley (2002, §2.33) term these *topped intersection structures*; see also Pfaltz and Šlapal (2013) for additional useful results.

locale *closure* =
  ordering ($\leq$) ($<$) — We use a partial order as a preorder does not ensure that the closure is idempotent
  for *less-eq* :: *'a* ⇒ *'a* ⇒ *bool* (**infix** ‹$\leq$› *50*)
    and *less* :: *'a* ⇒ *'a* ⇒ *bool* (**infix** ‹$<$› *50*)
+ **fixes** *cl* :: *'a* ⇒ *'a*
  assumes *cl*: $x \leq cl\ y \longleftrightarrow cl\ x \leq cl\ y$ — All-in-one non-strict Kuratowski axiom
begin

definition *closed* :: *'a set* **where** — These pre fixed points form a complete lattice ala Tarski/Knaster
  $closed = \{x.\ cl\ x \leq x\}$

lemma *closed-clI*:
  assumes $cl\ x \leq x$
  shows $x \in closed$
unfolding *closed-def* **using** *assms* **by** *simp*

lemma *expansive*:
  shows $x \leq cl\ x$
by (*simp add*: *cl refl*)

lemma *idempotent*[*simp*]:
  shows $cl\ (cl\ x) = cl\ x$
    and $cl \circ cl = cl$
using *cl antisym* **by** (*auto iff*: *expansive*)

lemma *monotone-cl*:
  shows *monotone* ($\leq$) ($\leq$) *cl*
by (*rule monotoneI*) (*meson cl expansive trans*)

lemmas *strengthen-cl*[*strg*] = *st-monotone*[*OF monotone-cl*]
lemmas *mono-cl*[*trans*] = *monotoneD*[*OF monotone-cl*]

lemma *least*:
  assumes $x \leq y$
  assumes $y \in closed$
  shows $cl\ x \leq y$
using *assms cl closed-def trans* **by** (*blast intro*: *expansive*)

lemma *least-conv*:
  assumes $y \in closed$
  shows $cl\ x \leq y \longleftrightarrow x \leq y$
using *assms expansive least trans* **by** *blast*

lemma *closed*[*iff*]:
  shows $cl\ x \in closed$
unfolding *closed-def* **by** (*simp add*: *refl*)

lemma *le-closedE*:
  assumes $x \leq cl\ y$
  assumes $y \in closed$
  shows $x \leq y$
using *assms closed-def trans* **by** *blast*

lemma *closed-conv*: — Typically used to manifest the closure using *subst*
  assumes $X \in closed$
  shows $X = cl\ X$
using *assms* **unfolding** *closed-def* **by** (*blast intro*: *antisym expansive*)

end

lemma (**in** *ordering*) *closure-axioms-alt-def*: — Equivalence with the Kuratowski closure axioms
  shows *closure-axioms* (≤) *cl* ⟷ (∀ *x*. *x* ≤ *cl x*) ∧ *monotone* (≤) (≤) *cl* ∧ (∀ *x*. *cl* (*cl x*) = *cl x*)
unfolding *closure-axioms-def monotone-def* **by** (*metis antisym trans refl*)

lemma (**in** *ordering*) *closureI*:
  assumes ⋀*x*. *x* ≤ *cl x*
  assumes *monotone* (≤) (≤) *cl*
  assumes ⋀*x*. *cl* (*cl x*) = *cl x*
  shows *closure* (≤) (<) *cl*
by (*blast intro*: *assms closure.intro*[*OF ordering-axioms*, *unfolded closure-axioms-alt-def*])

lemma *closure-inf-closure*:
  fixes *cl*$_1$ :: *'a*::*semilattice-inf* ⇒ *'a*
  assumes *closure-axioms* (≤) *cl*$_1$
  assumes *closure-axioms* (≤) *cl*$_2$
  shows *closure-axioms* (≤) (λ*X*. *cl*$_1$ *X* ⊓ *cl*$_2$ *X*)
using *assms* **unfolding** *closure-axioms-def* **by** (*meson order.trans inf-mono le-inf-iff order-refl*)

5.1 Complete lattices and algebraic closures

locale *closure-complete-lattice* =
  complete-lattice ⨅ ⨆ (⊓) (≤) (<) (⊔) ⊥ ⊤
+ *closure* (≤) (<) *cl*
    for *less-eqa* :: *'a* ⇒ *'a* ⇒ *bool* (**infix** ‹≤› *50*)
    and *lessa* (**infix** ‹<› *50*)
    and *infa* (**infixl** ‹⊓› *70*)
    and *supa* (**infixl** ‹⊔› *65*)
    and *bota* (‹⊥›)
    and *topa* (‹⊤›)
    and *Inf* (‹⨅›)
    and *Sup* (‹⨆›)
    and *cl* :: *'a* ⇒ *'a*
begin

lemma *cl-bot-least*:
  shows *cl* ⊥ ≤ *cl X*
using *cl* **by** *auto*

lemma *cl-Inf-closed*:
  shows *cl x* = ⨅ {*y* ∈ *closed*. *x* ≤ *y*}
by (*blast intro*: *sym*[*OF Inf-eqI*] *least expansive*)

lemma *cl-top*:
  shows *cl* ⊤ = ⊤
by (*simp add*: *top-le expansive*)

lemma *closed-top*[*iff*]:
  shows ⊤ ∈ *closed*
unfolding *closed-def* **by** *simp*

lemma *Sup-cl-le*:
  shows ⨆ (*cl* ‘ *X*) ≤ *cl* (⨆ *X*)
by (*meson cl expansive SUP-least Sup-le-iff*)

lemma *sup-cl-le*:
  shows *cl x* ⊔ *cl y* ≤ *cl* (*x* ⊔ *y*)

using *Sup-cl-le*[**where** X=$\{x, y\}$] **by** *simp*

lemma *cl-Inf-le*:
  shows *cl* $(\bigsqcap X) \leq \bigsqcap (cl\ `\ X)$
by (*meson cl expansive INF-greatest Inf-lower2*)

lemma *cl-inf-le*:
  shows *cl* $(x \sqcap y) \leq cl\ x \sqcap cl\ y$
using *cl-Inf-le*[**where** X=$\{x, y\}$] **by** *simp*

lemma *closed-Inf*:
  assumes $X \subseteq closed$
  shows $\bigsqcap X \in closed$
unfolding *closed-def* **using** *assms* **by** (*simp add*: *least Inf-greatest Inf-lower subset-eq*)

lemmas *closed-Inf'*[*intro*] = *closed-Inf*[*OF subsetI*]

lemma *closed-inf*[*intro*]:
  assumes $P \in closed$
  assumes $Q \in closed$
  shows $P \sqcap Q \in closed$
using *assms closed-Inf*[**where** X=$\{P, Q\}$] **by** *simp*

lemmas *mono2mono*[*cont-intro*, *partial-function-mono*] = *monotone2monotone*[*OF monotone-cl*, *simplified*]

definition *dense* :: *'a set* **where**
  dense = $\{x.\ cl\ x = \top\}$

lemma *dense-top*:
  shows $\top \in dense$
by (*simp add*: *dense-def cl-top*)

lemma *dense-Sup*:
  assumes $X \subseteq dense$
  assumes $X \neq \{\}$
  shows $\bigsqcup X \in dense$
using *assms* **by** (*fastforce simp*: *dense-def order.eq-iff intro*: *order.trans*[*OF - Sup-cl-le*] *elim*: *SUP-upper2*)

lemma *dense-sup*:
  assumes $P \in dense$
  assumes $Q \in dense$
  shows $P \sqcup Q \in dense$
using *assms dense-def top-le sup-cl-le* **by** *auto*

lemma *dense-le*:
  assumes $P \in dense$
  assumes $P \leq Q$
  shows $Q \in dense$
using *assms dense-def top-le mono-cl* **by** *auto*

lemma *dense-inf-closed*:
  shows $dense \cap closed = \{\top\}$
using *dense-def closed-conv closed-top* **by** *fastforce*

end

locale *closure-complete-lattice-class* =
  closure-complete-lattice $(\leq)$ $(<)$ $(\sqcap)$ $(\sqcup)$ $\bot$ :: - :: *complete-lattice* $\top$ *Inf Sup*

Traditionally closures for logical purposes are taken to be "algebraic", aka "consequence operators" (Davey and Priestley 2002, Definition 7.12), where *compactness* does the work of the finite/singleton sets.

locale *closure-complete-lattice-algebraic* = — Davey and Priestley (2002, Definition 7.12)
  closure-complete-lattice
+ **assumes** *algebraic-le*: $cl\ x \leq \bigsqcup (cl\ `\ (\{y.\ y \leq x\} \cap compact\text{-}points))$ — The converse is given by monotonicity
begin

lemma *algebraic*:
  shows $cl\ x = \bigsqcup (cl\ `\ (\{y.\ y \leq x\} \cap compact\text{-}points))$
by (*clarsimp simp*: *order.eq-iff Sup-le-iff algebraic-le expansive simp flip*: *cl elim*!: *order.trans*)

lemma *cont-cl*: — Equivalent to *algebraic-le* Davey and Priestley (2002, Theorem 7.14)
  shows *cont* $\bigsqcup$ $(\leq)$ $\bigsqcup$ $(\leq)$ *cl*
proof(*rule contI*)
  fix $X :: 'a\ set$
  assume X: *Complete-Partial-Order.chain* $(\leq)$ X $X \neq \{\}$
  show $cl\ (\bigsqcup X) = \bigsqcup (cl\ `\ X)$ (**is** *?lhs* = *?rhs*)
  proof(*rule order.antisym*[*OF - Sup-cl-le*])
    have *?lhs* = $\bigsqcup (cl\ `\ (\{y.\ y \leq \bigsqcup X\} \cap compact\text{-}points))$ **by** (*subst algebraic*) *simp*
    also from X **have** $\ldots \leq \bigsqcup (cl\ `\ (\{Y\ |Y\ x.\ Y \leq x \wedge x \in X\} \cap compact\text{-}points))$
      by (*auto dest*: *chain-directed compact-points-directedD intro*: *SUP-upper simp*: *Sup-le-iff*)
    also have $\ldots \leq$ *?rhs*
      by (*clarsimp simp*: *Sup-le-iff SUP-upper2 monotoneD*[*OF monotone-cl*])
    finally show *?lhs* $\leq$ *?rhs* **.**
  qed
qed

lemma *mcont-cl*:
  shows *mcont* $\bigsqcup$ $(\leq)$ $\bigsqcup$ $(\leq)$ *cl*
by (*simp add*: *mcontI*[*OF - cont-cl*])

lemma *mcont2mcont-cl*[*cont-intro*]:
  assumes *mcont luba orda* $\bigsqcup$ $(\leq)$ *P*
  shows *mcont luba orda* $\bigsqcup$ $(\leq)$ $(\lambda x.\ cl\ (P\ x))$
using *assms ccpo.mcont2mcont*[*OF complete-lattice-ccpo*] *mcont-cl* **by** *blast*

end

locale *closure-complete-lattice-algebraic-class* =
  closure-complete-lattice-algebraic $(\leq)$ $(<)$ $(\sqcap)$ $(\sqcup)$ $\bot$ :: - :: *complete-lattice* $\top$ *Inf Sup*

Our closures often satisfy the stronger condition of *distributivity* (see Scott (1980, §2)).

locale *closure-complete-lattice-distributive* =
  closure-complete-lattice
+ **assumes** *cl-Sup-le*: $cl\ (\bigsqcup X) \leq \bigsqcup (cl\ `\ X) \sqcup cl\ \bot$
begin

lemma *cl-Sup*:
  shows $cl\ (\bigsqcup X) = \bigsqcup (cl\ `\ X) \sqcup cl\ \bot$
by (*simp add*: *order.eq-iff Sup-cl-le cl-Sup-le cl-bot-least*)

lemma *cl-Sup-not-empty*:
  assumes $X \neq \{\}$
  shows $cl\ (\bigsqcup X) = \bigsqcup (cl\ `\ X)$
by (*metis assms cl-Sup cl-bot-least SUP-eq-const SUP-upper2 sup.orderE*)

lemma *cl-sup*:
  shows $cl\ (X \sqcup Y) = cl\ X \sqcup cl\ Y$

using *cl-Sup*[**where** X=$\{X,\ Y\}$] **by** (*simp add*: *sup-absorb1 cl-bot-least le-supI2*)

lemma *closed-sup*[*intro*]:
  assumes $P \in$ *closed*
  assumes $Q \in$ *closed*
  shows $P \sqcup Q \in$ *closed*
by (*metis assms cl-sup closed-conv closed*)

lemma *closed-Sup*: — Alexandrov: https://en.wikipedia.org/wiki/Alexandrov_topology
  assumes $X \subseteq$ *closed*
  shows $\bigsqcup X \sqcup$ *cl* $\bot \in$ *closed*
using *assms* **by** (*fastforce simp*: *cl-Sup cl-sup Sup-le-iff simp flip*: *closed-conv intro*: *closed-clI Sup-upper le-supI1*)

lemmas *closed-Sup'*[*intro*] = *closed-Sup*[*OF subsetI*]

lemma *cont-cl*:
  shows *cont* $\bigsqcup$ ($\leq$) $\bigsqcup$ ($\leq$) *cl*
by (*simp add*: *cl-Sup-not-empty contI*)

lemma *mcont-cl*:
  shows *mcont* $\bigsqcup$ ($\leq$) $\bigsqcup$ ($\leq$) *cl*
by (*simp add*: *mcontI*[*OF - cont-cl*])

lemma *mcont2mcont-cl*[*cont-intro*]:
  assumes *mcont luba orda* $\bigsqcup$ ($\leq$) *F*
  shows *mcont luba orda* $\bigsqcup$ ($\leq$) ($\lambda x.$ *cl* (*F x*))
using *assms ccpo.mcont2mcont*[*OF complete-lattice-ccpo*] *mcont-cl* **by** *blast*

lemma *closure-sup-irreducible-on*: — converse requires the closure to be T0
  assumes *sup-irreducible-on closed* (*cl x*)
  shows *sup-irreducible-on closed x*
using *assms sup-irreducible-on-def closed-conv closed-sup* **by** *auto*

end

locale *closure-complete-lattice-distributive-class* =
  closure-complete-lattice-distributive ($\leq$) ($<$) ($\sqcap$) ($\sqcup$) $\bot$ :: - :: *complete-lattice* $\top$ *Inf Sup*

locale *closure-complete-distrib-lattice-distributive-class* =
  closure-complete-lattice-distributive ($\leq$) ($<$) ($\sqcap$) ($\sqcup$) $\bot$ :: - :: *complete-distrib-lattice* $\top$ *Inf Sup*
begin

The lattice arising from the closed elements for a distributive closure is completely distributive, i.e., *Inf* and *Sup* distribute. See Davey and Priestley (2002, Section 10.23).

lemma *closed-complete-distrib-lattice-axiomI'*:
  assumes $\forall A{\in}A.\ \forall x{\in}A.\ x \in$ *closed*
  shows ($\bigsqcap X{\in}A.\ \bigsqcup X \sqcup$ *cl* $\bot$)
        $\leq \bigsqcup$ (*Inf* ‘ $\{f$ ‘ A |*f*. ($\forall X{\subseteq}$*closed*. *f X* $\in$ *closed*) $\wedge$ ($\forall Y \in A.\ f\ Y \in Y$)$\}$) $\sqcup$ *cl* $\bot$
proof −
  from *assms*
  have $\exists f'.\ f$ ‘ $A = f'$ ‘ $A \wedge$ ($\forall X{\subseteq}$*closed*. $f'\ X \in$ *closed*) $\wedge$ ($\forall Y{\in}A.\ f'\ Y \in Y$)
    if $\forall Y{\in}A.\ f\ Y \in Y$ **for** *f*
    using *that* **by** (*fastforce intro*!: *exI*[**where** $x{=}\lambda x.$ *if f x* $\in$ *closed then f x else cl* $\bot$])
  then show *?thesis*
    by (*clarsimp simp*: *Inf-Sup Sup-le-iff simp flip*: *INF-sup intro*!: *le-supI1 Sup-upper*)
qed

lemma *closed-complete-distrib-lattice-axiomI*[*intro*]:

assumes $\forall A{\in}A.\ \forall x{\in}A.\ x \in closed$
shows $(\bigsqcap X{\in}A.\ \bigsqcup X \sqcup cl\ \bot)$
$\leq \bigsqcup (Inf\ `\ \{B.\ (\exists f.\ (\forall x.\ (\forall x{\in}x.\ x \in closed) \longrightarrow f\ x \in closed)$
$\wedge\ B = f\ `\ A \wedge (\forall Y{\in}A.\ f\ Y \in Y)) \wedge (\forall x{\in}B.\ x \in closed)\})$
$\sqcup\ cl\ \bot$
by (*rule order.trans*[*OF closed-complete-distrib-lattice-axiomI'*[*OF assms*]])
(*use assms* **in** ‹*clarsimp simp*: *Sup-le-iff ac-simps*›; *fast intro*!: *exI imageI Sup-upper le-supI2*)

lemma *closed-strict-complete-distrib-lattice-axiomI*[*intro*]:
assumes $cl\ \bot = \bot$
assumes $\forall A{\in}A.\ \forall x{\in}A.\ x \in closed$
shows $(\bigsqcap X{\in}A.\ \bigsqcup X)$
$\leq \bigsqcup (Inf\ `\ \{x.\ (\exists f.\ (\forall x.\ (\forall x{\in}x.\ x \in closed) \longrightarrow f\ x \in closed)$
$\wedge\ x = f\ `\ A \wedge (\forall Y{\in}A.\ f\ Y \in Y)) \wedge (\forall x{\in}x.\ x \in closed)\})$
using *closed-complete-distrib-lattice-axiomI*[*simplified assms*(*1*), *simplified*, *OF assms*(*2*)] **.**

end

5.2 Closures over powersets

locale *closure-powerset* =
closure-complete-lattice-class cl **for** $cl :: 'a\ set \Rightarrow 'a\ set$
begin

lemmas $expansive' = subsetD[OF\ expansive]$

lemma *closedI*[*intro*]:
assumes $\bigwedge x.\ x \in cl\ X \Longrightarrow x \in X$
shows $X \in closed$
unfolding *closed-def* **using** *assms* **by** *blast*

lemma *closedE*:
assumes $x \in cl\ Y$
assumes $Y \in closed$
shows $x \in Y$
using *assms closed-def* **by** *auto*

lemma *cl-mono*:
assumes $x \in cl\ X$
assumes $X \subseteq Y$
shows $x \in cl\ Y$
using *assms* **by** (*rule subsetD*[*OF monotoneD*[*OF monotone-cl*], *rotated*])

lemma *cl-bind-le*:
shows $X \gg\!= cl \circ f \leq cl\ (X \gg\!= f)$
by (*metis bind-UNION bind-image Sup-cl-le*)

lemma *pointwise-distributive-iff*:
shows $(\forall X.\ cl\ (\bigcup X) = \bigcup (cl\ `\ X) \cup cl\ \{\}) \longleftrightarrow (\forall X.\ cl\ X = (\bigcup x{\in}X.\ cl\ \{x\}) \cup cl\ \{\})$ (**is** *?lhs* $\longleftrightarrow$ *?rhs*)
proof(*rule iffI*)
show *?lhs* $\Longrightarrow$ *?rhs* **by** (*metis UN-singleton image-image*)
next
assume *?rhs*
note *distributive* = ‹*?rhs*›[*rule-format*]
have *cl-union*: $cl\ (X \cup Y) = cl\ X \cup cl\ Y$ (**is** *?lhs1* = *?rhs1*) **for** *X Y*
proof(*rule antisym*[*OF - sup-cl-le*])
from *cl expansive'* **show** *?lhs1* $\subseteq$ *?rhs1* **by** (*subst distributive*) *auto*
qed

have *cl-insert*: *cl* (*insert x X*) = *cl* {*x*} ∪ *cl X* **for** *x X*
by (*metis cl-union insert-is-Un*)
show *?lhs*
proof(*intro allI antisym*)
show *cl* (⋃ *X*) ⊆ ⋃ (*cl* ‘ *X*) ∪ *cl* {} **for** *X*
by (*subst distributive*) (*clarsimp*; *metis UnCI cl-insert insert-Diff*)
qed (*simp add*: *cl-bot-least Sup-cl-le*)
qed

lemma *Sup-prime-on-singleton*:
shows *Sup-prime-on closed* (*cl* {*x*})
unfolding *Sup-prime-on-def* **by** (*meson UnionE expansive′ insert-subsetI least bot-least singletonI subsetD*)

end

locale *closure-powerset-algebraic* =
closure-powerset
+ *closure-complete-lattice-algebraic-class*

locale *closure-powerset-distributive* =
closure-powerset
+ *closure-complete-distrib-lattice-distributive-class*
begin

lemmas *distributive* = *pointwise-distributive-iff*[*THEN iffD1*, *rule-format*, *OF cl-Sup*]

lemma *algebraic-axiom*: — Davey and Priestley (2002, Theorem 7.14)
shows *cl x* ⊆ ⋃ (*cl* ‘ ({*y*. *y* ⊆ *x*} ∩ *local.compact-points*))
unfolding *compact-points-def complete-lattice-class.compact-points-def*[*symmetric*]
by (*metis Int-iff algebraic cl-Sup-not-empty complete-lattice-class.compact-pointsI emptyE*
finite.intros(*1*) *mem-Collect-eq order-bot-class.bot-least order.refl*)

lemma *cl-insert*:
shows *cl* (*insert x X*) = *cl* {*x*} ∪ *cl X*
by (*metis cl-sup insert-is-Un*)

lemma *cl-UNION*:
shows *cl* (⋃*i*∈*I*. *f i*) = (⋃*i*∈*I*. *cl* (*f i*)) ∪ *cl* {}
by (*auto simp add*: *cl-Sup SUP-upper*)

lemma *closed-UNION*:
assumes ⋀*i*. *i* ∈ *I* ⟹ *f i* ∈ *closed*
shows (⋃*i*∈*I*. *f i*) ∪ *cl* {} ∈ *closed*
using *assms closed-def* **by** (*auto simp*: *cl-Sup cl-sup*)

lemma *sort-of-inverse*: — Pfaltz and Šlapal (2013, Proposition 2.5)
assumes *y* ∈ *cl X* − *cl* {}
shows ∃ *x*∈*X*. *y* ∈ *cl* {*x*}
using *assms* **by** (*subst* (*asm*) *distributive*) *blast*

lemma *cl-diff-le*:
shows *cl x* − *cl y* ⊆ *cl* (*x* − *y*)
by (*metis Diff-subset-conv Un-Diff-cancel Un-upper2 cl-sup*)

lemma *cl-bind*:
shows *cl* (*X* ⤜ *f*) = (*X* ⤜ *cl* ∘ *f*) ∪ *cl* {}
by (*simp add*: *bind-UNION cl-Sup*)

lemma *sup-irreducible-on-singleton*:
shows *sup-irreducible-on closed* (*cl* {*a*})
by (*rule sup-irreducible-onI*)
(*metis Un-iff sup-bot.right-neutral expansive insert-subset least antisym local.sup.commute sup-ge2*)

end

5.3 Matroids and antimatroids

The *exchange* axiom characterises *matroids* (see, for instance, §6.1), while the *anti-exchange* axiom characterises *antimatroids* (see e.g. §7.1).

References:

- Pfaltz and Šlapal (2013) provide an overview of these concepts
- https://en.wikipedia.org/wiki/Antimatroid

definition *anti-exchange* :: ('*a set* ⇒ '*a set*) ⇒ *bool* **where**
anti-exchange cl ⟷ (∀ *X x y*. *x* ≠ *y* ∧ *y* ∈ *cl* (*insert x X*) − *cl X* ⟶ *x* ∉ *cl* (*insert y X*) − *cl X*)

definition *exchange* :: ('*a set* ⇒ '*a set*) ⇒ *bool* **where**
exchange cl ⟷ (∀ *X x y*. *y* ∈ *cl* (*insert x X*) − *cl X* ⟶ *x* ∈ *cl* (*insert y X*) − *cl X*)

lemmas *anti-exchangeI* = *iffD2*[*OF anti-exchange-def*, *rule-format*]
lemmas *exchangeI* = *iffD2*[*OF exchange-def*, *rule-format*]

lemma *anti-exchangeD*:
assumes *y* ∈ *cl* (*insert x X*) − *cl X*
assumes *x* ≠ *y*
assumes *anti-exchange cl*
shows *x* ∉ *cl* (*insert y X*) − *cl X*
using *assms* **unfolding** *anti-exchange-def* **by** *blast*

lemma *exchange-Image*: — Some matroids arise from equivalence relations. Note *sym r* ∧ *trans r* ⟶ *Refl r*
shows *exchange* (*Image r*) ⟷ *sym r* ∧ *trans r*
by (*auto 6 0 simp*: *exchange-def sym-def intro*!: *transI elim*: *transE*)

locale *closure-powerset-distributive-exchange* =
closure-powerset-distributive
+ **assumes** *exchange*: *exchange cl*
begin

lemma *exchange-exchange*:
assumes *x* ∈ *cl* {*y*}
assumes *x* ∉ *cl* {}
shows *y* ∈ *cl* {*x*}
using *assms exchange* **unfolding** *exchange-def* **by** *blast*

lemma *exchange-closed-inter*:
assumes *Q* ∈ *closed*
shows *cl P* ∩ *Q* = *cl* (*P* ∩ *Q*) (**is** *?lhs* = *?rhs*)
and *Q* ∩ *cl P* = *cl* (*P* ∩ *Q*) (**is** *?thesis1*)
proof −
show *?lhs* = *?rhs*
proof(*rule antisym*)
have ((⋃*x*∈*P*. *cl* {*x*}) ∪ *cl* {}) ∩ *Q* ⊆ (⋃*x*∈*P* ∩ *cl Q*. *cl* {*x*}) ∪ *cl* {}
by *clarsimp* (*metis IntI UnCI cl-insert exchange-exchange mk-disjoint-insert*)

then show *?lhs* ⊆ *?rhs*
by (*simp flip*: *distributive closed-conv*[*OF assms*])
from *assms* **show** *?rhs* ⊆ *?lhs*
using *cl-inf-le closed-conv* **by** *blast*
qed
then show *?thesis1* **by** *blast*
qed

lemma *exchange-both-closed-inter*:
assumes $P \in$ *closed*
assumes $Q \in$ *closed*
shows *cl* $(P \cap Q) = P \cap Q$
using *assms closed-conv closed-inf* **by** *force*

end

lemma *anti-exchange-Image*: — when r is asymmetric on distinct points
shows *anti-exchange* (*Image r*) $\longleftrightarrow$ $(\forall\, x\ y.\ x \neq y \wedge (x,\ y) \in r \longrightarrow (y,\ x) \notin r)$
by (*auto simp*: *anti-exchange-def*)

locale *closure-powerset-distributive-anti-exchange* =
closure-powerset-distributive
+ **assumes** *anti-exchange*: *anti-exchange cl*

5.4 Composition

Conditions under which composing two closures yields a closure. See also Pfaltz and Šlapal (2013).

lemma *closure-comp*:
assumes *closure lesseqa lessa* cl_1
assumes *closure lesseqa lessa* cl_2
assumes $\bigwedge X.\ cl_1\ (cl_2\ X) = cl_2\ (cl_1\ X)$
shows *closure lesseqa lessa* $(\lambda X.\ cl_1\ (cl_2\ X))$
using *assms* **by** (*clarsimp simp*: *closure-def closure-axioms-def*) *metis*

lemma *closure-complete-lattice-comp*:
assumes *closure-complete-lattice Infa Supa infa lesseqa lessa supa bota topa* cl_1
assumes *closure-complete-lattice Infa Supa infa lesseqa lessa supa bota topa* cl_2
assumes $\bigwedge X.\ cl_1\ (cl_2\ X) = cl_2\ (cl_1\ X)$
shows *closure-complete-lattice Infa Supa infa lesseqa lessa supa bota topa* $(\lambda X.\ cl_1\ (cl_2\ X))$
using *assms*(*1*)[*unfolded closure-complete-lattice-def*]
closure-comp[*OF closure-complete-lattice.axioms*(*2*)[*OF assms*(*1*)]
closure-complete-lattice.axioms(*2*)[*OF assms*(*2*)]]
assms(*3*)
by (*blast intro*: *closure-complete-lattice.intro*)

lemma *closure-powerset-comp*:
assumes *closure-powerset* cl_1
assumes *closure-powerset* cl_2
assumes $\bigwedge X.\ cl_1\ (cl_2\ X) = cl_2\ (cl_1\ X)$
shows *closure-powerset* $(\lambda X.\ cl_1\ (cl_2\ X))$
using *assms* **by** (*simp add*: *closure-complete-lattice-class-def closure-complete-lattice-comp closure-powerset-def*)

lemma *closure-powerset-distributive-comp*:
assumes *closure-powerset-distributive* cl_1
assumes *closure-powerset-distributive* cl_2
assumes $\bigwedge X.\ cl_1\ (cl_2\ X) = cl_2\ (cl_1\ X)$
shows *closure-powerset-distributive* $(\lambda X.\ cl_1\ (cl_2\ X))$
proof −

have $cl_1\ (cl_2\ (\bigcup\ X)) \subseteq (\bigcup X{\in}X.\ cl_1\ (cl_2\ X)) \cup cl_1\ (cl_2\ \{\})$ **for** X
apply (*subst* (*1 2 3*) *closure-powerset-distributive.distributive*[*OF assms*(*1*)])
apply (*subst* (*1 2 3*) *closure-powerset-distributive.distributive*[*OF assms*(*2*)])
apply *fast*
done
moreover
from *assms* **have** *closure-axioms* $(\subseteq)$ $(\lambda X.\ cl_1\ (cl_2\ X))$
by (*metis closure-powerset-distributive-def closure-complete-lattice-class-def closure-def closure-powerset.axioms closure-powerset-comp closure-complete-lattice.axioms*(*2*))
ultimately show *?thesis*
by (*intro-locales*; *blast intro*: *closure-complete-lattice-distributive-axioms.intro*)
qed

5.5 Path independence

Pfaltz and Šlapal (2013, Prop 1.1): "an expansive operator is a closure operator iff it is path independent."
References:

- $AFP/Stable_Matching/Choice_Functions.thy

context *semilattice-sup*
begin

definition *path-independent* :: $('a \Rightarrow 'a) \Rightarrow bool$ **where**
path-independent $f \longleftrightarrow (\forall\, x\ y.\ f\ (x \sqcup y) = f\ (f\ x \sqcup f\ y))$

lemma *cl-path-independent*:
shows *closure* $(\leq)$ $(<)$ *cl* $\longleftrightarrow$ *path-independent cl* $\wedge$ $(\forall\, x.\ x \leq cl\ x)$ (**is** *?lhs* $\longleftrightarrow$ *?rhs*)
proof(*rule iffI*)
show *?lhs* $\Longrightarrow$ *?rhs*
by (*auto 5 0 simp*: *closure-def closure-axioms-def path-independent-def order.eq-iff elim*: *le-supE*)
show *?rhs* $\Longrightarrow$ *?lhs*
by *unfold-locales* (*metis path-independent-def le-sup-iff sup.absorb-iff2 sup.idem*)
qed

end

5.6 Some closures

interpretation *id-cl*: *closure-powerset-distributive id*
by *standard auto*

5.6.1 Reflexive, symmetric and transitive closures

The reflexive closure *reflcl* is very well behaved. Note the new bottom is *Id*. The reflexive transitive closure *rtrancl* and transitive closure *trancl* are clearly not distributive.
rtrancl is neither matroidal nor antimatroidal.

interpretation *reflcl-cl*: *closure-powerset-distributive-exchange reflcl*
by *standard* (*auto simp*: *exchange-def*)

interpretation *symcl-cl*: *closure-powerset-distributive-exchange* $\lambda X.\ X \cup X^{-1}$
by *standard* (*auto simp*: *exchange-def*)

interpretation *trancl-cl*: *closure-powerset trancl*
by *standard* (*metis r-into-trancl' subsetI trancl-id trancl-mono trans-trancl*)

interpretation *rtrancl-cl*: *closure-powerset rtrancl*

by *standard* (*use rtrancl-subset-rtrancl* **in** *blast*)

lemma *rtrancl-closed-Id*:
shows $Id \in$ *rtrancl-cl.closed*
using *rtrancl-cl.idempotent rtrancl-empty* **by** *fastforce*

lemma *rtrancl-closed-reflcl-closed*:
shows *rtrancl-cl.closed* $\subseteq$ *reflcl-cl.closed*
using *rtrancl-cl.closed-conv* **by** *fast*

5.6.2 Relation image

lemma *idempotent-Image*:
assumes $r \subseteq Y \times Y$
assumes *refl-on Y r*
assumes *trans r*
assumes $X \subseteq Y$
shows $r `` r `` X = r `` X$
using *assms*
by (*auto elim*: *transE intro*: *refl-onD refl-onD*)

lemmas *distributive-Image* = *Image-eq-UN*

lemma *closure-powerset-distributive-ImageI*:
assumes *cl* = *Image r*
assumes *refl r*
assumes *trans r*
shows *closure-powerset-distributive cl*
proof −
have *closure-axioms* ($\subseteq$) *cl*
unfolding *order.closure-axioms-alt-def*
proof (*intro conjI*)
show $\forall x.\ x \subseteq cl\ x$
unfolding ‹*cl* = *Image r*›
using ‹*refl r*›
by (*metis subset-refl Image-Id refl-alt-def*[*of r*] *Image-mono*[*of Id r*])
next
show *mono cl*
unfolding ‹*cl* = *Image r*›
using *mono-Image* **.**
next
show $\forall x.\ cl\ (cl\ x) = cl\ x$
unfolding ‹*cl* = *Image r*›
using ‹*refl r*› ‹*trans r*›
by (*metis top-greatest top-greatest*[*of r*] *UNIV-Times-UNIV idempotent-Image*[*of r UNIV*])
qed
with ‹*cl* = *Image r*› **show** *?thesis*
by − (*intro-locales*; *auto simp*: *closure-complete-lattice-distributive-axioms-def*)
qed

lemma *closure-powerset-distributive-exchange-ImageI*:
assumes *cl* = *Image r*
assumes *equiv UNIV r* — symmetric, transitive and universal domain
shows *closure-powerset-distributive-exchange cl*
using *closure-powerset-distributive-ImageI*[*OF assms*(*1*)] *exchange-Image*[*of r*] *assms*
unfolding *closure-powerset-distributive-exchange-def closure-powerset-distributive-exchange-axioms-def*
by (*metis equivE*)

interpretation *Image-rtrancl*: *closure-powerset-distributive Image* (r^*)
by (*rule closure-powerset-distributive-ImageI*) *auto*

5.6.3 Kleene closure

We define Kleene closure in the traditional way with respect to some axioms that our various lattices satisfy. As trace models are not going to validate $x \cdot \bot = \bot$ (Kozen 1994, Axiom 13), we cannot reuse existing developments of Kleene Algebra (and Concurrent Kleene Algebra (Hoare, Möller, Struth, and Wehrman 2011)). In general it is not distributive.

locale *weak-kleene* =
  fixes *unit* :: $'a$::*complete-lattice* ($\langle\varepsilon\rangle$)
  fixes *comp* :: $'a \Rightarrow 'a \Rightarrow 'a$ (**infixl** $\langle\cdot\rangle$ *60*)
  assumes *comp-assoc*: $(x \cdot y) \cdot z = x \cdot (y \cdot z)$
  assumes *weak-comp-unitL*: $\varepsilon \leq x \Longrightarrow \varepsilon \cdot x = x$
  assumes *comp-unitR*: $x \cdot \varepsilon = x$
  assumes *comp-supL*: $(x \sqcup y) \cdot z = (x \cdot z) \sqcup (y \cdot z)$
  assumes *comp-supR*: $x \cdot (y \sqcup z) = (x \cdot y) \sqcup (x \cdot z)$
  assumes *mcont-compL*: *mcont Sup* ($\leq$) *Sup* ($\leq$) ($\lambda x.\ x \cdot y$)
  assumes *mcont-compR*: *mcont Sup* ($\leq$) *Sup* ($\leq$) ($\lambda y.\ x \cdot y$)
  assumes *comp-botL*: $\bot \cdot x = \bot$
begin

lemma *mcont2mcont-comp*:
  assumes *mcont Supa orda Sup* ($\leq$) *f*
  assumes *mcont Supa orda Sup* ($\leq$) *g*
  shows *mcont Supa orda Sup* ($\leq$) ($\lambda x.\ f\ x \cdot g\ x$)
by (*simp add*: *ccpo.mcont2mcont'*[*OF complete-lattice-ccpo mcont-compL - assms*(*1*)]
        ccpo.mcont2mcont'[*OF complete-lattice-ccpo mcont-compR - assms*(*2*)])

lemma *mono2mono-comp*:
  assumes *monotone orda* ($\leq$) *f*
  assumes *monotone orda* ($\leq$) *g*
  shows *monotone orda* ($\leq$) ($\lambda x.\ f\ x \cdot g\ x$)
using *mcont-mono*[*OF mcont-compL*] *mcont-mono*[*OF mcont-compR*] *assms*
by (*clarsimp simp*: *monotone-def*) (*meson order.trans*)

context
  notes *mcont2mcont-comp*[*cont-intro*]
  notes *mono2mono-comp*[*cont-intro*, *partial-function-mono*]
  notes *st-monotone*[*OF mcont-mono*[*OF mcont-compL*], *strg*]
  notes *st-monotone*[*OF mcont-mono*[*OF mcont-compR*], *strg*]
begin

context
  notes [[*function-internals*]] — Exposes the induction rules we need
begin

partial-function (*lfp*) *star* :: $'a \Rightarrow 'a$ **where**
  $star\ x = (x \cdot star\ x) \sqcup \varepsilon$

partial-function (*lfp*) *rev-star* :: $'a \Rightarrow 'a$ **where**
  $rev\text{-}star\ x = (rev\text{-}star\ x \cdot x) \sqcup \varepsilon$

end

lemmas *parallel-star-induct-1-1* =
  parallel-fixp-induct-1-1[*OF*
    complete-lattice-partial-function-definitions complete-lattice-partial-function-definitions

star.mono star.mono star-def star-def]

lemma *star-bot*:
shows *star* $\bot = \varepsilon$
by (*subst star.simps*) (*simp add*: *comp-botL*)

lemma *epsilon-star-le*:
shows $\varepsilon \leq$ *star P*
by (*subst star.simps*) *simp*

lemma *monotone-star*:
shows *mono star*
proof(*rule monotoneI*)
fix *x y* :: $'a$
assume $x \leq y$ **show** *star x* $\leq$ *star y*
proof(*induct rule*: *star.fixp-induct*[*case-names adm bot step*])
case (*step R*) **show** *?case*
apply (*strengthen ord-to-strengthen*(*1*)[*OF* ‹$x \leq y$›])
apply (*strengthen ord-to-strengthen*(*1*)[*OF step*])
apply (*rule order.refl*)
done
qed *simp-all*
qed

lemma *expansive-star*:
shows $x \leq$ *star x*
by (*subst star.simps*, *subst star.simps*)
(*simp add*: *comp-supL comp-supR comp-unitR le-supI1 flip*: *sup.assoc*)

lemma *star-comp-star*:
shows *star x* · *star x* = *star x* (**is** *?lhs* = *?rhs*)
proof(*rule antisym*)
show *?lhs* $\leq$ *?rhs*
proof(*induct rule*: *star.fixp-induct*[**where** *P*=$\lambda R.$ *R x* · *star x* $\leq$ *star x*, *case-names adm bot step*])
case (*step R*) **show** *?case*
apply (*simp add*: *comp-supL weak-comp-unitL*[*OF epsilon-star-le*] *comp-assoc*)
apply (*strengthen ord-to-strengthen*(*1*)[*OF step*])
apply (*subst* (*2*) *star.simps*)
apply *simp*
done
qed (*simp-all add*: *comp-botL*)
show *?rhs* $\leq$ *?lhs*
by (*strengthen ord-to-strengthen*(*2*)[*OF epsilon-star-le*])
(*simp add*: *weak-comp-unitL*[*OF epsilon-star-le*])
qed

lemma *idempotent-star*:
shows *star* (*star x*) = *star x* (**is** *?lhs* = *?rhs*)
proof(*rule antisym*)
show *?lhs* $\leq$ *?rhs*
proof(*induct rule*: *star.fixp-induct*[**where** *P*=$\lambda R.$ *R* (*star x*) $\leq$ *?rhs*, *case-names adm bot step*])
next
case (*step R*) **then show** *?case*
by (*metis comp-supR epsilon-star-le le-iff-sup le-sup-iff star-comp-star*)
qed *simp-all*
show *?rhs* $\leq$ *?lhs*
by (*simp add*: *expansive-star*)
qed

interpretation *star*: *closure-complete-lattice-class star*
by *standard* (*metis order.trans expansive-star idempotent-star monotoneD*[*OF monotone-star*])

lemma *star-epsilon*:
shows *star* $\varepsilon = \varepsilon$
by (*metis idempotent-star star-bot*)

lemma *epsilon-rev-star-le*:
shows $\varepsilon \leq$ *rev-star P*
by (*subst rev-star.simps*) *simp*

lemma *rev-star-comp-rev-star*:
shows *rev-star x* $\cdot$ *rev-star x* = *rev-star x* (**is** *?lhs* = *?rhs*)
proof(*rule antisym*)
show *?lhs* $\leq$ *?rhs*
proof(*induct rule*: *rev-star.fixp-induct*[**where** *P*=λ*R. rev-star x* $\cdot$ *R x* $\leq$ *rev-star x*, *case-names adm bot step*])
case *bot* **show** *?case*
by (*subst* (*2*) *rev-star.simps*) (*simp add*: *le-supI1 mcont-monoD*[*OF mcont-compR*])
next
case (*step R*) **then show** *?case*
by (*simp add*: *comp-supR epsilon-rev-star-le comp-unitR flip*: *comp-assoc*)
(*metis comp-supL le-iff-sup le-supI1 rev-star.simps*)
qed *simp*
show *?rhs* $\leq$ *?lhs*
by (*metis comp-supR comp-unitR epsilon-rev-star-le le-iff-sup*)
qed

lemma *star-rev-star*:
shows *star* = *rev-star* (**is** *?lhs* = *?rhs*)
proof(*intro fun-eqI antisym*)
show *?lhs P* $\leq$ *?rhs P* **for** *P*
proof(*induct rule*: *star.fixp-induct*[*case-names adm bot step*])
case (*step R*)
have *expansive*: *x* $\leq$ *rev-star x* **for** *x*
apply (*subst rev-star.simps*)
apply (*subst rev-star.simps*)
apply (*simp add*: *comp-supL sup-assoc*)
apply (*metis comp-supR comp-unitR sup-ge2 le-supI2 sup-commute weak-comp-unitL*)
done
show *?case*
by (*strengthen ord-to-strengthen*(*1*)[*OF step*])
(*metis comp-supL epsilon-rev-star-le expansive rev-star-comp-rev-star le-sup-iff sup.absorb-iff2*)
qed *simp-all*
show *?rhs P* $\leq$ *?lhs P* **for** *P*
proof(*induct rule*: *rev-star.fixp-induct*[*case-names adm bot step*])
case (*step R*) **show** *?case*
by (*strengthen ord-to-strengthen*(*1*)[*OF step*])
(*metis epsilon-star-le expansive-star mcont-monoD*[*OF mcont-compR*] *le-supI star-comp-star*)
qed *simp-all*
qed

lemmas *star-fixp-rev-induct* = *rev-star.fixp-induct*[*folded star-rev-star*]

interpretation *rev-star*: *closure-complete-lattice-class rev-star*
by (*rule star.closure-complete-lattice-class-axioms*[*unfolded star-rev-star*])

lemma *rev-star-bot*:

shows *rev-star* $\bot = \varepsilon$
by (*simp add*: *star-bot flip*: *star-rev-star*)

lemma *rev-star-epsilon*:
shows *rev-star* $\varepsilon = \varepsilon$
by (*simp add*: *star-epsilon flip*: *star-rev-star*)

lemmas *star-unfoldL* = *star.simps*

lemma *star-unfoldR*:
shows *star* $x = (star\ x \cdot x) \sqcup \varepsilon$
by (*subst* (*1 2*) *star-rev-star*) (*simp flip*: *rev-star.simps*)

lemmas *rev-star-unfoldR* = *rev-star.simps*

lemma *rev-star-unfoldL*:
shows *rev-star* $x = (x \cdot rev\text{-}star\ x) \sqcup \varepsilon$
by (*simp flip*: *star-rev-star star-unfoldL*)

lemma *fold-starL*:
shows $x \cdot star\ x \leq star\ x$
by (*subst* (*2*) *star.simps*) *simp*

lemma *fold-starR*:
shows $star\ x \cdot x \leq star\ x$
by (*metis inf-sup-ord*(*3*) *star-unfoldR*)

lemma *fold-rev-starL*:
shows $x \cdot rev\text{-}star\ x \leq rev\text{-}star\ x$
by (*simp add*: *fold-starL flip*: *star-rev-star*)

lemma *fold-rev-starR*:
shows $rev\text{-}star\ x \cdot x \leq rev\text{-}star\ x$
by (*simp add*: *fold-starR flip*: *star-rev-star*)

declare *star.strengthen-cl*[*strg*] *rev-star.strengthen-cl*[*strg*]

end

end

locale *kleene* = *weak-kleene* +
assumes *comp-unitL*: $\varepsilon \cdot x = x$ — satisfied by ('*a*, '*s*, '*v*) *prog* but not ('*a*, '*s*, '*v*) *spec*

6 Galois connections

Here we collect some classical results for Galois connections. These are drawn from Backhouse (2000); Davey and Priestley (2002); Melton, Schmidt, and Strecker (1985); Müller-Olm (1997) amongst others. The canonical reference is likely Gierz, Hofmann, Keimel, Lawson, Mislove, and Scott (2003).

Our focus is on constructing closures (§5) conveniently; we are less interested in the fixed-point story. Many of these results hold for preorders; we simply work with partial orders (via the *ordering* locale). Similarly *conditionally complete lattices* are often sufficient, but for convenience we just assume (unconditional) completeness.

locale *galois* =
orda: *ordering less-eqa lessa*
+ *ordb*: *ordering less-eqb lessb*
for *less-eqa* (**infix** $\langle \leq_a \rangle$ *50*)
and *lessa* (**infix** $\langle <_a \rangle$ *50*)

and *less-eqb* (**infix** ‹$\leq_b$› *50*)
and *lessb* (**infix** ‹$<_b$› *50*)
+ **fixes** *lower* :: $'a \Rightarrow 'b$
fixes *upper* :: $'b \Rightarrow 'a$
assumes *galois*: $lower\ x \leq_b y \longleftrightarrow x \leq_a upper\ y$
begin

lemma *monotone-lower*:
shows *monotone* ($\leq_a$) ($\leq_b$) *lower*
by (*rule monotoneI*) (*use galois orda.trans ordb.refl* **in** *blast*)

lemma *monotone-upper*:
shows *monotone* ($\leq_b$) ($\leq_a$) *upper*
by (*rule monotoneI*) (*use galois ordb.trans orda.refl* **in** *blast*)

lemmas *strengthen-lower*[*strg*] = *st-monotone*[*OF monotone-lower*]
lemmas *strengthen-upper*[*strg*] = *st-monotone*[*OF monotone-upper*]

lemma *upper-lower-expansive*:
shows $x \leq_a upper\ (lower\ x)$
using *galois ordb.refl* **by** *blast*

lemma *lower-upper-contractive*:
shows $lower\ (upper\ x) \leq_b x$
using *galois orda.refl* **by** *blast*

lemma *comp-galois*: — Backhouse (2000, Lemma 19). Observe that the roles of upper and lower have swapped.
fixes *less-eqc* :: $'c \Rightarrow 'c \Rightarrow bool$ (**infix** ‹$\leq_c$› *50*)
fixes *lessc* :: $'c \Rightarrow 'c \Rightarrow bool$ (**infix** ‹$<_c$› *50*)
fixes $h :: 'a \Rightarrow 'c$
fixes $k :: 'b \Rightarrow 'c$
assumes *partial-preordering* ($\leq_c$)
assumes *monotone* ($\leq_a$) ($\leq_c$) h
assumes *monotone* ($\leq_b$) ($\leq_c$) k
shows $(\forall x.\ h\ (upper\ x) \leq_c k\ x) \longleftrightarrow (\forall x.\ h\ x \leq_c k\ (lower\ x))$
using *assms*(*1*) *monotoneD*[*OF assms*(*2*)] *monotoneD*[*OF assms*(*3*)]
by (*meson lower-upper-contractive partial-preordering.trans upper-lower-expansive*)

lemma *lower-upper-le-iff*: — Backhouse (2000, Lemma 23)
assumes $\forall x\ y.\ lower'\ x \leq_b y \longleftrightarrow x \leq_a upper'\ y$
shows $(\forall x.\ lower'\ x \leq_b lower\ x) \longleftrightarrow (\forall y.\ upper\ y \leq_a upper'\ y)$
using *assms* **by** (*meson lower-upper-contractive orda.trans ordb.trans upper-lower-expansive*)

lemma *lower-upper-unique*: — Backhouse (2000, Lemma 24)
assumes $\forall x\ y.\ lower'\ x \leq_b y \longleftrightarrow x \leq_a upper'\ y$
shows $lower' = lower \longleftrightarrow upper' = upper$
using *assms galois lower-upper-contractive orda.eq-iff ordb.eq-iff upper-lower-expansive* **by** *blast*

lemma *upper-lower-idem*:
shows $upper\ (lower\ (upper\ (lower\ x))) = upper\ (lower\ x)$
by (*meson galois lower-upper-contractive orda.antisym ordb.trans upper-lower-expansive*)

lemma *lower-upper-idem*:
shows $lower\ (upper\ (lower\ (upper\ x))) = lower\ (upper\ x)$
by (*metis galois ordb.antisym upper-lower-expansive upper-lower-idem*)

lemma *lower-upper-lower*: — Melton et al. (1985, Proposition 1.2(2))
shows $lower \circ upper \circ lower = lower$

and *lower* (*upper* (*lower x*)) = *lower x*
using *galois lower-upper-contractive ordb.antisym upper-lower-expansive upper-lower-idem* **by** *auto*

lemma *upper-lower-upper*: — Melton et al. (1985, Proposition 1.2(2))
shows *upper* ∘ *lower* ∘ *upper* = *upper*
and *upper* (*lower* (*upper x*)) = *upper x*
by (*simp-all add*: *fun-eq-iff*)
(*metis galois monotone-upper monotoneD orda.antisym orda.refl upper-lower-expansive*)+

definition *cl* :: *'a* ⇒ *'a* **where** — The opposite composition yields a kernel operator
cl x = *upper* (*lower x*)

lemma *cl-axiom*:
shows (*x* ≤$_a$ *cl y*) = (*cl x* ≤$_a$ *cl y*)
by (*metis cl-def galois lower-upper-lower*(*2*))

sublocale *closure* (≤$_a$) (<$_a$) *cl* — incorporates definitions and lemmas into this namespace
by *standard* (*rule cl-axiom*)

lemma *cl-upper*:
shows *cl* (*upper P*) = *upper P*
by (*simp add*: *cl-def upper-lower-upper*)

lemma *closed-upper*:
shows *upper P* ∈ *closed*
by (*simp add*: *closed-def cl-upper orda.refl*)

lemma *inj-lower-iff-surj-upper*:
shows *inj lower* ⟷ *surj upper*
by (*metis inj-def surj-def lower-upper-lower*(*2*) *upper-lower-upper*(*2*))

lemma *inj-lower-iff-upper-lower-id*:
shows *inj lower* ⟷ *upper* ∘ *lower* = *id*
by (*metis fun.map-comp id-comp inj-iff inj-on-id inj-on-imageI2 lower-upper-lower*(*1*))

lemma *upper-inj-iff-surj-lower*:
shows *inj upper* ⟷ *surj lower*
by (*metis inj-def surj-def lower-upper-lower*(*2*) *upper-lower-upper*(*2*))

lemma *inj-upper-iff-lower-upper-id*:
shows *inj upper* ⟷ *lower* ∘ *upper* = *id*
by (*metis fun.map-comp id-comp inj-iff inj-on-id inj-on-imageI2 upper-lower-upper*(*1*))

lemma *lower-downset-upper*: — Davey and Priestley (2002, Lemma 7.32): inverse image of lower on a downset is the downset of upper
shows *lower* −‘ {*a*. *a* ≤$_b$ *y*} = {*a*. *a* ≤$_a$ *upper y*}
by (*simp add*: *galois*)

lemma *lower-downset*: — Davey and Priestley (2002, Lemma 7.32); equivalent to the Galois axiom
shows ∃!*x*. *lower* −‘ {*a*. *a* ≤$_b$ *y*} = {*a*. *a* ≤$_a$ *x*}
by (*metis lower-downset-upper mem-Collect-eq orda.antisym orda.refl*)

end

setup ‹*Sign.mandatory-path galois*›

lemma *axioms-alt*:
fixes *less-eqa* (**infix** ‹≤$_a$› *50*)

fixes *less-eqb* (**infix** ‹$\leq_b$› *50*)
fixes *lower* :: *'a* ⇒ *'b*
fixes *upper* :: *'b* ⇒ *'a*
assumes *oa*: *ordering less-eqa lessa*
assumes *ob*: *ordering less-eqb lessb*
assumes *ul*: ∀ *x*. *x* $\leq_a$ *upper* (*lower x*)
assumes *lu*: ∀ *x*. *lower* (*upper x*) $\leq_b$ *x*
assumes *ml*: *monotone* ($\leq_a$) ($\leq_b$) *lower*
assumes *mu*: *monotone* ($\leq_b$) ($\leq_a$) *upper*
shows *lower x* $\leq_b$ *y* ⟷ *x* $\leq_a$ *upper y*
by (*metis lu ml monotoneD mu oa ob ordering.axioms*(*1*) *partial-preordering.trans ul*)

lemma *compose*:
fixes *lower*$_1$ :: *'b* ⇒ *'c*
fixes *lower*$_2$ :: *'a* ⇒ *'b*
fixes *less-eqa* :: *'a* ⇒ *'a* ⇒ *bool*
assumes *galois less-eqb lessb less-eqc lessc lower*$_1$ *upper*$_1$
assumes *galois less-eqa lessa less-eqb lessb lower*$_2$ *upper*$_2$
shows *galois less-eqa lessa less-eqc lessc* (*lower*$_1$ ∘ *lower*$_2$) (*upper*$_2$ ∘ *upper*$_1$)
using *assms* **unfolding** *galois-def galois-axioms-def* **by** *simp*

locale *complete-lattice* =
cla: *complete-lattice Inf*$_a$ *Sup*$_a$ ($\sqcap_a$) ($\leq_a$) ($<_a$) ($\sqcup_a$) $\bot_a$ $\top_a$
+ *clb*: *complete-lattice Inf*$_b$ *Sup*$_b$ ($\sqcap_b$) ($\leq_b$) ($<_b$) ($\sqcup_b$) $\bot_b$ $\top_b$
+ *galois* ($\leq_a$) ($<_a$) ($\leq_b$) ($<_b$) *lower upper*
for *less-eqa* :: *'a* ⇒ *'a* ⇒ *bool* (**infix** ‹$\leq_a$› *50*)
and *lessa* (**infix** ‹$<_a$› *50*)
and *infa* (**infixl** ‹$\sqcap_a$› *70*)
and *supa* (**infixl** ‹$\sqcup_a$› *65*)
and *bota* (‹$\bot_a$›)
and *topa* (‹$\top_a$›)
and *Inf*$_a$ *Sup*$_a$
and *less-eqb* :: *'b* ⇒ *'b* ⇒ *bool* (**infix** ‹$\leq_b$› *50*)
and *lessb* (**infix** ‹$<_b$› *50*)
and *infb* (**infixl** ‹$\sqcap_b$› *70*)
and *supb* (**infixl** ‹$\sqcup_b$› *65*)
and *botb* (‹$\bot_b$›)
and *topb* (‹$\top_b$›)
and *Inf*$_b$ *Sup*$_b$
and *lower* :: *'a* ⇒ *'b*
and *upper* :: *'b* ⇒ *'a*
begin

lemma *lower-bot*:
shows *lower* $\bot_a$ = $\bot_b$
by (*simp add*: *clb.le-bot galois*)

lemmas *mono2mono-lower*[*cont-intro*, *partial-function-mono*] = *monotone2monotone*[*OF monotone-lower*, *simplified*]

lemma *lower-Sup*: — Melton et al. (1985, Proposition 1.2(6)): *lower* is always a distributive operation
shows *lower* (*Sup*$_a$ *X*) = *Sup*$_b$ (*lower* ‘ *X*) (**is** *?lhs* = *?rhs*)
proof(*rule clb.order.antisym*)
show *?lhs* $\leq_b$ *?rhs* **by** (*meson cla.Sup-least clb.SUP-upper galois*)
show *?rhs* $\leq_b$ *?lhs* **by** (*meson cla.Sup-le-iff clb.SUP-le-iff galois upper-lower-expansive*)
qed

lemma *lower-SUP*:

shows $lower\ (Sup_a\ (f\ `\ X)) = Sup_b\ ((\lambda x.\ lower\ (f\ x))\ `\ X)$
by (*simp add*: *lower-Sup image-image*)

lemma *lower-sup*:
shows $lower\ (X \sqcup_a Y) = lower\ X \sqcup_b lower\ Y$
using *lower-Sup*[**where** $X=\{X, Y\}$] **by** *simp*

lemma *lower-Inf-le*:
shows $lower\ (Inf_a\ X) \leq_b Inf_b\ (lower\ `\ X)$
by (*simp add*: *cla.Inf-lower2 clb.le-INF-iff galois upper-lower-expansive*)

lemma *lower-INF-le*:
shows $lower\ (Inf_a\ (f\ `\ X)) \leq_b Inf_b\ ((\lambda x.\ lower\ (f\ x))\ `\ X)$
by (*simp add*: *clb.order.trans*[*OF lower-Inf-le*] *image-image*)

lemma *lower-inf-le*:
shows $lower\ (x \sqcap_a y) \leq_b lower\ x \sqcap_b lower\ y$
using *lower-Inf-le*[**where** $X=\{x, y\}$] **by** *simp*

lemma *mcont-lower*: — Backhouse (2000): fixed point theory based on Galois connections is less general than using countable chains
shows $mcont\ Sup_a\ (\leq_a)\ Sup_b\ (\leq_b)\ lower$
by (*meson contI lower-Sup mcontI monotone-lower*)

lemma *mcont2mcont-lower*[*cont-intro*]:
assumes *mcont luba orda* $Sup_a\ (\leq_a)\ P$
shows *mcont luba orda* $Sup_b\ (\leq_b)\ (\lambda x.\ lower\ (P\ x))$
using *assms mcont-lower*
partial-function-definitions.mcont2mcont[*OF clb.complete-lattice-partial-function-definitions*]
by *blast*

lemma *upper-top*:
shows $upper\ \top_b = \top_a$
by (*simp add*: *cla.top-le flip*: *galois*)

lemma *Sup-upper-le*:
shows $Sup_a\ (upper\ `\ X) \leq_a upper\ (Sup_b\ X)$
by (*meson cla.SUP-le-iff clb.Sup-upper2 galois lower-upper-contractive*)

lemma *sup-upper-le*:
shows $upper\ x \sqcup_a upper\ y \leq_a upper\ (x \sqcup_b y)$
using *Sup-upper-le*[**where** $X=\{x, y\}$] **by** *simp*

lemma *upper-Inf*: — Melton et al. (1985, Proposition 1.2(6))
shows $upper\ (Inf_b\ X) = Inf_a\ (upper\ `\ X)$ (**is** *?lhs* = *?rhs*)
proof(*rule cla.order.antisym*)
show *?lhs* $\leq_a$ *?rhs* **by** (*meson cla.INF-greatest clb.le-Inf-iff galois lower-upper-contractive*)
show *?rhs* $\leq_a$ *?lhs* **by** (*meson cla.INF-lower clb.le-Inf-iff galois*)
qed

lemma *upper-INF*:
shows $upper\ (Inf_b\ (f\ `\ X)) = Inf_a\ ((\lambda x.\ upper\ (f\ x))\ `\ X)$
by (*simp add*: *image-image upper-Inf*)

lemma *upper-inf*:
shows $upper\ (X \sqcap_b Y) = upper\ X \sqcap_a upper\ Y$
using *upper-Inf*[**where** $X=\{X, Y\}$] **by** *simp*

In a complete lattice *lower* is determined by *upper* and vice-versa.

lemma *lower-Inf-upper*:
  shows $lower\ X = Inf_b\ \{Y.\ X \leq_a upper\ Y\}$
by (*auto simp flip*: *galois intro*: *clb.Inf-eqI*[*symmetric*])

lemma *upper-Sup-lower*:
  shows $upper\ X = Sup_a\ \{Y.\ lower\ Y \leq_b X\}$
by (*auto simp*: *galois intro*: *cla.Sup-eqI*[*symmetric*])

lemma *upper-downwards-closure-lower*: — Melton et al. (1985, Lemma 2.1)
  shows $upper\ x = Sup_a\ (lower -` \{y.\ y \leq_b x\})$
by (*simp add*: *upper-Sup-lower*)

sublocale *closure-complete-lattice* $(\leq_a)$ $(<_a)$ $(\sqcap_a)$ $(\sqcup_a)$ $\bot_a$ $\top_a$ Inf_a Sup_a *cl*
by (*rule closure-complete-lattice.intro*[*OF cla.complete-lattice-axioms closure-axioms*])

end

locale *complete-lattice-distributive* =
  galois.complete-lattice
+ **assumes** *upper-Sup-le*: $upper\ (Sup_b\ X) \leq_a Sup_a\ (upper`X)$ — Stronger than Scott continuity, which only asks for this for chain or directed X.
begin

lemma *upper-Sup*:
  shows $upper\ (Sup_b\ X) = Sup_a\ (upper`X)$
by (*simp add*: *Sup-upper-le cla.dual-order.antisym upper-Sup-le*)

lemma *upper-bot*:
  shows $upper\ \bot_b = \bot_a$
using *upper-Sup*[**where** $X=\{\}$] **by** *simp*

lemma *upper-sup*:
  shows $upper\ (x \sqcup_b y) = upper\ x \sqcup_a upper\ y$
by (*rule upper-Sup*[**where** $X=\{x, y\}$, *simplified*])

lemmas *mono2mono-upper*[*cont-intro*, *partial-function-mono*] = *monotone2monotone*[*OF monotone-upper*, *simplified*]

lemma *mcont-upper*:
  shows *mcont* Sup_b $(\leq_b)$ Sup_a $(\leq_a)$ *upper*
by (*meson contI upper-Sup mcontI monotone-upper*)

lemma *mcont2mcont-upper*[*cont-intro*]:
  assumes *mcont luba orda* Sup_b $(\leq_b)$ *P*
  shows *mcont luba orda* Sup_a $(\leq_a)$ $(\lambda x.\ upper\ (P\ x))$
by (*simp add*: *ccpo.mcont2mcont'*[*OF cla.complete-lattice-ccpo mcont-upper - assms*])

sublocale *closure-complete-lattice-distributive* $(\leq_a)$ $(<_a)$ $(\sqcap_a)$ $(\sqcup_a)$ $\bot_a$ $\top_a$ Inf_a Sup_a *cl*
by *standard* (*simp add*: *cl-def upper-Sup lower-Sup image-image*)

lemma *cl-bot*:
  shows $cl\ \bot_a = \bot_a$
by (*simp add*: *cl-def lower-bot upper-bot*)

lemma *closed-bot*[*iff*]:
  shows $\bot_a \in closed$

by (*simp add*: *cl-bot closed-clI*)

end

locale *complete-lattice-class* =
  galois.complete-lattice
    (≤) (<) (⊓) (⊔) ⊥ :: - :: *complete-lattice* ⊤ *Inf Sup*
    (≤) (<) (⊓) (⊔) ⊥ :: - :: *complete-lattice* ⊤ *Inf Sup*
begin

sublocale *closure-complete-lattice-class cl* **..**

end

locale *complete-lattice-distributive-class* =
  galois.complete-lattice-distributive
    (≤) (<) (⊓) (⊔) ⊥ :: - :: *complete-lattice* ⊤ *Inf Sup*
    (≤) (<) (⊓) (⊔) ⊥ :: - :: *complete-lattice* ⊤ *Inf Sup*
begin

sublocale *galois.complete-lattice-class* **..**
sublocale *closure-complete-lattice-distributive-class cl* **..**

end

lemma *existence-lower-preserves-Sup*: — Hoare and He (1987, p8 of Oxford TR PRG-44) amongst others
  fixes *lower* :: -::*complete-lattice* ⇒ -::*complete-lattice*
  assumes *mono lower*
  shows (∀ *x y*. *lower x* ≤ *y* ⟷ *x* ≤ ⨆{ *Y*. *lower Y* ≤ *y*}) ⟷ (∀ *X*. *lower* (⨆ *X*) ≤ ⨆(*lower* ‘ *X*)) (**is** *?lhs* ⟷ *?rhs*)
proof(*rule iffI*)
  show *?lhs* ⟹ *?rhs*
    by (*metis SUP-upper Sup-least*)
  show *?rhs* ⟹ *?lhs*
    by (*fastforce intro*: *Sup-upper SUP-least order.trans elim*: *order.trans*[*OF monoD*[*OF assms*]])
qed

lemma *lower-preserves-SupI*:
  assumes *mono lower*
  assumes ⋀*X*. *lower* (⨆ *X*) ≤ ⨆(*lower* ‘ *X*)
  assumes ⋀*x*. *upper x* = ⨆{*X*. *lower X* ≤ *x*}
  shows *galois.complete-lattice-class lower upper*
by *standard* (*metis assms galois.existence-lower-preserves-Sup*)

lemma *existence-upper-preserves-Inf*:
  fixes *upper* :: -::*complete-lattice* ⇒ -::*complete-lattice*
  assumes *mono upper*
  shows (∀ *x y*. ⨅{ *Y*. *x* ≤ *upper Y*} ≤ *y* ⟷ *x* ≤ *upper y*) ⟷ (∀ *X*. ⨅(*upper* ‘ *X*) ≤ *upper* (⨅ *X*)) (**is** *?lhs* ⟷ *?rhs*)
proof(*rule iffI*)
  assume *?lhs*
  interpret *gcl*: *galois.complete-lattice-class* λ*x*. ⨅{ *Y*. *x* ≤ *upper Y*} *upper*
    by *standard* (*use* ‹*?lhs*› **in** *blast*)
  from *gcl.upper-Inf* **show** *?rhs* **by** *simp*
next
  show *?rhs* ⟹ *?lhs*
    by (*auto intro*: *Inf-lower order.trans*[*rotated*] *INF-greatest order.trans*[*OF - monoD*[*OF assms*], *rotated*])
qed

lemma *upper-preserves-InfI*:
  assumes *mono upper*
  assumes $\bigwedge X.\ \bigsqcap (upper\ `\ X) \leq upper\ (\bigsqcap X)$
  assumes $\bigwedge x.\ lower\ x = \bigsqcap \{X.\ x \leq upper\ X\}$
  shows *galois.complete-lattice-class lower upper*
by *standard* (*metis assms galois.existence-upper-preserves-Inf*)

locale *powerset* =
  galois.complete-lattice-class lower upper
for *lower* :: $'a\ set \Rightarrow 'b\ set$
and *upper* :: $'b\ set \Rightarrow 'a\ set$
begin

lemma *lower-insert*:
  shows *lower* (*insert x X*) = *lower* $\{x\} \cup$ *lower X*
by (*metis insert-is-Un lower-sup*)

lemma *lower-distributive*:
  shows *lower X* = $(\bigcup x{\in}X.\ lower\ \{x\})$
using *lower-Sup*[**where** $X{=}\{\{x\}\ |x.\ x \in X\}$] **by** (*auto simp*: *Union-singleton*)

sublocale *closure-powerset cl* **..**

end

locale *powerset-distributive* =
  galois.powerset
+ *galois.complete-lattice-distributive-class*
begin

lemma *upper-insert*:
  shows *upper* (*insert x X*) = *upper* $\{x\} \cup$ *upper X*
by (*metis insert-is-Un upper-sup*)

lemma *cl-distributive-axiom*:
  shows *cl* $(\bigcup\ X) \subseteq \bigcup\ (cl\ `\ X)$
by (*simp add*: *cl-def lower-Sup upper-Sup*)

sublocale *closure-powerset-distributive cl*
by *standard* (*simp add*: *cl-distributive-axiom cla.le-supI1*)

end

Müller-Olm (1997, Theorems 3.3.1, 3.3.2): relation image forms a Galois connection. See also Davey and Priestley (2002, Exercise 7.18).

definition $lower_R :: ('a \times 'b)\ set \Rightarrow 'a\ set \Rightarrow 'b\ set$ **where**
  $lower_R\ R\ A = R\ ``\ A$

definition $upper_R :: ('a \times 'b)\ set \Rightarrow 'b\ set \Rightarrow 'a\ set$ **where**
  $upper_R\ R\ B = \{a.\ \forall b.\ (a, b) \in R \longrightarrow b \in B\}$

interpretation *relation*: *galois.powerset galois.lower*$_R$ *R galois.upper*$_R$ *R*
unfolding *galois.lower*$_R$*-def galois.upper*$_R$*-def* **by** *standard blast*

context *galois.powerset*
begin

lemma *relations-galois*:
defines $R \equiv \{(a, b).\ b \in lower\ \{a\}\}$
shows $lower = galois.lower_R\ R$
and $upper = galois.upper_R\ R$
proof −
show $lower = galois.lower_R\ R$
proof(*rule HOL.ext*)
fix X
have $lower\ X = (\bigcup x{\in}X.\ lower\ \{x\})$ **by** (*rule lower-distributive*)
also have $\ldots = (\bigcup x{\in}X.\ galois.lower_R\ R\ \{x\})$ **by** (*simp add*: *galois.lower*$_R$*-def R-def*)
also have $\ldots = galois.lower_R\ R\ X$ **unfolding** *galois.lower*$_R$*-def R-def* **by** *blast*
finally show $lower\ X = galois.lower_R\ R\ X$ **.**
qed
then show $upper = galois.upper_R\ R$
using *galois galois.relation.lower-upper-unique* **by** *blast*
qed

end

setup ‹*Sign.parent-path*›

6.1 Some Galois connections

setup ‹*Sign.mandatory-path galois*›

locale *complete-lattice-class-monomorphic*
= *galois.complete-lattice-class upper lower*
for *upper* :: *'a*::*complete-lattice* ⇒ *'a* **and** *lower* :: *'a* ⇒ *'a* — Avoid *'a itself* parameters

interpretation *conj-imp*: *galois.complete-lattice-class* (⊓) *x* ($\longrightarrow_B$) *x* **for** *x* :: -::*boolean-algebra* — Classic example
by *standard* (*simp add*: *boolean-implication.conv-sup inf-commute shunt1*)

There are very well-behaved Galois connections arising from the image (and inverse image) of sets under a function; stuttering is one instance (§8.1).

locale *image-vimage* =
fixes *f* :: *'a* ⇒ *'b*
begin

definition *lower* :: *'a set* ⇒ *'b set* **where**
lower X = *f* ‘ *X*

definition *upper* :: *'b set* ⇒ *'a set* **where**
upper X = *f* −‘ *X*

lemma *upper-empty*[*iff*]:
shows *upper* {} = {}
unfolding *upper-def* **by** *simp*

sublocale *galois.powerset-distributive lower upper*
unfolding *lower-def upper-def* **by** *standard* (*simp-all add*: *image-subset-iff-subset-vimage vimage-Union*)

abbreviation *equivalent* :: *'a relp* **where**
equivalent x y ≡ *f x* = *f y*

lemma *equiv*:
shows *Equiv-Relations.equivp equivalent*
by (*simp add*: *equivpI reflpI symp-def transp-def*)

lemma *equiv-cl-singleton*:
  assumes *equivalent x y*
  shows *cl* $\{x\}$ = *cl* $\{y\}$
using *assms* **by** (*simp add*: *cl-def galois.image-vimage.lower-def*)

lemma *cl-alt-def*:
  shows *cl X* = $\{(x, y).$ *equivalent x y*$\}$ `` *X*
by (*simp add*: *cl-def lower-def upper-def vimage-image-eq*)

sublocale *closure-powerset-distributive-exchange cl*
by *standard* (*auto simp*: *cl-alt-def intro*: *exchangeI*)

lemma *closed-in*:
  assumes $x \in P$
  assumes *equivalent x y*
  assumes *P*: $P \in$ *closed*
  shows $y \in P$
using *assms*(*1−2*) *closed-conv*[*OF P*] **unfolding** *cl-alt-def* **by** *blast*

lemma *clE*:
  assumes $x \in$ *cl P*
  obtains *y* **where** *equivalent y x* **and** $y \in P$
using *assms* **unfolding** *cl-alt-def* **by** *blast*

lemma *clI*[*intro*]:
  assumes $x \in P$
  assumes *equivalent x y*
  shows $y \in$ *cl P*
unfolding *cl-alt-def* **using** *assms* **by** *blast*

lemma *closed-diff*[*intro*]:
  assumes $X \in$ *closed*
  assumes $Y \in$ *closed*
  shows $X - Y \in$ *closed*
by (*rule closedI*) (*metis Diff-iff assms clE closed-in*)

lemma *closed-uminus*[*intro*]:
  assumes $X \in$ *closed*
  shows $-X \in$ *closed*
using *closed-diff*[**where** *X*=*UNIV*, *OF* - *assms*] **by** *fastforce*

end

locale *image-vimage-monomorphic*
  = *galois.image-vimage f*
      for *f* :: $'a \Rightarrow 'a$ — Avoid $'a$ *itself* parameters

locale *image-vimage-idempotent*
  = *galois.image-vimage-monomorphic* +
    assumes *f-idempotent*: $\bigwedge x.$ *f* (*f x*) = *f x*
begin

lemma *f-idempotent-comp*:
  shows $f \circ f = f$
by (*simp add*: *comp-def f-idempotent*)

lemma *idemI*:

assumes $f\ x \in P$
shows $x \in cl\ P$
using *assms f-idempotent* **by** (*auto simp*: *cl-alt-def*)

lemma *f-cl*:
shows $f\ x \in cl\ P \longleftrightarrow x \in cl\ P$
by (*simp add*: *cl-alt-def f-idempotent*)

lemma *f-closed*:
assumes $P \in closed$
shows $f\ x \in P \longleftrightarrow x \in P$
by (*metis assms closed-conv f-cl*)

lemmas *f-closedI* = *iffD1*[*OF f-closed*]

end

setup ‹*Sign.parent-path*›

7 Heyting algebras

Our (complete) lattices are Heyting algebras. The following development is oriented towards using the derived Heyting implication in a logical fashion. As there are no standard classes for semi-(complete-)lattices we simply work with complete lattices.

References:

- Esakia, Bezhanishvili, Holliday, and Evseev (2019) – fundamental theory
- van Dalen (2004, Lemma 5.2.1) – some equivalences
- https://en.wikipedia.org/wiki/Pseudocomplement – properties

class *heyting-algebra* = *complete-lattice* +
assumes *inf-Sup-distrib1*: $\bigwedge Y{::}'a\ set.\ \bigwedge x{::}'a.\ x \sqcap (\bigsqcup Y) = (\bigsqcup y{\in}Y.\ x \sqcap y)$
begin

definition *heyting* :: $'a \Rightarrow 'a \Rightarrow 'a$ (**infixr** ‹$\longrightarrow_H$› *53*) **where**
$x \longrightarrow_H y = \bigsqcup \{z.\ x \sqcap z \leq y\}$

lemma *heyting*: — The Galois property for ($\sqcap$) and $\longrightarrow_H$
shows $z \leq x \longrightarrow_H y \longleftrightarrow z \sqcap x \leq y$ (**is** *?lhs* $\longleftrightarrow$ *?rhs*)
proof(*rule iffI*)
from *inf-Sup-distrib1* **have** $\bigsqcup \{a.\ x \sqcap a \leq y\} \sqcap x \leq y$ **by** (*simp add*: *SUP-le-iff inf-commute*)
then show *?lhs* $\Longrightarrow$ *?rhs* **unfolding** *heyting-def* **by** (*meson inf-mono order.trans order-refl*)
show *?rhs* $\Longrightarrow$ *?lhs* **by** (*simp add*: *heyting-def Sup-upper inf.commute*)
qed

end

setup ‹*Sign.mandatory-path heyting*›

context *heyting-algebra*
begin

lemma *commute*:
shows $x \sqcap z \leq y \longleftrightarrow z \leq (x \longrightarrow_H y)$
by (*simp add*: *heyting inf.commute*)

lemmas *uncurry* = *iffD1*[*OF heyting*]
lemmas *curry* = *iffD2*[*OF heyting*]

lemma *curry-conv*:
  shows $(x \sqcap y \longrightarrow_H z) = (x \longrightarrow_H y \longrightarrow_H z)$
by (*simp add*: *order-eq-iff*) (*metis heyting eq-refl inf.assoc*)

lemma *swap*:
  shows $P \longrightarrow_H Q \longrightarrow_H R = Q \longrightarrow_H P \longrightarrow_H R$
by (*metis curry-conv inf.commute*)

lemma *absorb*:
  shows $y \sqcap (x \longrightarrow_H y) = y$
    and $(x \longrightarrow_H y) \sqcap y = y$
by (*simp-all add*: *curry inf-absorb1 ac-simps*)

lemma *detachment*:
  shows $x \sqcap (x \longrightarrow_H y) = x \sqcap y$ (**is** *?thesis1*)
    and $(x \longrightarrow_H y) \sqcap x = x \sqcap y$ (**is** *?thesis2*)
proof −
  show *?thesis1* **by** (*metis absorb*(*1*) *uncurry inf.assoc inf.commute inf.idem inf-iff-le*(*2*))
  then show *?thesis2* **by** (*simp add*: *ac-simps*)
qed

lemma *discharge*:
  assumes $x' \leq x$
  shows $x' \sqcap (x \longrightarrow_H y) = x' \sqcap y$ (**is** *?thesis1*)
    and $(x \longrightarrow_H y) \sqcap x' = y \sqcap x'$ (**is** *?thesis2*)
proof −
  from *assms* **show** *?thesis1* **by** (*metis curry-conv detachment*(*2*) *inf.absorb1*)
  then show *?thesis2* **by** (*simp add*: *ac-simps*)
qed

lemma *trans*:
  shows $(x \longrightarrow_H y) \sqcap (y \longrightarrow_H z) \leq x \longrightarrow_H z$
by (*metis curry detachment*(*2*) *swap uncurry inf-le2*)

lemma *rev-trans*:
  shows $(y \longrightarrow_H z) \sqcap (x \longrightarrow_H y) \leq x \longrightarrow_H z$
by (*simp add*: *inf.commute trans*)

lemma *discard*:
  shows $Q \leq P \longrightarrow_H Q$
by (*simp add*: *curry*)

lemma *infR*:
  shows $x \longrightarrow_H y \sqcap z = (x \longrightarrow_H y) \sqcap (x \longrightarrow_H z)$
by (*simp add*: *order-eq-iff curry uncurry detachment le-infI2*)

lemma *mono*:
  assumes $x' \leq x$
  assumes $y \leq y'$
  shows $x \longrightarrow_H y \leq x' \longrightarrow_H y'$
using *assms* **by** (*metis curry detachment*(*1*) *uncurry inf-commute inf-absorb2 le-infI1*)

lemma *strengthen*[*strg*]:
  assumes *st-ord* $(\neg F)$ X X'
  assumes *st-ord* F Y Y'

shows *st-ord F* $(X \longrightarrow_H Y)$ $(X' \longrightarrow_H Y')$
using *assms* **by** (*cases F*; *simp add*: *heyting.mono*)

lemma *mono2mono*[*cont-intro*, *partial-function-mono*]:
assumes *monotone orda* $(\geq)$ *F*
assumes *monotone orda* $(\leq)$ *G*
shows *monotone orda* $(\leq)$ $(\lambda x.\ F\ x \longrightarrow_H G\ x)$
by (*simp add*: *monotoneI curry discharge le-infI1 monotoneD*[*OF assms*(*1*)] *monotoneD*[*OF assms*(*2*)])

lemma *mp*:
assumes $x \leq y \longrightarrow_H z$
assumes $x \leq y$
shows $x \leq z$
by (*meson assms uncurry inf-greatest order.refl order-trans*)

lemma *botL*:
shows $\bot \longrightarrow_H x = \top$
by (*simp add*: *heyting top-le*)

lemma *top-conv*:
shows $x \longrightarrow_H y = \top \longleftrightarrow x \leq y$
by (*metis curry detachment*(*2*) *inf-iff-le*(*1*) *inf-top.left-neutral*)

lemma *refl*[*simp*]:
shows $x \longrightarrow_H x = \top$
by (*simp add*: *top-conv*)

lemma *topL*[*simp*]:
shows $\top \longrightarrow_H x = x$
by (*metis detachment*(*1*) *inf-top-left*)

lemma *topR*[*simp*]:
shows $x \longrightarrow_H \top = \top$
by (*simp add*: *top-conv*)

lemma *K*[*simp*]:
shows $x \longrightarrow_H (y \longrightarrow_H x) = \top$
by (*simp add*: *discard top-conv*)

subclass *distrib-lattice*
proof — Esakia et al. (2019, Proposition 1.5.3)
have $x \sqcap (y \sqcup z) \leq x \sqcap y \sqcup x \sqcap z$ **for** *x y z* :: *'a*
using *commute* **by** *fastforce*
then have $x \sqcap (y \sqcup z) = (x \sqcap y) \sqcup (x \sqcap z)$ **for** *x y z* :: *'a*
by (*simp add*: *order.eq-iff le-infI2*)
then show $x \sqcup (y \sqcap z) = (x \sqcup y) \sqcap (x \sqcup z)$ **for** *x y z* :: *'a*
by (*rule distrib-imp1*)
qed

lemma *supL*:
shows $(x \sqcup y) \longrightarrow_H z = (x \longrightarrow_H z) \sqcap (y \longrightarrow_H z)$
by (*simp add*: *order-eq-iff mono curry uncurry inf-sup-distrib1*)

subclass (**in** *complete-distrib-lattice*) *heyting-algebra* **by** *standard* (*rule inf-Sup*)

lemma *inf-Sup-distrib*:
shows $x \sqcap \bigsqcup Y = (\bigsqcup y \in Y.\ x \sqcap y)$
and $\bigsqcup Y \sqcap x = (\bigsqcup y \in Y.\ x \sqcap y)$

by (*simp-all add*: *inf-Sup-distrib1 inf-commute*)

lemma *inf-SUP-distrib*:
  shows $x \sqcap (\bigsqcup i{\in}I.\ Y\ i) = (\bigsqcup i{\in}I.\ x \sqcap\ Y\ i)$
    and $(\bigsqcup i{\in}I.\ Y\ i) \sqcap x = (\bigsqcup i{\in}I.\ Y\ i \sqcap x)$
by (*simp-all add*: *inf-Sup-distrib image-image ac-simps*)

end

lemma *eq-boolean-implication*: — the implications coincide in *boolean-algebra*s
  fixes x :: -::*boolean-algebra*
  shows $x \longrightarrow_H y = x \longrightarrow_B y$
by (*simp add*: *order.eq-iff boolean-implication-def heyting.detachment heyting.curry flip*: *shunt1*)

lemmas *simp-thms* =
  heyting.botL
  heyting.topL
  heyting.topR
  heyting.refl

lemma *Sup-prime-Sup-irreducible-iff*:
  fixes x :: -::*heyting-algebra*
  shows *Sup-prime* $x \longleftrightarrow$ *Sup-irreducible* x
by (*fastforce simp*: *Sup-prime-on-def Sup-irreducible-on-def inf.order-iff heyting.inf-Sup-distrib*
        intro: *Sup-prime-on-imp-Sup-irreducible-on*)

Logical rules ala HOL **lemma** *bspec*:
  fixes P :: - $\Rightarrow$ (-::*heyting-algebra*)
  shows $x \in X \Longrightarrow (\bigsqcap x{\in}X.\ P\ x \longrightarrow_H\ Q\ x) \sqcap P\ x \leq Q\ x$ (**is** *?X* $\Longrightarrow$ *?thesis1*)
    and $x \in X \Longrightarrow P\ x \sqcap (\bigsqcap x{\in}X.\ P\ x \longrightarrow_H\ Q\ x) \leq Q\ x$ (**is** - $\Longrightarrow$ *?thesis2*)
    and $(\bigsqcap x.\ P\ x \longrightarrow_H\ Q\ x) \sqcap P\ x \leq Q\ x$ (**is** *?thesis3*)
    and $P\ x \sqcap (\bigsqcap x.\ P\ x \longrightarrow_H\ Q\ x) \leq Q\ x$ (**is** *?thesis4*)
proof −
  show *?X* $\Longrightarrow$ *?thesis1* **by** (*meson INF-lower heyting.uncurry*)
  then show *?X* $\Longrightarrow$ *?thesis2* **by** (*simp add*: *inf-commute*)
  show *?thesis3* **by** (*simp add*: *Inf-lower heyting.commute inf-commute*)
  then show *?thesis4* **by** (*simp add*: *inf-commute*)
qed

lemma *INFL*:
  fixes Q :: -::*heyting-algebra*
  shows $(\bigsqcap x{\in}X.\ P\ x \longrightarrow_H\ Q) = (\bigsqcup x{\in}X.\ P\ x) \longrightarrow_H\ Q$ (**is** *?lhs* = *?rhs*)
proof(*rule antisym*)
  show *?lhs* $\leq$ *?rhs* **by** (*meson INFE SUPE order.refl heyting.commute heyting.uncurry*)
  show *?rhs* $\leq$ *?lhs* **by** (*simp add*: *INFI SupI heyting.mono*)
qed

lemmas *SUPL* = *heyting.INFL*[*symmetric*]

lemma *INFR*:
  fixes P :: -::*heyting-algebra*
  shows $(\bigsqcap x{\in}X.\ P \longrightarrow_H\ Q\ x) = (P \longrightarrow_H (\bigsqcap x{\in}X.\ Q\ x))$ (**is** *?lhs* = *?rhs*)
by (*simp add*: *order-eq-iff INFI INF-lower heyting.mono*)
  (*meson INFI INF-lower heyting.curry heyting.uncurry*)

lemmas *Inf-simps* = — "Miniscoping: pushing in universal quantifiers."
  Inf-inf
  inf-Inf

INF-inf-const1
INF-inf-const2
heyting.INFL
heyting.INFR

lemma *SUPL-le*:
fixes Q :: -::*heyting-algebra*
shows $(\bigsqcup x{\in}X.\ P\ x \longrightarrow_H Q) \leq (\bigsqcap x{\in}X.\ P\ x) \longrightarrow_H Q$
by (*simp add*: *INF-lower SUPE heyting.mono*)

lemma *SUPR-le*:
fixes P :: -::*heyting-algebra*
shows $(\bigsqcup x{\in}X.\ P \longrightarrow_H Q\ x) \leq P \longrightarrow_H (\bigsqcup x{\in}X.\ Q\ x)$
by (*simp add*: *SUPE SUP-upper heyting.mono*)

lemma *SUP-inf*:
fixes Q :: -::*heyting-algebra*
shows $(\bigsqcup x{\in}X.\ P\ x \sqcap Q) = (\bigsqcup x{\in}X.\ P\ x) \sqcap Q$
by (*simp add*: *heyting.inf-SUP-distrib*(*2*))

lemma *inf-SUP*:
fixes P :: -::*heyting-algebra*
shows $(\bigsqcup x{\in}X.\ P \sqcap Q\ x) = P \sqcap (\bigsqcup x{\in}X.\ Q\ x)$
by (*simp add*: *heyting.inf-SUP-distrib*(*1*))

lemmas *Sup-simps* = — "Miniscoping: pushing in universal quantifiers."
sup-SUP
SUP-sup
heyting.inf-SUP
heyting.SUP-inf

lemma *mcont2mcont-inf*[*cont-intro*]:
fixes F :: - ⇒ $'a$::*heyting-algebra*
fixes G :: - ⇒ $'a$::*heyting-algebra*
assumes *mcont luba orda Sup* $(\leq)$ F
assumes *mcont luba orda Sup* $(\leq)$ G
shows *mcont luba orda Sup* $(\leq)$ $(\lambda x.\ F\ x \sqcap G\ x)$
proof −
have *mcont-inf1*: *mcont Sup* $(\leq)$ *Sup* $(\leq)$ $(\lambda y.\ x \sqcap y)$ **for** x :: $'a$::*heyting-algebra*
by (*auto intro*!: *contI mcontI monotoneI intro*: *le-infI2 simp flip*: *heyting.inf-SUP-distrib*)
then have *mcont-inf2*: *mcont Sup* $(\leq)$ *Sup* $(\leq)$ $(\lambda x.\ x \sqcap y)$ **for** y :: $'a$::*heyting-algebra*
by (*subst inf.commute*) (*rule mcont-inf1*)
from *assms mcont-inf1 mcont-inf2* **show** *?thesis*
by (*best intro*: *ccpo.mcont2mcont'*[*OF complete-lattice-ccpo*] *ccpo.mcont-const*[*OF complete-lattice-ccpo*])
qed

lemma *closure-imp-distrib-le*: — Abadi and Plotkin (1993, Lemma 3.3), generalized
fixes P Q :: - :: *heyting-algebra*
assumes *cl*: *closure-axioms* $(\leq)$ *cl*
assumes *cl-inf*: $\bigwedge x\ y.\ cl\ x \sqcap cl\ y \leq cl\ (x \sqcap y)$
shows $P \longrightarrow_H Q \leq cl\ P \longrightarrow_H cl\ Q$
proof −
from *cl* **have** $(P \longrightarrow_H Q) \sqcap cl\ P \leq cl\ (P \longrightarrow_H Q) \sqcap cl\ P$
by (*metis* (*mono-tags*) *closure-axioms-def inf-mono order.refl*)
also have $\ldots \leq cl\ ((P \longrightarrow_H Q) \sqcap P)$
by (*simp add*: *cl-inf*)
also from *cl* **have** $\ldots \leq cl\ Q$
by (*metis* (*mono-tags*) *closure-axioms-def order.refl heyting.mono heyting.uncurry*)

finally show *?thesis*
by (*simp add*: *heyting*)
qed

setup ‹*Sign.parent-path*›

Pseudocomplements **definition** *pseudocomplement* :: *'a*::*heyting-algebra* ⇒ *'a* (‹¬$_H$ -› [*75*] *75*) **where**
$\neg_H x = x \longrightarrow_H \bot$

lemma *pseudocomplementI*:
shows $x \leq \neg_H y \longleftrightarrow x \sqcap y \leq \bot$
by (*simp add*: *pseudocomplement-def heyting*)

setup ‹*Sign.mandatory-path pseudocomplement*›

lemma *monotone*:
shows *antimono pseudocomplement*
by (*simp add*: *antimonoI heyting.mono pseudocomplement-def*)

lemmas *strengthen*[*strg*] = *st-monotone*[*OF pseudocomplement.monotone*]
lemmas *mono* = *monotoneD*[*OF pseudocomplement.monotone*]
lemmas *mono2mono*[*cont-intro*, *partial-function-mono*]
= *monotone2monotone*[*OF pseudocomplement.monotone*, *simplified*, *of orda P* **for** *orda P*]

lemma *eq-boolean-negation*: — the negations coincide in *boolean-algebra*s
fixes *x* :: -::{*boolean-algebra*,*heyting-algebra*}
shows $\neg_H x = -x$
by (*simp add*: *pseudocomplement-def heyting.eq-boolean-implication*)

lemma *heyting*:
shows $x \longrightarrow_H \neg_H x = \neg_H x$
by (*simp add*: *pseudocomplement-def order-eq-iff heyting heyting.detachment*)

lemma *Inf*:
shows $x \sqcap \neg_H x = \bot$
and $\neg_H x \sqcap x = \bot$
by (*simp-all add*: *pseudocomplement-def heyting.detachment*)

lemma *double-le*:
shows $x \leq \neg_H \neg_H x$
by (*simp add*: *pseudocomplement-def heyting.detachment heyting.curry*)

interpretation *double*: *closure-complete-lattice-class pseudocomplement* ∘ *pseudocomplement*
by *standard* (*simp*; *meson order.trans pseudocomplement.double-le pseudocomplement.mono*)

lemma *triple*:
shows $\neg_H \neg_H \neg_H x = \neg_H x$
by (*simp add*: *order-eq-iff pseudocomplement.double-le pseudocomplement.mono*)

lemma *contrapos-le*:
shows $x \longrightarrow_H y \leq \neg_H y \longrightarrow_H \neg_H x$
by (*simp add*: *heyting.curry heyting.trans pseudocomplement-def*)

lemma *sup-inf*: — half of de Morgan
shows $\neg_H(x \sqcup y) = \neg_H x \sqcap \neg_H y$
by (*simp add*: *pseudocomplement-def heyting.supL*)

lemma *inf-sup-weak*: — the weakened other half of de Morgan

shows $\neg_H(x \sqcap y) = \neg_H\neg_H(\neg_H x \sqcup \neg_H y)$
by (*metis* (*no-types*, *opaque-lifting*) *pseudocomplement-def heyting.curry-conv heyting.supL inf-commute pseudocomplement.triple*)

lemma *fix-triv*:
assumes $x = \neg_H x$
shows $x = y$
using *assms* **by** (*metis antisym bot.extremum inf.idem inf-le2 pseudocomplementI*)

lemma *double-top*:
shows $\neg_H\neg_H(x \sqcup \neg_H x) = \top$
by (*metis pseudocomplement-def heyting.refl pseudocomplement.Inf*(*1*) *pseudocomplement.sup-inf*)

lemma *Inf-inf*:
fixes P :: - $\Rightarrow$ (-::*heyting-algebra*)
shows $(\bigsqcap x.\ P\ x) \sqcap \neg_H P\ x = \bot$
by (*simp add*: *pseudocomplement-def Inf-lower heyting.discharge*(*1*))

lemma *SUP-le*: — half of de Morgan
fixes P :: - $\Rightarrow$ (-::*heyting-algebra*)
shows $(\bigsqcup x{\in}X.\ P\ x) \leq \neg_H(\bigsqcap x{\in}X.\ \neg_H P\ x)$
by (*rule SUP-least*) (*meson INF-lower order.trans pseudocomplement.double-le pseudocomplement.mono*)

lemma *SUP-INF-le*:
fixes P :: - $\Rightarrow$ (-::*heyting-algebra*)
shows $(\bigsqcup x{\in}X.\ \neg_H P\ x) \leq \neg_H(\bigsqcap x{\in}X.\ P\ x)$
by (*simp add*: *INF-lower SUPE pseudocomplement.mono*)

lemma *SUP*:
fixes P :: - $\Rightarrow$ (-::*heyting-algebra*)
shows $\neg_H(\bigsqcup x{\in}X.\ P\ x) = (\bigsqcap x{\in}X.\ \neg_H P\ x)$
by (*simp add*: *order.eq-iff SUP-upper le-INF-iff pseudocomplement.mono*)
(*metis inf-commute pseudocomplement.SUP-le pseudocomplementI*)

setup ‹*Sign.parent-path*›

7.1 Downwards closure of preorders (downsets)

A *downset* (also *lower set* and *order ideal*) is a subset of a preorder that is closed under the order relation. (An *ideal* is a downset that is *directed*.) Some results require antisymmetry (a partial order).

References:

- Vickers (1989), early chapters.
- https://en.wikipedia.org/wiki/Alexandrov_topology
- Abadi and Plotkin (1991, §3)

setup ‹*Sign.mandatory-path downwards*›

definition *cl* :: *'a*::*preorder set* $\Rightarrow$ *'a set* **where**
cl P = {*x* |*x y*. *y* $\in$ *P* $\wedge$ *x* $\leq$ *y*}

setup ‹*Sign.parent-path*›

interpretation *downwards*: *closure-powerset-distributive downwards.cl* — On preorders
proof *standard*
show (*P* $\subseteq$ *downwards.cl Q*) $\longleftrightarrow$ (*downwards.cl P* $\subseteq$ *downwards.cl Q*) **for** *P Q* :: *'a set*

unfolding *downwards.cl-def* **by** (*auto dest*: *order-trans*)
show *downwards.cl* $(\bigcup X) \subseteq \bigcup$ (*downwards.cl* ‘ X) $\cup$ *downwards.cl* {} **for** X :: *'a set set*
unfolding *downwards.cl-def* **by** *auto*
qed

interpretation *downwards*: *closure-powerset-distributive-anti-exchange* (*downwards.cl*::-::*order set* ⇒ -)
— On partial orders; see Pfaltz and Šlapal (2013)
by *standard* (*unfold downwards.cl-def*; *blast intro*: *anti-exchangeI antisym*)

setup ‹*Sign.mandatory-path downwards*›

lemma *cl-empty*:
shows *downwards.cl* {} = {}
unfolding *downwards.cl-def* **by** *simp*

lemma *closed-empty*[*iff*]:
shows {} ∈ *downwards.closed*
using *downwards.cl-def* **by** *fastforce*

lemma *clI*[*intro*]:
assumes $y \in P$
assumes $x \leq y$
shows $x \in$ *downwards.cl* P
unfolding *closure.closed-def downwards.cl-def* **using** *assms* **by** *blast*

lemma *clE*:
assumes $x \in$ *downwards.cl* P
obtains y **where** $y \in P$ **and** $x \leq y$
using *assms* **unfolding** *downwards.cl-def* **by** *fast*

lemma *closed-in*:
assumes $x \in P$
assumes $y \leq x$
assumes $P \in$ *downwards.closed*
shows $y \in P$
using *assms* **unfolding** *downwards.cl-def downwards.closed-def* **by** *blast*

lemma *order-embedding*: — On preorders; see Davey and Priestley (2002, §1.35)
fixes x :: -::*preorder*
shows *downwards.cl* $\{x\} \subseteq$ *downwards.cl* $\{y\} \longleftrightarrow x \leq y$
using *downwards.cl* **by** (*blast elim*: *downwards.clE*)

The lattice of downsets of a set X is always a *heyting-algebra*.
References:

- Ono (2019, §7.5); uses upsets, points to Stone (1938) as the origin
- Esakia et al. (2019, §2.2)
- https://en.wikipedia.org/wiki/Intuitionistic_logic#Heyting_algebra_semantics

definition *imp* :: *'a*::*preorder set* ⇒ *'a set* ⇒ *'a set* **where**
imp P Q = {σ. $\forall \sigma' \leq \sigma$. $\sigma' \in P \longrightarrow \sigma' \in Q$}

lemma *imp-refl*:
shows *downwards.imp* P P = *UNIV*
by (*simp add*: *downwards.imp-def*)

lemma *imp-contained*:

assumes $P \subseteq Q$
shows *downwards.imp P Q* = *UNIV*
unfolding *downwards.imp-def* **using** *assms* **by** *fast*

lemma *heyting-imp*:
assumes $P \in$ *downwards.closed*
shows $P \subseteq$ *downwards.imp Q R* $\longleftrightarrow$ $P \cap Q \subseteq R$
using *assms* **unfolding** *downwards.imp-def downwards.closed-def* **by** *blast*

lemma *imp-mp'*:
assumes $\sigma \in$ *downwards.imp P Q*
assumes $\sigma \in P$
shows $\sigma \in Q$
using *assms* **by** (*simp add*: *downwards.imp-def*)

lemma *imp-mp*:
shows $P \cap$ *downwards.imp P Q* $\subseteq Q$
and *downwards.imp P Q* $\cap P \subseteq Q$
by (*meson IntD1 IntD2 downwards.imp-mp' subsetI*)+

lemma *imp-contains*:
assumes $X \subseteq Q$
assumes $X \in$ *downwards.closed*
shows $X \subseteq$ *downwards.imp P Q*
using *assms* **by** (*auto simp*: *downwards.imp-def elim*: *downwards.closed-in*)

lemma *imp-downwards*:
assumes $y \in$ *downwards.imp P Q*
assumes $x \leq y$
shows $x \in$ *downwards.imp P Q*
using *assms order-trans* **by** (*force simp*: *downwards.imp-def*)

lemma *closed-imp*:
shows *downwards.imp P Q* $\in$ *downwards.closed*
by (*meson downwards.clE downwards.closedI downwards.imp-downwards*)

The set *downwards.imp P Q* is the greatest downset contained in the Boolean implication $P \longrightarrow_B Q$, i.e., *downwards.imp* is the *kernel* of $(\longrightarrow_B)$ (Zwiers 1989). Note that "kernel" is a choice or interior function.

lemma *imp-boolean-implication-subseteq*:
shows *downwards.imp P Q* $\subseteq P \longrightarrow_B Q$
unfolding *downwards.imp-def boolean-implication.set-alt-def* **by** *blast*

lemma *downwards-closed-imp-greatest*:
assumes $R \subseteq P \longrightarrow_B Q$
assumes $R \in$ *downwards.closed*
shows $R \subseteq$ *downwards.imp P Q*
using *assms* **unfolding** *boolean-implication.set-alt-def downwards.imp-def downwards.closed-def* **by** *blast*

definition *kernel* :: *'a*::*order set* $\Rightarrow$ *'a set* **where**
kernel X = $\bigsqcup \{Q \in$ *downwards.closed*. $Q \subseteq X\}$

lemma *kernel-def2*:
shows *downwards.kernel X* = $\{\sigma.\ \forall \sigma' \leq \sigma.\ \sigma' \in X\}$ (**is** *?lhs* = *?rhs*)
proof(*rule antisym*)
show *?lhs* $\subseteq$ *?rhs*
unfolding *downwards.kernel-def* **using** *downwards.closed-conv* **by** *blast*
next
have $x \in$ *?lhs* **if** $x \in$ *?rhs* **for** x

unfolding *downwards.kernel-def* **using** *that*
by (*auto elim*: *downwards.clE intro*: *exI*[**where** *x*=*downwards.cl* {*x*}])
then show *?rhs* ⊆ *?lhs* **by** *blast*
qed

lemma *kernel-contractive*:
shows *downwards.kernel X* ⊆ *X*
unfolding *downwards.kernel-def* **by** *blast*

lemma *kernel-idempotent*:
shows *downwards.kernel* (*downwards.kernel X*) = *downwards.kernel X*
unfolding *downwards.kernel-def* **by** *blast*

lemma *kernel-monotone*:
shows *mono downwards.kernel*
unfolding *downwards.kernel-def* **by** (*rule monotoneI*) *blast*

lemma *closed-kernel-conv*:
shows *X* ∈ *downwards.closed* ⟷ *downwards.kernel X* = *X*
unfolding *downwards.kernel-def2 downwards.closed-def* **by** (*blast elim*: *downwards.clE*)

lemma *closed-kernel*:
shows *downwards.kernel X* ∈ *downwards.closed*
by (*simp add*: *downwards.closed-kernel-conv downwards.kernel-idempotent*)

lemma *kernel-cl*:
shows *downwards.kernel* (*downwards.cl X*) = *downwards.cl X*
using *downwards.closed-kernel-conv* **by** *blast*

lemma *cl-kernel*:
shows *downwards.cl* (*downwards.kernel X*) = *downwards.kernel X*
by (*simp flip*: *downwards.closed-conv add*: *downwards.closed-kernel*)

lemma *kernel-boolean-implication*:
fixes *P* :: -::*order*
shows *downwards.kernel* (*P* ⟶$_B$ *Q*) = *downwards.imp P Q*
unfolding *downwards.kernel-def2 boolean-implication.set-alt-def downwards.imp-def* **by** *blast*

setup ‹*Sign.parent-path*›

8 Safety logic

Following Abadi and Lamport (1995); Abadi and Plotkin (1991, 1993) (see also Abadi and Merz (1996, §5.5)), we work in the complete lattice of stuttering-closed safety properties (i.e., stuttering-closed downsets) and use this for logical purposes. We avoid many syntactic issues via a shallow embedding into HOL.

8.1 Stuttering

We define *stuttering equivalence* ala Lamport (1994). This allows any agent to repeat any state at any time. We define a normalisation function (♮) on ('*a*, '*s*, '*v*) *trace.t* and extract the (matroidal) closure over sets of these from the Galois connection *galois.image-vimage*.

setup ‹*Sign.mandatory-path trace*›

primrec *natural'* :: '*s* ⇒ ('*a* × '*s*) *list* ⇒ ('*a* × '*s*) *list* **where**
natural' s [] = []
| *natural' s* (*x* # *xs*) = (*if snd x* = *s then natural' s xs else x* # *natural'* (*snd x*) *xs*)

setup ‹*Sign.mandatory-path final'*›

lemma *natural'*[*simp*]:
  shows *trace.final' s* (*trace.natural' s xs*) = *trace.final' s xs*
by (*induct xs arbitrary*: *s*) *simp-all*

lemma *natural'-cong*:
  assumes *s* = *s'*
  assumes *trace.natural' s xs* = *trace.natural' s xs'*
  shows *trace.final' s xs* = *trace.final' s' xs'*
by (*metis assms trace.final'.natural'*)

setup ‹*Sign.parent-path*›

setup ‹*Sign.mandatory-path natural'*›

lemma *natural'*:
  shows *trace.natural' s* (*trace.natural' s xs*) = *trace.natural' s xs*
by (*induct xs arbitrary*: *s*) *simp-all*

lemma *length*:
  shows *length* (*trace.natural' s xs*) ≤ *length xs*
by (*induct xs arbitrary*: *s*) (*simp-all add*: *le-SucI*)

lemma *subseq*:
  shows *subseq* (*trace.natural' s xs*) *xs*
by (*induct xs arbitrary*: *s*) *auto*

lemma *remdups-adj*:
  shows *s* # *map snd* (*trace.natural' s xs*) = *remdups-adj* (*s* # *map snd xs*)
by (*induct xs arbitrary*: *s*) *simp-all*

lemma *append*:
  shows *trace.natural' s* (*xs* @ *ys*) = *trace.natural' s xs* @ *trace.natural'* (*trace.final' s xs*) *ys*
by (*induct xs arbitrary*: *s*) *simp-all*

lemma *eq-Nil-conv*:
  shows *trace.natural' s xs* = [] ⟷ *snd* ‘ *set xs* ⊆ {*s*}
    and [] = *trace.natural' s xs* ⟷ *snd* ‘ *set xs* ⊆ {*s*}
by (*induct xs arbitrary*: *s*) *simp-all*

lemma *eq-Cons-conv*:
  shows *trace.natural' s xs* = *y* # *ys*
    ⟷ (∃ *xs' ys'*. *xs* = *xs'* @ *y* # *ys'* ∧ *snd* ‘ *set xs'* ⊆ {*s*} ∧ *snd y* ≠ *s* ∧ *trace.natural'* (*snd y*) *ys'* = *ys*) (**is** *?lhs* ⟷ *?rhs*)
    and *y* # *ys* = *trace.natural' s xs*
    ⟷ (∃ *xs' ys'*. *xs* = *xs'* @ *y* # *ys'* ∧ *snd* ‘ *set xs'* ⊆ {*s*} ∧ *snd y* ≠ *s* ∧ *trace.natural'* (*snd y*) *ys'* = *ys*) (**is** *?thesis1*)
proof −
  show *?lhs* ⟷ *?rhs*
  proof(*rule iffI*)
    show *?lhs* ⟹ *?rhs*
    proof(*induct xs*)
      case (*Cons x xs*) **show** *?case*
      proof(*cases s* = *snd x*)
        case *True* **with** *Cons*
        obtain *xs' ys'*
          where *xs* = *xs'* @ *y* # *ys'* **and** *snd* ‘ *set xs'* ⊆ {*s*} **and** *snd y* ≠ *s*

```
        and trace.natural' (snd y) ys' = ys
       by auto
     with True show ?thesis by (simp add: exI[where x=x # xs'])
   qed (use Cons.prems in force)
  qed simp
  show ?rhs ⟹ ?lhs
   by (auto simp: trace.natural'.append trace.natural'.eq-Nil-conv)
 qed
 then show ?thesis1
  by (rule eq-commute-conv)
qed

lemma eq-append-conv:
 shows trace.natural' s xs = ys @ zs
   ⟷ (∃ ys' zs'. xs = ys' @ zs' ∧ trace.natural' s ys' = ys ∧ trace.natural' (trace.final' s ys) zs' = zs) (is ?lhs
= ?rhs)
  and ys @ zs = trace.natural' s xs
    ⟷ (∃ ys' zs'. xs = ys' @ zs' ∧ trace.natural' s ys' = ys ∧ trace.natural' (trace.final' s ys) zs' = zs) (is
?thesis1)
proof −
 show ?lhs ⟷ ?rhs
 proof(rule iffI)
  show ?lhs ⟹ ?rhs
  proof(induct ys arbitrary: s xs)
   case (Cons y ys s xs)
   from Cons.prems
   obtain ys' zs'
    where xs = ys' @ y # zs' and snd ‘ set ys' ⊆ {s}
     and snd y ≠ s and trace.natural' (snd y) zs' = ys @ zs
    by (clarsimp simp: trace.natural'.eq-Cons-conv)
   with Cons.hyps[where s=snd y and xs=zs'] show ?case
    by (clarsimp simp: trace.natural'.eq-Cons-conv) (metis append.assoc append-Cons)
  qed fastforce
  show ?rhs ⟹ ?lhs
   by (auto simp: trace.natural'.append)
 qed
 then show ?thesis1
  by (rule eq-commute-conv)
qed

lemma replicate:
 shows trace.natural' s (replicate i as) = (if snd as = s ∨ i = 0 then [] else [as])
by (auto simp: gr0-conv-Suc trace.natural'.eq-Nil-conv)

lemma map-natural':
 shows trace.natural' (sf s) (map (map-prod af sf) (trace.natural' s xs))
     = trace.natural' (sf s) (map (map-prod af sf) xs)
by (induct xs arbitrary: s; simp; metis)

lemma map-inj-on-sf:
 assumes inj-on sf (insert s (snd ‘ set xs))
 shows trace.natural' (sf s) (map (map-prod af sf) xs) = map (map-prod af sf) (trace.natural' s xs)
using assms
proof(induct xs arbitrary: s)
 case (Cons x xs s)
 from Cons.prems have sf (snd x) ≠ sf s if snd x ≠ s
  by (meson image-eqI inj-onD insert-iff list.set-intros(1) that)
 with Cons.prems show ?case
```

by (*auto intro*: *Cons.hyps*)
qed *simp*

lemma *amap-noop*:
assumes *trace.natural′ s xs* = *map* (*map-prod af id*) *zs*
shows *trace.natural′ s zs* = *zs*
using *assms* **by** (*induct xs arbitrary*: *s zs*) (*auto split*: *if-split-asm*)

lemma *take*:
shows ∃*j*≤*length xs*. *take i* (*trace.natural′ s xs*) = *trace.natural′ s* (*take j xs*)
proof(*induct xs arbitrary*: *s i*)
case (*Cons x xs s i*) **then show** *?case* **by** (*cases i*; *fastforce*)
qed *simp*

lemma *idle-prefix*:
assumes *snd* ‘ *set xs* ⊆ {*s*}
shows *trace.natural′ s* (*xs* @ *ys*) = *trace.natural′ s ys*
using *assms* **by** (*simp add*: *trace.natural′.append trace.natural′.eq-Nil-conv*)

lemma *prefixE*:
assumes *trace.natural′ s ys* = *trace.natural′ s* (*xs* @ *xsrest*)
obtains *xs′ xs′rest* **where** *trace.natural′ s xs* = *trace.natural′ s xs′* **and** *ys* = *xs′* @ *xs′rest*
by (*metis assms trace.natural′.eq-append-conv*(*2*))

lemma *aset-conv*:
shows *a* ∈ *trace.aset* (*trace.T s* (*trace.natural′ s xs*) *v*)
⟷ (∃ *s′ s″*. (*a*, *s′*, *s″*) ∈ *set* (*trace.transitions′ s xs*) ∧ *s′* ≠ *s″*)
by (*induct xs arbitrary*: *s*) (*auto simp*: *trace.aset.simps*)

setup ‹*Sign.parent-path*›

definition *natural* :: (′*a*, ′*s*, ′*v*) *trace.t* ⇒ (′*a*, ′*s*, ′*v*) *trace.t* (‹♮›) **where**
♮*σ* = *trace.T* (*trace.init σ*) (*trace.natural′* (*trace.init σ*) (*trace.rest σ*)) (*trace.term σ*)

setup ‹*Sign.mandatory-path natural*›

lemma *sel*[*simp*]:
shows *trace.init* (♮*σ*) = *trace.init σ*
and *trace.rest* (♮*σ*) = *trace.natural′* (*trace.init σ*) (*trace.rest σ*)
and *trace.term* (♮*σ*) = *trace.term σ*
by (*simp-all add*: *trace.natural-def*)

lemma *simps*:
shows ♮(*trace.T s* [] *v*) = *trace.T s* [] *v*
and ♮(*trace.T s* ((*a*, *s*) # *xs*) *v*) = ♮(*trace.T s xs v*)
and ♮(*trace.T s* (*trace.natural′ s xs*) *v*) = ♮(*trace.T s xs v*)
by (*simp-all add*: *trace.natural-def trace.natural′.natural′*)

lemma *idempotent*[*simp*]:
shows ♮(♮*σ*) = ♮*σ*
by (*simp add*: *trace.natural-def trace.natural′.natural′*)

lemma *idle*:
assumes *snd* ‘ *set xs* ⊆ {*s*}
shows ♮(*trace.T s xs v*) = *trace.T s* [] *v*
by (*simp add*: *assms trace.natural-def trace.natural′.eq-Nil-conv*)

lemma *trace-conv*:

shows $\natural(trace.T\ s\ xs\ v) = \natural\sigma \longleftrightarrow trace.init\ \sigma = s \wedge trace.natural'\ s\ xs = trace.natural'\ s\ (trace.rest\ \sigma) \wedge trace.term\ \sigma = v$
and $\natural\sigma = \natural(trace.T\ s\ xs\ v) \longleftrightarrow trace.init\ \sigma = s \wedge trace.natural'\ s\ xs = trace.natural'\ s\ (trace.rest\ \sigma) \wedge trace.term\ \sigma = v$
by (*cases* σ; *fastforce simp*: *trace.natural-def*)+

lemma *map-natural*:
shows $\natural(trace.map\ af\ sf\ vf\ (\natural\sigma)) = \natural(trace.map\ af\ sf\ vf\ \sigma)$
by (*simp add*: *trace.natural-def trace.natural'.map-natural'*)

lemma *continue*:
shows $\natural(\sigma\ @{-}_S\ xsv) = \natural\sigma\ @{-}_S\ (trace.natural'\ (trace.final\ \sigma)\ (fst\ xsv),\ snd\ xsv)$
by (*simp add*: *trace.natural-def trace.natural'.append split*: *option.split*)

lemma *replicate*:
shows $\natural(trace.T\ s\ (replicate\ i\ as)\ v)$
$= (trace.T\ s\ (if\ snd\ as = s \vee i = 0\ then\ []\ else\ [as])\ v)$
by (*simp add*: *trace.natural-def trace.natural'.replicate*)

lemma *monotone*:
shows *mono* $\natural$
using *trace.natural.continue* **by** (*fastforce intro*: *monoI simp*: *trace.less-eq-t-def*)

lemmas *strengthen*[*strg*] = *st-monotone*[*OF trace.natural.monotone*]
lemmas *mono* = *monotoneD*[*OF trace.natural.monotone*]
lemmas *mono2mono*[*cont-intro*, *partial-function-mono*]
= *monotone2monotone*[*OF trace.natural.monotone*, *simplified*, *of orda P* **for** *orda P*]

lemma *less-eqE*:
assumes $t \leq u$
assumes $\natural u' = \natural u$
obtains t' **where** $\natural t = \natural t'$ **and** $t' \leq u'$
using *assms*
by *atomize-elim*
(*fastforce simp*: *trace.natural-def trace.split-Ex trace.less-eq-None*(*2*)[*unfolded prefix-def*]
elim!: *trace.less-eqE prefixE trace.natural'.prefixE*)

lemma *less-eq-natural*:
assumes $\sigma_1 \leq \natural\sigma_2$
shows $\natural\sigma_1 = \sigma_1$
using *assms*
by (*cases* σ_1)
(*auto simp*: *trace.natural-def prefix-def trace.natural'.eq-append-conv trace.natural'.natural'*
elim!: *trace.less-eqE*)

lemma *map-le*:
assumes $\natural\sigma_1 \leq \natural\sigma_2$
shows $\natural(trace.map\ af\ sf\ vf\ \sigma_1) \leq \natural(trace.map\ af\ sf\ vf\ \sigma_2)$
using *trace.natural.mono*[*OF trace.map.mono*[*OF assms*], *simplified trace.natural.map-natural*] **.**

lemma *map-inj-on-sf*:
assumes *inj-on sf* (*trace.sset* σ)
shows $\natural(trace.map\ af\ sf\ vf\ \sigma) = trace.map\ af\ sf\ vf\ (\natural\sigma)$
using *assms* **by** (*cases* σ) (*simp add*: *trace.natural-def trace.natural'.map-inj-on-sf trace.sset.simps*)

lemma *take*:
shows $\exists j.\ \natural(trace.take\ i\ \sigma) = trace.take\ j\ (\natural\sigma)$
by (*meson trace.natural.mono trace.less-eq-take-def*)

lemma *take-natural*:
shows $\natural(\mathit{trace.take}\ i\ (\natural\sigma)) = \mathit{trace.take}\ i\ (\natural\sigma)$
using *trace.natural.less-eq-natural* **by** *blast*

lemma *takeE*:
shows $[\![\sigma_1 = \natural(\mathit{trace.take}\ i\ \sigma_2);\ \bigwedge j.\ [\![\sigma_1 = \mathit{trace.take}\ j\ (\natural\sigma_2)]\!] \Longrightarrow \mathit{thesis}]\!] \Longrightarrow \mathit{thesis}$
and $[\![\natural(\mathit{trace.take}\ i\ \sigma_2) = \sigma_1;\ \bigwedge j.\ [\![\sigma_1 = \mathit{trace.take}\ j\ (\natural\sigma_2)]\!] \Longrightarrow \mathit{thesis}]\!] \Longrightarrow \mathit{thesis}$
using *trace.natural.take* **by** *blast+*

setup ‹*Sign.parent-path*›

setup ‹*Sign.mandatory-path aset*›

lemma *natural-conv*:
shows $a \in \mathit{trace.aset}\ (\natural\sigma) \longleftrightarrow (\exists\, s\ s'.\ (a,\ s,\ s') \in \mathit{trace.steps}\ \sigma)$
by (*simp add*: *trace.natural-def trace.steps'-def trace.natural'.aset-conv*)

setup ‹*Sign.parent-path*›

setup ‹*Sign.mandatory-path sset*›

lemma *natural'*[*simp*]:
shows $\mathit{trace.sset}\ (\mathit{trace.T}\ s_0\ (\mathit{trace.natural'}\ s_0\ xs)\ v) = \mathit{trace.sset}\ (\mathit{trace.T}\ s_0\ xs\ v)$
by (*induct xs arbitrary*: s_0) (*simp-all add*: *trace.sset.simps*)

lemma *natural*[*simp*]:
shows $\mathit{trace.sset}\ (\natural\sigma) = \mathit{trace.sset}\ \sigma$
by (*simp add*: *trace.natural-def*)

setup ‹*Sign.parent-path*›

setup ‹*Sign.mandatory-path vset*›

lemma *natural*[*simp*]:
shows $\mathit{trace.vset}\ (\natural\sigma) = \mathit{trace.vset}\ \sigma$
by (*cases* σ) (*simp add*: *trace.natural-def trace.t.simps*(*8*))

setup ‹*Sign.parent-path*›

setup ‹*Sign.mandatory-path take*›

lemma *natural*:
shows $\exists j{\le}\mathit{Suc}\ (\mathit{length}\ (\mathit{trace.rest}\ \sigma)).\ \mathit{trace.take}\ i\ (\natural\sigma) = \natural(\mathit{trace.take}\ j\ \sigma)$
using *trace.natural'.take*[**where** $i{=}i$ **and** $s{=}\ \mathit{trace.init}\ \sigma$ **and** $xs{=}\mathit{trace.rest}\ \sigma$]
by (*auto simp*: *trace.natural-def trace.take-def not-le*) (*use Suc-n-not-le-n* **in** *blast*)

lemma *naturalE*:
shows $[\![\sigma_1 = \mathit{trace.take}\ i\ (\natural\sigma_2);\ \bigwedge j.\ [\![j \le \mathit{Suc}\ (\mathit{length}\ (\mathit{trace.rest}\ \sigma_2));\ \sigma_1 = \natural(\mathit{trace.take}\ j\ \sigma_2)]\!] \Longrightarrow \mathit{thesis}]\!] \Longrightarrow$
thesis
and $[\![\mathit{trace.take}\ i\ (\natural\sigma_2) = \sigma_1;\ \bigwedge j.\ [\![j \le \mathit{Suc}\ (\mathit{length}\ (\mathit{trace.rest}\ \sigma_2));\ \natural(\mathit{trace.take}\ j\ \sigma_2) = \sigma_1]\!] \Longrightarrow \mathit{thesis}]\!] \Longrightarrow$
thesis
using *trace.take.natural*[*of* σ_2 *i*] **by** *force+*

setup ‹*Sign.parent-path*›

lemma *steps'-alt-def*:
shows $\mathit{trace.steps'}\ s\ xs = \mathit{set}\ (\mathit{trace.transitions'}\ s\ (\mathit{trace.natural'}\ s\ xs))$

by (*induct xs arbitrary*: *s*) *auto*

setup ‹*Sign.mandatory-path steps'*›

lemma *natural'*:
  shows *trace.steps' s* (*trace.natural' s xs*) = *trace.steps' s xs*
unfolding *trace.steps'-def* **by** (*induct xs arbitrary*: *s*) *auto*

lemma *asetD*:
  assumes *trace.steps σ* ⊆ *r*
  shows ∀ *a*. *a* ∈ *trace.aset* (♮*σ*) ⟶ *a* ∈ *fst* ‘ *r*
using *assms* **by** (*force simp*: *trace.aset.natural-conv*)

lemma *range-initE*:
  assumes *trace.steps' s*$_0$ *xs* ⊆ *range af* × *range sf* × *range sf*
  assumes (*a*, *s*, *s'*) ∈ *trace.steps' s*$_0$ *xs*
  obtains *s*$_0$' **where** *s*$_0$ = *sf s*$_0$'
using *assms* **by** (*induct xs arbitrary*: *s s*$_0$) (*auto simp*: *trace.steps'-alt-def split*: *if-split-asm*)

lemma *map-range-conv*:
  shows *trace.steps'* (*sf s*) *xs* ⊆ *range af* × *range sf* × *range sf*
    ⟷ (∃ *xs'*. *trace.natural'* (*sf s*) *xs* = *map* (*map-prod af sf*) *xs'*) (**is** *?lhs* ⟷ *?rhs*)
proof(*rule iffI*)
  show *?lhs* ⟹ *?rhs*
    by (*induct xs arbitrary*: *s*) (*auto 0 3 simp*: *Cons-eq-map-conv*)
  show *?rhs* ⟹ *?lhs*
    by (*force simp*: *trace.steps'-alt-def trace.transitions'.map map-prod-conv*)
qed

lemma *step-conv*:
  shows *trace.steps' s xs* = {*x*}
    ⟷ *fst* (*snd x*) = *s* ∧ *fst* (*snd x*) ≠ *snd* (*snd x*)
      ∧ (∃ *ys zs*. *snd* ‘ *set ys* ⊆ {*s*} ∧ *snd* ‘ *set zs* ⊆ {*snd* (*snd x*)}
          ∧ *xs* = *ys* @ [(*fst x*, *snd* (*snd x*))] @ *zs*) (**is** *?lhs* ⟷ *?rhs*)
proof(*rule iffI*)
  show *?lhs* ⟹ *?rhs*
    by (*fastforce dest*!: *arg-cong*[**where** *f*=*set*]
        simp: *trace.steps'-alt-def set-singleton-conv set-replicate-conv-if*
          trace.transitions'.eq-Nil-conv trace.transitions'.eq-Cons-conv
          trace.natural'.eq-Nil-conv trace.natural'.eq-Cons-conv
        split: *if-split-asm*)
  show *?rhs* ⟹ *?lhs*
    by (*clarsimp simp*: *trace.steps'.append*)
qed

setup ‹*Sign.parent-path*›

interpretation *stuttering*: *galois.image-vimage-idempotent* ♮
by (*simp add*: *galois.image-vimage-idempotent.intro*)

abbreviation *stuttering-equiv-syn* :: ('*a*, '*s*, '*v*) *trace.t* ⇒ ('*a*, '*s*, '*v*) *trace.t* ⇒ *bool* (**infix** ‹≃$_S$› *50*) **where**
  σ$_1$ ≃$_S$ *σ*$_2$ ≡ *trace.stuttering.equivalent σ*$_1$ *σ*$_2$

setup ‹*Sign.mandatory-path stuttering*›

setup ‹*Sign.mandatory-path cl*›

setup ‹*Sign.mandatory-path downwards*›

lemma *cl*:
shows *trace.stuttering.cl* (*downwards.cl P*) = *downwards.cl* (*trace.stuttering.cl P*) (**is** *?lhs* = *?rhs*)
proof(*rule antisym*)
show *?lhs* ⊆ *?rhs*
by (*clarsimp simp*: *trace.stuttering.cl-alt-def downwards.cl-def trace.less-eq-t-def*)
(*metis trace.final'.natural' trace.natural.continue trace.natural.sel*(*1*,*2*))
next
show *?rhs* ⊆ *?lhs*
by (*clarsimp elim*!: *downwards.clE trace.stuttering.clE*)
(*erule* (*1*) *trace.natural.less-eqE*; *fastforce*)
qed

lemma *closed*:
assumes *P* ∈ *downwards.closed*
shows *trace.stuttering.cl P* ∈ *downwards.closed*
by (*metis assms downwards.closedI downwards.closed-conv trace.stuttering.cl.downwards.cl*)

setup ‹*Sign.parent-path*›

setup ‹*Sign.mandatory-path closed*›

lemma *downwards-imp*: — Abadi and Plotkin (1993, p13)
assumes *P* ∈ *trace.stuttering.closed*
assumes *Q* ∈ *trace.stuttering.closed*
shows *downwards.imp P Q* ∈ *trace.stuttering.closed*
by (*simp add*: *assms downwards.closed-imp downwards.heyting-imp downwards.imp-mp*
trace.stuttering.cl.downwards.closed trace.stuttering.closed-clI
trace.stuttering.exchange-closed-inter trace.stuttering.least)

setup ‹*Sign.parent-path*›

setup ‹*Sign.parent-path*›

setup ‹*Sign.mandatory-path equiv*›

lemma *simps*:
shows *snd* ‘ *set xs* ⊆ {*s*} ⟹ *trace.T s* (*xs* @ *ys*) *v* ≃*S trace.T s ys v*
and *snd* ‘ *set ys* ⊆ {*trace.final' s xs*} ⟹ *trace.T s* (*xs* @ *ys*) *v* ≃*S trace.T s xs v*
and *snd* ‘ *set xs* ⊆ {*snd x*} ⟹ *trace.T s* (*x* # *xs* @ *ys*) *v* ≃*S trace.T s* (*x* # *ys*) *v*
by (*fastforce simp*: *trace.natural-def trace.natural'.append trace.natural'.eq-Nil-conv*)+

lemma *append-cong*:
assumes *s* = *s'*
assumes *trace.natural' s xs* = *trace.natural' s xs'*
assumes *trace.natural'* (*trace.final' s xs*) *ys* = *trace.natural'* (*trace.final' s xs*) *ys'*
assumes *v* = *v'*
shows *trace.T s* (*xs* @ *ys*) *v* ≃*S trace.T s'* (*xs'* @ *ys'*) *v'*
using *assms* **by** (*simp add*: *trace.natural-def trace.natural'.append cong*: *trace.final'.natural'-cong*)

lemma *E*:
assumes *trace.T s xs v* ≃*S trace.T s' xs' v'*
obtains *trace.natural' s xs* = *trace.natural' s' xs'* **and** *s* = *s'* **and** *v* = *v'*
using *assms* **by** (*fastforce simp*: *trace.natural-def*)

lemma *append-conv*:
shows *trace.T s* (*xs* @ *ys*) *v* ≃*S σ*
⟷ (∃ *xs' ys'*. *σ* = *trace.T s* (*xs'* @ *ys'*) *v* ∧ *trace.natural' s xs* = *trace.natural' s xs'*

$\wedge$ *trace.natural′* (*trace.final′ s xs*) *ys* = *trace.natural′* (*trace.final′ s xs*) *ys′*) (**is** *?thesis1*)
and $\sigma \simeq_S$ *trace.T s* (*xs* @ *ys*) *v*
$\longleftrightarrow$ ($\exists$ *xs′ ys′*. σ = *trace.T s* (*xs′* @ *ys′*) *v* $\wedge$ *trace.natural′ s xs* = *trace.natural′ s xs′*
$\wedge$ *trace.natural′* (*trace.final′ s xs*) *ys* = *trace.natural′* (*trace.final′ s xs*) *ys′*) (**is** *?thesis2*)
proof −
show *?thesis1*
by (*cases* σ)
(*fastforce simp*: *trace.natural′.append trace.natural′.eq-append-conv*
elim: *trace.stuttering.equiv.E*
intro: *trace.stuttering.equiv.append-cong*)
then show *?thesis2*
by (*rule eq-commute-conv*)
qed

lemma *map*:
assumes $\sigma_1 \simeq_S \sigma_2$
shows *trace.map af sf vf* $\sigma_1 \simeq_S$ *trace.map af sf vf* σ_2
by (*metis assms trace.natural.map-natural*)

lemma *steps*:
assumes $\sigma_1 \simeq_S \sigma_2$
shows *trace.steps* σ_1 = *trace.steps* σ_2
using *assms* **by** (*force simp*: *trace.steps′-alt-def trace.natural-def*)

setup ‹*Sign.parent-path*›

setup ‹*Sign.parent-path*›

setup ‹*Sign.parent-path*›

8.2 The *('a, 's, 'v) spec* lattice

Our workhorse lattice consists of all sets of traces that are downwards and stuttering closed. This combined closure is neither matroidal nor antimatroidal (§5.3).

We define the lattice as a type and instantiate the relevant type classes. In the following read $P \leq Q$ ($P \subseteq Q$ in the powerset model) as "Q follows from P" or "P entails Q".

setup ‹*Sign.mandatory-path raw*›

setup ‹*Sign.mandatory-path spec*›

definition *cl* :: ('*a*, '*s*, '*v*) *trace.t set* ⇒ ('*a*, '*s*, '*v*) *trace.t set* **where**
cl P = *downwards.cl* (*trace.stuttering.cl P*)

setup ‹*Sign.parent-path*›

interpretation *spec*: *closure-powerset-distributive raw.spec.cl*
unfolding *raw.spec.cl-def*
by (*simp add*: *closure-powerset-distributive-comp downwards.closure-powerset-distributive-axioms*
trace.stuttering.closure-powerset-distributive-axioms trace.stuttering.cl.downwards.cl)

setup ‹*Sign.mandatory-path spec*›

setup ‹*Sign.mandatory-path cl*›

lemma *empty*[*simp*]:
shows *raw.spec.cl* {} = {}
by (*simp add*: *raw.spec.cl-def downwards.cl-empty trace.stuttering.cl-bot*)

setup ‹*Sign.parent-path*›

setup ‹*Sign.mandatory-path closed*›

lemma *I*:
  assumes $P \in$ *downwards.closed*
  assumes $P \in$ *trace.stuttering.closed*
  shows $P \in$ *raw.spec.closed*
by (*metis assms raw.spec.cl-def downwards.closed-conv raw.spec.closed trace.stuttering.closed-conv*)

lemma *empty*[*intro*]:
  shows $\{\} \in$ *raw.spec.closed*
using *raw.spec.cl.empty* **by** *blast*

lemma *downwards-closed*:
  assumes $P \in$ *raw.spec.closed*
  shows $P \in$ *downwards.closed*
by (*metis assms downwards.closed raw.spec.cl-def raw.spec.closed-conv*)

lemma *stuttering-closed*:
  assumes $P \in$ *raw.spec.closed*
  shows $P \in$ *trace.stuttering.closed*
using *assms raw.spec.cl-def raw.spec.closed-conv* **by** *fast*

lemma *downwards-imp*:
  assumes $P \in$ *raw.spec.closed*
  assumes $Q \in$ *raw.spec.closed*
  shows *downwards.imp P Q* $\in$ *raw.spec.closed*
by (*meson assms downwards.closed-imp raw.spec.closed.I raw.spec.closed.stuttering-closed*
    trace.stuttering.cl.closed.downwards-imp)

lemma *heyting-downwards-imp*:
  assumes $P \in$ *raw.spec.closed*
  shows $P \subseteq$ *downwards.imp Q R* $\longleftrightarrow P \cap Q \subseteq R$
by (*simp add*: *assms downwards.heyting-imp raw.spec.closed.downwards-closed*)

lemma *takeE*:
  assumes $\sigma \in P$
  assumes $P \in$ *raw.spec.closed*
  shows *trace.take i* $\sigma \in P$
by (*meson assms downwards.closed-in raw.spec.closed.downwards-closed trace.less-eq-take*)

setup ‹*Sign.parent-path*›

setup ‹*Sign.parent-path*›

setup ‹*Sign.parent-path*›

typedef (*'a*, *'s*, *'v*) *spec* = *raw.spec.closed* :: (*'a*, *'s*, *'v*) *trace.t set set*
morphisms *unMkS MkS*
by *blast*

setup-lifting *type-definition-spec*

instantiation *spec* :: (*type*, *type*, *type*) *complete-distrib-lattice*
begin

lift-definition *bot-spec* :: ('*a*, '*s*, '*v*) *spec* **is** *empty* **..**
lift-definition *top-spec* :: ('*a*, '*s*, '*v*) *spec* **is** *UNIV* **..**
lift-definition *sup-spec* :: ('*a*, '*s*, '*v*) *spec* ⇒ ('*a*, '*s*, '*v*) *spec* ⇒ ('*a*, '*s*, '*v*) *spec* **is** *sup* **..**
lift-definition *inf-spec* :: ('*a*, '*s*, '*v*) *spec* ⇒ ('*a*, '*s*, '*v*) *spec* ⇒ ('*a*, '*s*, '*v*) *spec* **is** *inf* **..**
lift-definition *less-eq-spec* :: ('*a*, '*s*, '*v*) *spec* ⇒ ('*a*, '*s*, '*v*) *spec* ⇒ *bool* **is** *less-eq* **.**
lift-definition *less-spec* :: ('*a*, '*s*, '*v*) *spec* ⇒ ('*a*, '*s*, '*v*) *spec* ⇒ *bool* **is** *less* **.**
lift-definition *Inf-spec* :: ('*a*, '*s*, '*v*) *spec set* ⇒ ('*a*, '*s*, '*v*) *spec* **is** *Inf* **..**
lift-definition *Sup-spec* :: ('*a*, '*s*, '*v*) *spec set* ⇒ ('*a*, '*s*, '*v*) *spec* **is** λ*X*. *Sup X* ⊔ *raw.spec.cl* {} **..**

instance
by (*standard*; *transfer*; *auto simp*: *raw.spec.closed-strict-complete-distrib-lattice-axiomI*[*OF raw.spec.cl.empty*])

end

declare
SUPE[**where** '*a*=('*a*, '*s*, '*v*) *spec*, *intro*!]
SupE[**where** '*a*=('*a*, '*s*, '*v*) *spec*, *intro*!]
Sup-le-iff[**where** '*a*=('*a*, '*s*, '*v*) *spec*, *simp*]
SupI[**where** '*a*=('*a*, '*s*, '*v*) *spec*, *intro*]
SUPI[**where** '*a*=('*a*, '*s*, '*v*) *spec*, *intro*]
rev-SUPI[**where** '*a*=('*a*, '*s*, '*v*) *spec*, *intro?*]
INFE[**where** '*a*=('*a*, '*s*, '*v*) *spec*, *intro*]

Observations about this type:

- it is not a BNF (datatype) as it uses the powerset
- it fails to be T0 or sober due to the lack of limit points (completeness) in ('*a*, '*s*, '*v*) *trace.t*
  - also stuttering closure precludes T0
- the *complete-distrib-lattice* instance shows that arbitrary/infinitary *Sup*s and *Inf*s distribute
  - in other words: safety properties are closed under arbitrary intersections and unions
  - in other words: Alexandrov
- conclude: the lack of limit points makes this model easier to work in and adds expressivity
  - see §24 for further discussion

setup ‹*Sign.mandatory-path spec*›

lemmas *antisym* = *antisym*[**where** '*a*=('*a*, '*s*, '*v*) *spec*]
lemmas *eq-iff* = *order.eq-iff*[**where** '*a*=('*a*, '*s*, '*v*) *spec*]

setup ‹*Sign.parent-path*›

8.3 Irreducible elements

The irreducible elements of ('*a*, '*s*, '*v*) *trace.t* are the closures of singletons.

setup ‹*Sign.mandatory-path raw*›

definition *singleton* :: ('*a*, '*s*, '*v*) *trace.t* ⇒ ('*a*, '*s*, '*v*) *trace.t set* **where**
singleton σ = *raw.spec.cl* {σ}

lemma *singleton-le-conv*:
shows *raw.singleton* $\sigma_1 \leq$ *raw.singleton* $\sigma_2 \longleftrightarrow \natural\sigma_1 \leq \natural\ \sigma_2$ (**is** *?lhs* ⟷ *?rhs*)
proof(*rule iffI*)

assume *?lhs*
then have $\sigma \in$ *downwards.cl* $\{\natural\ \sigma_2\}$ **if** $\sigma \leq \natural\ \sigma_1$ **for** σ
using *that trace.natural.mono*
by (*force simp*: *raw.singleton-def raw.spec.cl-def*
intro: *downwards.clI*[**where** $y{=}\natural\sigma_1$]
elim!: *downwards.clE trace.stuttering.clE*
dest!: *subsetD*[**where** $c{=}\sigma$]
dest: *trace.natural.less-eq-natural*)
then show *?rhs*
by (*fastforce simp flip*: *downwards.order-embedding*[**where** $x{=}\natural\sigma_1$]
elim: *downwards.clE trace.stuttering.clE*)
next
show *?rhs* $\Longrightarrow$ *?lhs*
by (*clarsimp simp*: *raw.singleton-def raw.spec.cl-def*)
(*metis downwards.order-embedding trace.natural.idempotent trace.stuttering.cl.downwards.cl*
trace.stuttering.cl-mono trace.stuttering.equiv-cl-singleton)
qed

setup ‹*Sign.parent-path*›

setup ‹*Sign.mandatory-path spec*›

lift-definition *singleton* :: ('*a*, '*s*, '*v*) *trace.t* ⇒ ('*a*, '*s*, '*v*) *spec* (‹⦇-⦈›) **is** *raw.singleton*
by (*simp add*: *raw.singleton-def*)

abbreviation *singleton-trace-syn* :: '*s* ⇒ ('*a* × '*s*) *list* ⇒ '*v option* ⇒ ('*a*, '*s*, '*v*) *spec* (‹⦇-, -, -⦈›) **where**
⦇*s*, *xs*, *v*⦈ ≡ ⦇*trace.T s xs v*⦈

setup ‹*Sign.mandatory-path singleton*›

lemma *Sup-prime*:
shows *Sup-prime* ⦇σ⦈
by (*clarsimp simp*: *Sup-prime-on-def*)
(*transfer*; *auto simp*: *raw.singleton-def elim*!: *Sup-prime-onE*[*OF raw.spec.Sup-prime-on-singleton*])

lemma *nchotomy*:
shows ∃ *X*∈*raw.spec.closed*. *x* = ⨆(*spec.singleton* ‘ *X*)
by *transfer*
(*use raw.spec.closed-conv* **in** ‹*auto simp*: *raw.singleton-def*
simp flip: *raw.spec.distributive*[*simplified*]›)

lemmas *exhaust* = *bexE*[*OF spec.singleton.nchotomy*]

lemma *collapse*[*simp*]:
shows ⨆(*spec.singleton* ‘ {σ. ⦇σ⦈ ≤ *P*}) = *P*
by (*rule spec.singleton.exhaust*[*of P*]; *blast intro*: *antisym*)

lemmas *not-bot* = *Sup-prime-not-bot*[*OF spec.singleton.Sup-prime*] — Non-triviality

setup ‹*Sign.parent-path*›

lemma *singleton-le-ext-conv*:
shows *P* ≤ *Q* ⟷ (∀σ. ⦇σ⦈ ≤ *P* ⟶ ⦇σ⦈ ≤ *Q*) (**is** *?lhs* ⟷ *?rhs*)
proof(*rule iffI*)
show *?rhs* $\Longrightarrow$ *?lhs*
by (*rule spec.singleton.exhaust*[**where** *x*=*P*]; *rule spec.singleton.exhaust*[**where** *x*=*Q*]; *blast*)
qed *fastforce*

lemmas *singleton-le-conv* = *raw.singleton-le-conv*[*transferred*]
lemmas *singleton-le-extI* = *iffD2*[*OF spec.singleton-le-ext-conv*, *rule-format*]

lemma *singleton-eq-conv*[*simp*]:
  shows $\langle\!|\sigma|\!\rangle = \langle\!|\sigma'|\!\rangle \longleftrightarrow \sigma \simeq_S \sigma'$
by (*auto simp*: *spec.eq-iff spec.singleton-le-conv*)

lemma *singleton-cong*:
  assumes $\sigma \simeq_S \sigma'$
  shows $\langle\!|\sigma|\!\rangle = \langle\!|\sigma'|\!\rangle$
using *assms* **by** *simp*

setup ‹*Sign.mandatory-path singleton*›

named-theorems *le-conv* ‹ *simplification rules for* ‹$\langle\!|\sigma|\!\rangle \leq$ *const* ...› ›

lemmas *antisym* = *antisym*[*OF spec.singleton-le-extI spec.singleton-le-extI*]

lemmas *top* = *spec.singleton.collapse*[*of* $\top$, *simplified*, *symmetric*]

lemma *monotone*:
  shows *mono spec.singleton*
by (*simp add*: *monoI trace.natural.mono spec.singleton-le-conv*)

lemmas *strengthen*[*strg*] = *st-monotone*[*OF spec.singleton.monotone*]
lemmas *mono* = *monoD*[*OF spec.singleton.monotone*]
lemmas *mono2mono*[*cont-intro*, *partial-function-mono*]
  = *monotone2monotone*[*OF spec.singleton.monotone*, *simplified*]

lemma *simps*[*simp*]:
  shows $\langle\!|\natural\sigma|\!\rangle = \langle\!|\sigma|\!\rangle$
    and $\langle\!|s, xs, v|\!\rangle \leq \langle\!|s,$ *trace.natural'* $s\ xs, v|\!\rangle$
    and *snd* ‘ *set xs* $\subseteq \{s\} \Longrightarrow \langle\!|s, xs$ @ $ys, v|\!\rangle = \langle\!|s, ys, v|\!\rangle$
    and *snd* ‘ *set ys* $\subseteq \{$*trace.final' s xs*$\} \Longrightarrow \langle\!|s, xs$ @ $ys, v|\!\rangle = \langle\!|s, xs, v|\!\rangle$
    and *snd* ‘ *set xs* $\subseteq \{$*snd x*$\} \Longrightarrow \langle\!|s, x \# xs$ @ $ys, v|\!\rangle = \langle\!|s, x \# ys, v|\!\rangle$
    and $\langle\!|s, (a, s) \# xs, v|\!\rangle = \langle\!|s, xs, v|\!\rangle$
by (*simp-all add*: *antisym spec.singleton-le-conv trace.stuttering.equiv.simps trace.natural.simps*)

lemma *Cons*: — self-applies, not usable by *simp*
  assumes *snd* ‘ *set as* $\subseteq \{s'\}$
  shows $\langle\!|s, (a, s') \# as, v|\!\rangle = \langle\!|s, [(a, s')], v|\!\rangle$
by (*simp add*: *assms spec.singleton.simps*(*4*)[**where** *xs*=$[(a, s')]$ **and** *ys*=*as*, *simplified*])

lemmas *Sup-irreducible* = *iffD1*[*OF heyting.Sup-prime-Sup-irreducible-iff spec.singleton.Sup-prime*]
lemmas *sup-irreducible* = *Sup-irreducible-on-imp-sup-irreducible-on*[*OF spec.singleton.Sup-irreducible*, *simplified*]
lemmas *Sup-leE*[*elim*] = *Sup-prime-onE*[*OF spec.singleton.Sup-prime*, *simplified*]
lemmas *sup-le-conv*[*simp*] = *sup-irreducible-le-conv*[*OF spec.singleton.sup-irreducible*]
lemmas *Sup-le-conv*[*simp*] = *Sup-prime-on-conv*[*OF spec.singleton.Sup-prime*, *simplified*]
lemmas *compact-point* = *Sup-prime-is-compact*[*OF spec.singleton.Sup-prime*]
lemmas *compact*[*cont-intro*] = *compact-points-are-ccpo-compact*[*OF spec.singleton.compact-point*]

lemma *Inf*:
  shows $\bigsqcap$(*spec.singleton* ‘ *X*) = $\bigsqcup$(*spec.singleton* ‘ $\{\sigma.\ \forall \sigma_1 \in X.\ \sigma \leq \natural\sigma_1\}$)
by (*fastforce simp*: *le-INF-iff spec.singleton-le-conv*
    dest: *spec.singleton.mono*
    intro: *spec.singleton.antisym*)

lemmas *inf* = *spec.singleton.Inf*[**where** *X*=$\{\sigma_1, \sigma_2\}$, *simplified*] **for** $\sigma_1\ \sigma_2$

lemma *less-eq-Some*[*simp*]:
  shows $\langle\!|s, xs, Some\ v|\!\rangle \leq \langle\!|\sigma|\!\rangle$
    $\longleftrightarrow$ *trace.term* σ = *Some* $v \wedge$ *trace.init* $\sigma = s \wedge$ *trace.natural'* s (*trace.rest* σ) = *trace.natural'* s *xs*
by (*auto simp*: *spec.singleton-le-conv trace.natural-def*)

lemma *less-eq-None*:
  shows [*iff*]: $\langle\!|s, xs, None|\!\rangle \leq \langle\!|s, xs, v'|\!\rangle$
by (*auto simp*: *spec.singleton-le-conv trace.natural-def trace.less-eq-None*)

lemma *map-cong*:
  assumes $\bigwedge a.\ a \in$ *trace.aset* ($\natural\sigma'$) $\Longrightarrow$ *af* a = *af'* a
  assumes $\bigwedge x.\ x \in$ *trace.sset* ($\natural\sigma'$) $\Longrightarrow$ *sf* x = *sf'* x
  assumes $\bigwedge v.\ v \in$ *trace.vset* ($\natural\sigma'$) $\Longrightarrow$ *vf* v = *vf'* v
  assumes $\natural\sigma = \natural\sigma'$
  shows $\langle\!|$*trace.map af sf vf* $\sigma|\!\rangle$ = $\langle\!|$*trace.map af' sf' vf'* $\sigma'|\!\rangle$
proof −
  from *assms* **have** *trace.map af sf vf* ($\natural$ σ) $\simeq_S$ *trace.map af' sf' vf'* ($\natural$ σ')
    by (*simp del*: *trace.sset.natural trace.vset.natural cong*: *trace.t.map-cong*)
  then show *?thesis*
    by (*simp add*: *trace.natural.map-natural*)
qed

lemma *map-le*:
  assumes $\langle\!|\sigma|\!\rangle \leq \langle\!|\sigma'|\!\rangle$
  shows $\langle\!|$*trace.map af sf vf* $\sigma|\!\rangle \leq \langle\!|$*trace.map af sf vf* $\sigma'|\!\rangle$
using *assms* **by** (*simp add*: *spec.singleton-le-conv trace.natural.map-le*)

lemma *takeI*:
  assumes $\langle\!|\sigma|\!\rangle \leq P$
  shows $\langle\!|$*trace.take i* $\sigma|\!\rangle \leq P$
by (*meson assms order.trans spec.singleton.mono trace.less-eq-take*)

setup ‹*Sign.parent-path*›

lemmas *assms-cong* = *order.assms-cong*[**where** *'a*=(*'a*, *'s*, *'v*) *spec*]
lemmas *concl-cong* = *order.concl-cong*[**where** *'a*=(*'a*, *'s*, *'v*) *spec*]

declare *spec.singleton.transfer*[*transfer-rule del*]

setup ‹*Sign.parent-path*›

8.4 Maps

Lift *trace.map* to the (*'a*, *'s*, *'v*) *spec* lattice via image and inverse image.

Note that the image may yield a set that is not stuttering closed (i.e., we need to close the obvious model-level definition of *spec.map* under stuttering) as arbitrary *sf* may introduce stuttering not present in *P*. In contrast the inverse image preserves stuttering. These issues are elided here through the use of *spec.singleton*.

setup ‹*Sign.mandatory-path spec*›

definition *map* :: (*'a* ⇒ *'b*) ⇒ (*'s* ⇒ *'t*) ⇒ (*'v* ⇒ *'w*) ⇒ (*'a*, *'s*, *'v*) *spec* ⇒ (*'b*, *'t*, *'w*) *spec* **where**
  map af sf vf P = $\bigsqcup$(*spec.singleton* ‘ *trace.map af sf vf* ‘ {σ. $\langle\!|\sigma|\!\rangle \leq P$})

definition *invmap* :: (*'a* ⇒ *'b*) ⇒ (*'s* ⇒ *'t*) ⇒ (*'v* ⇒ *'w*) ⇒ (*'b*, *'t*, *'w*) *spec* ⇒ (*'a*, *'s*, *'v*) *spec* **where**
  invmap af sf vf P = $\bigsqcup$(*spec.singleton* ‘ *trace.map af sf vf* − ‘ {σ. $\langle\!|\sigma|\!\rangle \leq P$})

abbreviation *amap* ::(*'a* ⇒ *'b*) ⇒ (*'a*, *'s*, *'v*) *spec* ⇒ (*'b*, *'s*, *'v*) *spec* **where**

amap af ≡ *spec.map af id id*
abbreviation *ainvmap* ::('*a* ⇒ '*b*) ⇒ ('*b*, '*s*, '*v*) *spec* ⇒ ('*a*, '*s*, '*v*) *spec* **where**
ainvmap af ≡ *spec.invmap af id id*
abbreviation *smap* ::('*s* ⇒ '*t*) ⇒ ('*a*, '*s*, '*v*) *spec* ⇒ ('*a*, '*t*, '*v*) *spec* **where**
smap sf ≡ *spec.map id sf id*
abbreviation *sinvmap* ::('*s* ⇒ '*t*) ⇒ ('*a*, '*t*, '*v*) *spec* ⇒ ('*a*, '*s*, '*v*) *spec* **where**
sinvmap sf ≡ *spec.invmap id sf id*
abbreviation *vmap* ::('*v* ⇒ '*w*) ⇒ ('*a*, '*s*, '*v*) *spec* ⇒ ('*a*, '*s*, '*w*) *spec* **where** — aka *liftM*
vmap vf ≡ *spec.map id id vf*
abbreviation *vinvmap* ::('*v* ⇒ '*w*) ⇒ ('*a*, '*s*, '*w*) *spec* ⇒ ('*a*, '*s*, '*v*) *spec* **where**
vinvmap vf ≡ *spec.invmap id id vf*

interpretation *map-invmap*: *galois.complete-lattice-distributive-class*
spec.map af sf vf
spec.invmap af sf vf **for** *af sf vf*
proof *standard*
show *spec.map af sf vf P* ≤ *Q* ⟷ *P* ≤ *spec.invmap af sf vf Q* (**is** *?lhs* ⟷ *?rhs*) **for** *P Q*
proof(*rule iffI*)
show *?lhs* ⟹ *?rhs*
by (*fastforce simp*: *spec.map-def spec.invmap-def intro*: *spec.singleton-le-extI*)
show *?rhs* ⟹ *?lhs*
by (*fastforce simp*: *spec.map-def spec.invmap-def*
dest: *order.trans*[*of - P*] *spec.singleton.map-le*[**where** *af*=*af* **and** *sf*=*sf* **and** *vf*=*vf*])
qed
show *spec.invmap af sf vf* (⨆ *X*) ≤ ⨆(*spec.invmap af sf vf* ‘ *X*) **for** *X*
by (*fastforce simp*: *spec.invmap-def*)
qed

setup ‹*Sign.mandatory-path singleton*›

lemma *map-le-conv*[*spec.singleton.le-conv*]:
shows ⟪*σ*⟫ ≤ *spec.map af sf vf P* ⟷ (∃ *σ'*. ⟪*σ'*⟫ ≤ *P* ∧ ⟪*σ*⟫ ≤ ⟪*trace.map af sf vf σ'*⟫)
by (*simp add*: *spec.map-def*)

lemma *invmap-le-conv*[*spec.singleton.le-conv*]:
shows ⟪*σ*⟫ ≤ *spec.invmap af sf vf P* ⟷ ⟪*trace.map af sf vf σ*⟫ ≤ *P*
by (*simp add*: *spec.invmap-def*) (*meson order.refl order.trans spec.singleton.map-le*)

setup ‹*Sign.parent-path*›

setup ‹*Sign.mandatory-path map*›

lemmas *bot* = *spec.map-invmap.lower-bot*

lemmas *monotone* = *spec.map-invmap.monotone-lower*
lemmas *mono* = *monotoneD*[*OF spec.map.monotone*]

lemmas *Sup* = *spec.map-invmap.lower-Sup*
lemmas *sup* = *spec.map-invmap.lower-sup*

lemmas *Inf-le* = *spec.map-invmap.lower-Inf-le* — Converse does not hold
lemmas *inf-le* = *spec.map-invmap.lower-inf-le* — Converse does not hold

lemmas *invmap-le* = *spec.map-invmap.lower-upper-contractive*

lemma *singleton*:
shows *spec.map af sf vf* ⟪*σ*⟫ = ⟪*trace.map af sf vf σ*⟫
by (*auto simp*: *spec.map-def spec.eq-iff spec.singleton-le-conv intro*: *trace.natural.map-le*)

lemma *top*:
assumes *surj af*
assumes *surj sf*
assumes *surj vf*
shows *spec.map af sf vf* $\top = \top$
by (*rule antisym*)
(*auto simp*: *assms spec.singleton.top spec.map.Sup spec.map.singleton surj-f-inv-f*
intro: *exI*[**where** *x*=*trace.map* (*inv af*) (*inv sf*) (*inv vf*) σ **for** σ])

lemma *id*:
shows *spec.map id id id P* = *P*
and *spec.map* ($\lambda x.\ x$) ($\lambda x.\ x$) ($\lambda x.\ x$) *P* = *P*
by (*simp-all add*: *spec.map-def flip*: *id-def*)

lemma *comp*:
shows *spec.map af sf vf* ∘ *spec.map ag sg vg* = *spec.map* (*af* ∘ *ag*) (*sf* ∘ *sg*) (*vf* ∘ *vg*) (**is** *?lhs* = *?rhs*)
and *spec.map af sf vf* (*spec.map ag sg vg P*) = *spec.map* ($\lambda a.$ *af* (*ag a*)) ($\lambda s.$ *sf* (*sg s*)) ($\lambda v.$ *vf* (*vg v*)) *P* (**is** *?thesis1*)
proof −
have *?lhs P* = *?rhs P* **for** *P*
by (*rule spec.singleton.exhaust*[**where** *x*=*P*])
(*simp add*: *spec.map.Sup spec.map.singleton image-image comp-def*)
then show *?lhs* = *?rhs* **and** *?thesis1* **by** (*simp-all add*: *comp-def*)
qed

lemmas *map* = *spec.map.comp*

lemma *inf-distr*:
shows *spec.map af sf vf P* ⊓ *Q* = *spec.map af sf vf* (*P* ⊓ *spec.invmap af sf vf Q*) (**is** *?lhs* = *?rhs*)
and *Q* ⊓ *spec.map af sf vf P* = *spec.map af sf vf* (*spec.invmap af sf vf Q* ⊓ *P*) (**is** *?thesis1*)
proof −
show *?lhs* = *?rhs*
proof(*rule antisym*)
obtain *X* **where** *Q*: *Q* = ⨆ (*spec.singleton* ‘ *X*) **using** *spec.singleton.nchotomy*[*of Q*] **by** *blast*
then have ∗: ⦇*trace.take j* (♮ σ_Q)⦈ ≤ *?rhs*
if ⦇σ_P⦈ ≤ *P*
and $\sigma_Q \in X$
and ♮ (*trace.take i* (*trace.map af sf vf* σ_P)) = *trace.take j* (♮ σ_Q)
for σ_P σ_Q *i j*
using *that*
by (*auto simp*: *spec.singleton.le-conv spec.singleton-le-conv heyting.inf-SUP-distrib*
spec.map-def spec.singleton.inf trace.less-eq-take-def
trace.take.map spec.singleton.takeI trace.take.take trace.natural.take-natural
intro!: *exI*[**where** *x*=*trace.take i* σ_P] *exI*[**where** *x*=*j*])
with *Q* **show** *?lhs* ≤ *?rhs*
by (*subst spec.map-def*)
(*fastforce simp*: *heyting.inf-SUP-distrib spec.singleton.inf trace.less-eq-take-def*
elim: *trace.take.naturalE*(*2*))
show *?rhs* ≤ *?lhs*
by (*simp add*: *le-infI1 spec.map-invmap.galois spec.map-invmap.upper-lower-expansive*)
qed
then show *?thesis1*
by (*simp add*: *inf.commute*)
qed

setup ‹*Sign.parent-path*›

setup ‹*Sign.mandatory-path smap*›

lemma *comp*:
  shows *spec.smap sf* ∘ *spec.smap sg* = *spec.smap* (*sf* ∘ *sg*)
    and *spec.smap sf* (*spec.smap sg P*) = *spec.smap* (λ*s*. *sf* (*sg s*)) *P*
by (*simp-all add*: *comp-def spec.map.comp id-def*)

setup ‹*Sign.parent-path*›

setup ‹*Sign.mandatory-path invmap*›

lemmas *bot* = *spec.map-invmap.upper-bot*
lemmas *top* = *spec.map-invmap.upper-top*

lemmas *monotone* = *spec.map-invmap.monotone-upper*
lemmas *mono* = *monotoneD*[*OF spec.invmap.monotone*]

lemmas *Sup* = *spec.map-invmap.upper-Sup*
lemmas *sup* = *spec.map-invmap.upper-sup*

lemmas *Inf* = *spec.map-invmap.upper-Inf*
lemmas *inf* = *spec.map-invmap.upper-inf*

lemma *singleton*:
  shows *spec.invmap af sf vf* ⟪*σ*⟫ = ⨆(*spec.singleton* ‘ {*σ'*. ⟪*trace.map af sf vf σ'*⟫ ≤ ⟪*σ*⟫})
by (*simp add*: *spec.invmap-def*)

lemma *id*:
  shows *spec.invmap id id id P* = *P*
    and *spec.invmap* (λ*x*. *x*) (λ*x*. *x*) (λ*x*. *x*) *P* = *P*
unfolding *id-def*[*symmetric*] **by** (*metis spec.map.id*(*1*) *spec.map-invmap.lower-upper-lower*(*2*))+

lemma *comp*:
  shows *spec.invmap af sf vf* (*spec.invmap ag sg vg P*) = *spec.invmap* (λ*x*. *ag* (*af x*)) (λ*s*. *sg* (*sf s*)) (λ*v*. *vg* (*vf v*)) *P* (**is** *?lhs P* = *?rhs P*)
    and *spec.invmap af sf vf* ∘ *spec.invmap ag sg vg* = *spec.invmap* (*ag* ∘ *af*) (*sg* ∘ *sf*) (*vg* ∘ *vf*) (**is** *?thesis1*)
proof −
  show *?lhs P* = *?rhs P* **for** *P*
    by (*auto intro*: *spec.singleton.antisym spec.singleton-le-extI simp*: *spec.singleton.le-conv*)
  then show *?thesis1*
    by (*simp add*: *fun-eq-iff comp-def*)
qed

lemmas *invmap* = *spec.invmap.comp*

lemma *invmap-inf-distr-le*:
  fixes *af* :: *'a* ⇒ *'b*
  fixes *sf* :: *'s* ⇒ *'t*
  fixes *vf* :: *'v* ⇒ *'w*
  shows *spec.invmap af sf vf P* ⊓ *Q* ≤ *spec.invmap af sf vf* (*P* ⊓ *spec.map af sf vf Q*)
    and *Q* ⊓ *spec.invmap af sf vf P* ≤ *spec.invmap af sf vf* (*spec.map af sf vf Q* ⊓ *P*)
by (*metis order.refl inf-mono spec.map-invmap.upper-inf spec.map-invmap.upper-lower-expansive*)+

setup ‹*Sign.parent-path*›

setup ‹*Sign.mandatory-path amap*›

```
lemma invmap-le: — af = id in spec.invmap
  shows spec.amap af (spec.invmap id sf vf P) ≤ spec.invmap id sf vf (spec.amap af P)
proof −
  have spec.invmap id sf vf P ≤ spec.invmap af sf vf (spec.amap af P) (is ?lhs ≤ ?rhs)
  proof(rule spec.singleton-le-extI)
    show ⟪σ⟫ ≤ ?rhs if ⟪σ⟫ ≤ ?lhs for σ
      using that by (simp add: spec.singleton.le-conv exI[where x=trace.map id sf vf σ] flip: id-def)
  qed
  then show ?thesis
    by (simp add: spec.map-invmap.galois spec.invmap.comp flip: id-def)
qed

lemma surj-invmap: — af = id in spec.invmap
  fixes P :: ('a, 't, 'w) spec
  fixes af :: 'a ⇒ 'b
  fixes sf :: 's ⇒ 't
  fixes vf :: 'v ⇒ 'w
  assumes surj af
  shows spec.amap af (spec.invmap id sf vf P) = spec.invmap id sf vf (spec.amap af P) (is ?lhs = ?rhs)
proof(rule antisym[OF spec.amap.invmap-le spec.singleton-le-extI])
  have 1: ∃σ₃. σ₂ = trace.map af id id σ₃ ∧ σ₁ ≃S σ₃
    if trace.map af id id σ₁ ≃S σ₂
   for σ₁ :: ('a, 't, 'w) trace.t and σ₂
  proof −
    have **: ∃ys'. ys = map (map-prod af id) ys' ∧ trace.natural' s xs = trace.natural' s ys'
      if trace.natural' s (map (map-prod af id) xs) = trace.natural' s ys
     for s :: 't and xs ys
      using that
    proof(induct ys arbitrary: s xs)
      case Nil then show ?case
        by (fastforce simp: trace.natural'.eq-Nil-conv)
    next
      case (Cons y ys s xs) show ?case
      proof(cases snd y = s)
        case True with Cons.prems show ?thesis
          by (fastforce dest: Cons.hyps
                        simp: iffD1[OF surj-iff ‹surj af›]
                   simp flip: id-def
                       intro: exI[where x=map-prod (inv af) id y # ys' for ys'])
      next
        case False with Cons.prems show ?thesis
          by (force dest!: Cons.hyps
                      simp: trace.natural'.eq-Cons-conv trace.natural'.idle-prefix
                            map-eq-append-conv snd-image-map-prod
                 simp flip: id-def
                     intro: exI[where x=(a, s) # xs for a s xs])
      qed
    qed
    from that show ?thesis
      by (cases σ₂) (clarsimp simp: ** trace.natural-def trace.split-Ex)
  qed
  have 2: ∃zs. xs = map (map-prod af id) zs ∧ map (map-prod id sf) zs = ys
    if xs-ys: map (map-prod id sf) xs = map (map-prod af id) ys
   for xs ys
  proof −
    have ∃zs. xs' = map (map-prod af id) zs ∧ map (map-prod id sf) zs = ys'
      if length xs' = length ys'
     and prefix xs' xs
```

```
    and prefix ys' ys
    and map (map-prod id sf) xs = map (map-prod af id) ys for xs' ys'
     using that
   proof(induct xs' ys' rule: rev-induct2)
     case (snoc x xs y ys) then show ?case
       by (cases x; cases y)
          (force simp: prefix-def simp flip: id-def intro: exI[where x=zs @ [(fst y, snd x)] for zs])
   qed simp
   from this[OF map-eq-imp-length-eq[OF xs-ys] prefix-order.refl prefix-order.refl xs-ys]
   show ?thesis .
 qed
 fix σ
 assume ⟪σ⟫ ≤ ?rhs
 then obtain σP where σP: ⟪σP⟫ ≤ P ⟪trace.map id sf vf σ⟫ ≤ ⟪trace.map af id id σP⟫
   by (clarsimp simp: spec.singleton.le-conv)
 then obtain i σP' where σP': trace.map id sf vf σ = trace.map af id id σP' trace.take i σP ≃S σP'
   by (fastforce simp: spec.singleton-le-conv trace.less-eq-take-def trace.take.map
                 dest: 1[OF sym]
                elim!: trace.take.naturalE)
 then obtain zs where zs: trace.rest σ = map (map-prod af id) zs map (map-prod id sf) zs = trace.rest σP'
   by (cases σ, cases σP') (clarsimp dest!: 2)
 from ‹⟪σP⟫ ≤ P› σP'(1) σP'(2)[symmetric] zs show ⟪σ⟫ ≤ ?lhs
   by (cases σ, cases σP')
      (clarsimp intro!: exI[where x=trace.T (trace.init σ) zs (trace.term σ)]
                elim!: order.trans[rotated]
                 simp: spec.singleton.le-conv spec.singleton-le-conv trace.natural.mono)
qed

setup ‹Sign.parent-path›

setup ‹Sign.parent-path›
```

8.5 The idle process

As observed by Abadi and Plotkin (1991), many laws require the processes involved to accept all initial states (see, for instance, §8.8). We call the minimal such process *spec.idle*. It is also the lower bound on specification by transition relation (§8.10).

```
setup ‹Sign.mandatory-path spec›

definition idle :: ('a, 's, 'v) spec where
  idle = (⨆ s. ⟪s, [], None⟫)

named-theorems idle-le ‹ rules for ‹spec.idle ≤ const ...› ›

setup ‹Sign.mandatory-path singleton›

lemma idle-le-conv[spec.singleton.le-conv]:
  shows ⟪σ⟫ ≤ spec.idle ⟷ trace.steps σ = {} ∧ trace.term σ = None
by (auto simp: spec.idle-def spec.singleton-le-conv trace.natural.simps trace.natural'.eq-Nil-conv
               trace.less-eq-None)

setup ‹Sign.parent-path›

setup ‹Sign.mandatory-path idle›

lemma minimal-le:
  shows ⟪s, [], None⟫ ≤ spec.idle
by (simp add: spec.singleton.idle-le-conv)
```

lemma *map-le*[*spec.idle-le*]:
  assumes *spec.idle* ≤ *P*
  assumes *surj sf*
  shows *spec.idle* ≤ *spec.map af sf vf P*
by (*strengthen ord-to-strengthen*(*2*)[*OF assms*(*1*)])
  (*use* ‹*surj sf*› **in** ‹*auto simp*: *spec.idle-def spec.map.Sup spec.map.singleton image-image*
    spec.singleton-le-conv trace.natural.simps trace.less-eq-None›)

lemma *invmap-le*:
  assumes *spec.idle* ≤ *P*
  shows *spec.idle* ≤ *spec.invmap af sf vf P*
by (*strengthen ord-to-strengthen*(*2*)[*OF assms*])
  (*auto simp*: *spec.idle-def spec.map.Sup spec.map.singleton simp flip*: *spec.map-invmap.galois*)

setup ‹*Sign.parent-path*›

setup ‹*Sign.mandatory-path map-invmap*›

lemma *cl-alt-def*:
  shows *spec.map-invmap.cl - - - af sf vf P*
    = ⨆{⟪*σ*⟫ |*σ σ'*. ⟪*σ'*⟫ ≤ *P* ∧ ⟪*trace.map af sf vf σ*⟫ ≤ ⟪*trace.map af sf vf σ'*⟫} (**is** *?lhs* = *?rhs*)
proof(*rule antisym*)
  show *?lhs* ≤ *?rhs*
    by (*rule spec.singleton.exhaust*[*of P*])
      (*fastforce simp*: *spec.map-invmap.cl-def*
        spec.map.Sup spec.map.singleton spec.invmap.Sup spec.invmap.singleton
        intro: *spec.singleton.mono*)
  show *?rhs* ≤ *?lhs*
    by (*clarsimp simp*: *spec.map-invmap.cl-def simp flip*: *spec.map.singleton*)
      (*simp add*: *order.trans*[*OF - spec.map.mono*] *flip*: *spec.map-invmap.galois*)
qed

lemma *cl-le-conv*[*spec.singleton.le-conv*]:
  shows ⟪*σ*⟫ ≤ *spec.map-invmap.cl - - - af sf vf P* ⟷ ⟪*trace.map af sf vf σ*⟫ ≤ *spec.map af sf vf P*
by (*simp add*: *spec.map-invmap.cl-def spec.singleton.invmap-le-conv*)

setup ‹*Sign.parent-path*›

setup ‹*Sign.parent-path*›

8.6 Actions

Our primitive actions are arbitrary relations on the state, labelled by the agent performing the state transition and a value to return.

For refinement purposes we need *idle* ≤ *action a F*; see §12.1.1.

setup ‹*Sign.mandatory-path spec*›

definition *action* :: ('*v* × '*a* × '*s* × '*s*) *set* ⇒ ('*a*, '*s*, '*v*) *spec* **where**
  action F = (⨆(*v*, *a*, *s*, *s'*)∈*F*. ⟪*s*, [(*a*, *s'*)], *Some v*⟫) ⊔ *spec.idle*

definition *guard* :: ('*s* ⇒ *bool*) ⇒ ('*a*, '*s*, *unit*) *spec* **where**
  guard g = *spec.action* ({()} × *UNIV* × *Diag g*)

definition *return* :: '*v* ⇒ ('*a*, '*s*, '*v*) *spec* **where**
  return v = *spec.action* ({*v*} × *UNIV* × *Id*)

abbreviation (*input*) *read* :: (*'s* ⇒ *'v option*) ⇒ (*'a*, *'s*, *'v*) *spec* **where**
  read f ≡ *spec.action* {(*v*, *a*, *s*, *s*) |*a s v*. *f s* = *Some v*}

abbreviation (*input*) *write* :: *'a* ⇒ (*'s* ⇒ *'s*) ⇒ (*'a*, *'s*, *unit*) *spec* **where**
  write a f ≡ *spec.action* {((), *a*, *s*, *f s*) |*s*. *True*}

lemma *action-le*[*case-names idle step*]:
  assumes *spec.idle* ≤ *P*
  assumes ⋀*v a s s'*. (*v*, *a*, *s*, *s'*) ∈ *F* ⟹ ⟪*s*, [(*a*, *s'*)], *Some v*⟫ ≤ *P*
  shows *spec.action F* ≤ *P*
by (*simp add*: *assms spec.action-def split-def*)

setup ‹*Sign.mandatory-path idle*›

lemma *action-le*[*spec.idle-le*]:
  shows *spec.idle* ≤ *spec.action F*
by (*simp add*: *spec.action-def*)

lemma *guard-le*[*spec.idle-le*]:
  shows *spec.idle* ≤ *spec.guard g*
by (*simp add*: *spec.guard-def spec.idle-le*)

lemma *return-le*[*spec.idle-le*]:
  shows *spec.idle* ≤ *spec.return v*
by (*simp add*: *spec.return-def spec.idle-le*)

setup ‹*Sign.parent-path*›

setup ‹*Sign.mandatory-path map*›

lemma *action-le*:
  fixes *F* :: (*'v* × *'a* × *'s* × *'s*) *set*
  shows *spec.map af sf vf* (*spec.action F*) ≤ *spec.action* (*map-prod vf* (*map-prod af* (*map-prod sf sf*)) ‘ *F*)
by (*force simp*: *spec.action-def spec.idle-def spec.map.Sup spec.map.sup spec.map.singleton SUP-le-iff*)

lemma *action*:
  fixes *F* :: (*'v* × *'a* × *'s* × *'s*) *set*
  shows *spec.map af sf vf* (*spec.action F*) ⊔ *spec.idle*
      = *spec.action* (*map-prod vf* (*map-prod af* (*map-prod sf sf*)) ‘ *F*) (**is** *?lhs* = *?rhs*)
proof(*rule antisym*)
  show *?lhs* ≤ *?rhs*
    by (*simp add*: *spec.idle-le spec.map.action-le*)
  show *?rhs* ≤ *?lhs*
    by (*force simp*: *spec.action-def spec.idle-def spec.map.sup spec.map.singleton spec.map.Sup*)
qed

lemma *surj-sf-action*:
  assumes *surj sf*
  shows *spec.map af sf vf* (*spec.action F*) = *spec.action* (*map-prod vf* (*map-prod af* (*map-prod sf sf*)) ‘ *F*)
by (*simp add*: *assms sup.absorb1 spec.idle-le flip*: *spec.map.action*)

setup ‹*Sign.parent-path*›

setup ‹*Sign.mandatory-path action*›

lemma *empty*:
  shows *spec.action* {} = *spec.idle*
by (*simp add*: *spec.action-def*)

lemma *idleI*:
  assumes *snd* ‘ *set xs* ⊆ {*s*}
  shows ⟪*s*, *xs*, *None*⟫ ≤ *spec.action F*
using *assms* **by** (*simp add*: *spec.action-def spec.singleton.le-conv*)

lemma *stepI*:
  assumes (*v*, *a*, *s*, *s′*) ∈ *F*
  assumes ∀ *v″*. *w* = *Some v″* ⟶ *v″* = *v*
  shows ⟪*s*, [(*a*, *s′*)], *w*⟫ ≤ *spec.action F*
using *assms* **by** (*cases w*; *force simp*: *spec.action-def spec.singleton.mono trace.less-eq-None*)

lemma *stutterI*:
  assumes (*v*, *a*, *s*, *s*) ∈ *F*
  shows ⟪*s*, [], *Some v*⟫ ≤ *spec.action F*
by (*fastforce simp*: *spec.singleton-le-conv trace.natural.simps*
        intro: *order.trans*[*OF - spec.action.stepI*[*OF assms*]])

lemma *stutter-stepI*:
  assumes (*v*, *a*, *s*, *s*) ∈ *F*
  shows ⟪*s*, [(*b*, *s*)], *Some v*⟫ ≤ *spec.action F*
by (*fastforce simp*: *spec.singleton-le-conv trace.natural.simps*
        intro: *order.trans*[*OF - spec.action.stepI*[*OF assms*]])

lemma *stutter-stepsI*:
  assumes (*v*, *a*, *s*, *s*) ∈ *F*
  assumes *snd* ‘ *set xs* ⊆ {*s*}
  shows ⟪*s*, *xs*, *Some v*⟫ ≤ *spec.action F*
by (*simp add*: *assms trace.natural′.eq-Nil-conv order.trans*[*OF - spec.action.stutterI*[*OF assms*(*1*)]])

lemma *monotone*:
  shows *mono spec.action*
by (*force simp*: *spec.action-def intro*: *monoI*)

lemmas *strengthen*[*strg*] = *st-monotone*[*OF spec.action.monotone*]
lemmas *mono* = *monotoneD*[*OF spec.action.monotone*]
lemmas *mono2mono*[*cont-intro*, *partial-function-mono*]
  = *monotone2monotone*[*OF spec.action.monotone*, *simplified*]

lemma *Sup*:
  shows *spec.action* (⋃ *X*) = (⨆ *F*∈*X*. *spec.action F*) ⊔ *spec.idle*
by (*force simp*: *spec.eq-iff spec.action-def*)

lemma
  shows *SUP*: *spec.action* (⋃ *x*∈*X*. *F x*) = (⨆ *x*∈*X*. *spec.action* (*F x*)) ⊔ *spec.idle*
    and *SUP-not-empty*: *X* ≠ {} ⟹ *spec.action* (⋃ *x*∈*X*. *F x*) = (⨆ *x*∈*X*. *spec.action* (*F x*))
by (*auto simp*: *spec.action.Sup image-image sup.absorb1 SUPI spec.idle-le simp flip*: *ex-in-conv*)

lemma *sup*:
  shows *spec.action* (*F* ∪ *G*) = *spec.action F* ⊔ *spec.action G*
using *spec.action.Sup*[**where** *X*={*F*, *G*}] **by** (*simp add*: *sup-absorb1 le-supI1 spec.idle-le*)

lemma *Inf-le*:
  shows *spec.action* (⋂ *Fs*) ≤ ⨅ (*spec.action* ‘ *Fs*)
by (*simp add*: *spec.action-def ac-simps SUP-le-iff SUP-upper le-INF-iff le-supI2*)

lemma *inf-le*:

shows *spec.action* $(F \cap G) \leq$ *spec.action* $F \sqcap$ *spec.action* G
using *spec.action.Inf-le*[**where** *Fs*=$\{F, G\}$] **by** *simp*

lemma *stutter-agents-le*:
assumes $[\![A \neq \{\}; r \neq \{\}]\!] \Longrightarrow B \neq \{\}$
assumes $r \subseteq Id$
shows *spec.action* $(\{v\} \times A \times r) \leq$ *spec.action* $(\{v\} \times B \times r)$
using *assms*
by (*subst spec.action-def*) (*fastforce simp*: *spec.idle-le intro*!: *spec.action.stutter-stepI*)

lemma *read-agents*:
assumes $A \neq \{\}$
assumes $B \neq \{\}$
assumes $r \subseteq Id$
shows *spec.action* $(\{v\} \times A \times r) =$ *spec.action* $(\{v\} \times B \times r)$
by (*rule antisym*[*OF spec.action.stutter-agents-le spec.action.stutter-agents-le*]; *rule assms*)

lemma *invmap-le*: — A typical refinement
fixes *af* :: $'a \Rightarrow 'b$
fixes *sf* :: $'s \Rightarrow 't$
fixes *vf* :: $'v \Rightarrow 'w$
shows *spec.action* (*map-prod vf* (*map-prod af* (*map-prod sf sf*)) $-`$ *F*) $\leq$ *spec.invmap af sf vf* (*spec.action F*)
by (*meson order.trans image-vimage-subset spec.action.mono spec.map.action-le spec.map-invmap.galois*)

setup ‹*Sign.parent-path*›

setup ‹*Sign.mandatory-path singleton*›

lemma *action-le-conv*:
shows $\langle\!|\sigma|\!\rangle \leq$ *spec.action F*
$\longleftrightarrow$ (*trace.steps* $\sigma = \{\} \wedge$ *case-option True* $(\lambda v.\ \exists a.\ (v, a,$ *trace.init* $\sigma,$ *trace.init* $\sigma) \in F)$ (*trace.term* σ))
$\vee\ (\exists x{\in}F.$ *trace.steps* $\sigma = \{$*snd* $x\} \wedge$ *case-option True* $((=)$ (*fst* $x))$ (*trace.term* σ)) (**is** *?lhs* $\longleftrightarrow$ *?rhs*)
proof(*rule iffI*)
show *?lhs* $\Longrightarrow$ *?rhs*
unfolding *spec.action-def spec.singleton.sup-le-conv*
proof(*induct rule*: *disjE*[*consumes 1, case-names step idle*])
case *step*
then obtain *v a s s'* **where** $*$: $\natural\sigma \leq \natural$(*trace.T s* $[(a, s')]$ (*Some v*)) **and** *F*: $(v, a, s, s') \in F$
by (*clarsimp simp*: *spec.singleton-le-conv*)
from $*$ **show** *?case*
proof(*induct rule*: *trace.less-eqE*)
case *prefix* **with** *F* **show** *?case*
by (*clarsimp simp add*: *trace.natural'.eq-Nil-conv prefix-Cons split*: *if-splits*)
(*force simp*: *trace.steps'-alt-def*)
next
case (*maximal v*) **with** *F* **show** *?case*
by (*clarsimp simp*: *trace.natural.trace-conv trace.natural'.eq-Nil-conv*
trace.natural'.eq-Cons-conv trace.steps'.append split: *if-splits*;
force)
qed
qed (*simp add*: *spec.singleton.le-conv*)
show *?rhs* $\Longrightarrow$ *?lhs*
by (*cases* σ)
(*auto simp*: *trace.steps'.step-conv*
intro: *spec.action.idleI spec.action.stutter-stepsI spec.action.stepI*
elim!: *order.trans*[*OF eq-refl*[*OF spec.singleton.Cons*]]
split: *option.split-asm*)
qed

lemma *action-Some-leE*:
assumes $\langle\!|\sigma|\!\rangle \leq$ *spec.action F*
assumes *trace.term* σ = *Some v*
obtains *x*
where $x \in F$
and *trace.init* σ = *fst* (*snd* (*snd x*))
and *trace.final* σ = *snd* (*snd* (*snd x*))
and *trace.steps* $\sigma \subseteq \{$*snd x*$\}$
and *v* = *fst x*
using *assms* **by** (*auto simp*: *spec.singleton.action-le-conv trace.steps'.step-conv trace.steps'.append*)

lemma *action-not-idle-leE*:
assumes $\langle\!|\sigma|\!\rangle \leq$ *spec.action F*
assumes $\natural\sigma \neq$ *trace.T* (*trace.init* σ) [] *None*
obtains *x*
where $x \in F$
and *trace.init* σ = *fst* (*snd* (*snd x*))
and *trace.final* σ = *snd* (*snd* (*snd x*))
and *trace.steps* $\sigma \subseteq \{$*snd x*$\}$
and *case-option True* ((=) (*fst x*)) (*trace.term* σ)
using *assms*
by (*cases* σ)
(*auto 0 0 simp*: *spec.singleton.action-le-conv trace.natural.idle option.case-eq-if*
trace.steps'.step-conv trace.steps'.append)

lemma *action-not-idle-le-splitE*:
assumes $\langle\!|\sigma|\!\rangle \leq$ *spec.action F*
assumes $\natural\sigma \neq$ *trace.T* (*trace.init* σ) [] *None*
obtains (*return*) *v a*
where (*v*, *a*, *trace.init* σ, *trace.init* σ) $\in F$
and *trace.steps* σ = {}
and *trace.term* σ = *Some v*
| (*step*) *v a ys zs*
where (*v*, *a*, *trace.init* σ, *trace.final* σ) $\in F$
and *trace.init* $\sigma \neq$ *trace.final* σ
and *snd* ‘ *set ys* $\subseteq \{$*trace.init* $\sigma\}$
and *snd* ‘ *set zs* $\subseteq \{$*trace.final* $\sigma\}$
and *trace.rest* σ = *ys* @ [(*a*, *trace.final* σ)] @ *zs*
and *case-option True* ((=) *v*) (*trace.term* σ)
using *assms*
by (*cases* σ)
(*auto 0 0 simp*: *spec.singleton.action-le-conv trace.natural.idle option.case-eq-if*
trace.steps'.step-conv trace.steps'.append
cong: *if-cong*)

lemma *guard-le-conv*[*spec.singleton.le-conv*]:
shows $\langle\!|\sigma|\!\rangle \leq$ *spec.guard g* $\longleftrightarrow$ *trace.steps* σ = {} $\wedge$ (*case-option True* $\langle g$ (*trace.init* $\sigma)\rangle$ (*trace.term* σ))
by (*fastforce simp*: *spec.guard-def spec.singleton.action-le-conv trace.steps'.step-conv*
split: *option.split*)

lemma *return-le-conv*[*spec.singleton.le-conv*]:
shows $\langle\!|\sigma|\!\rangle \leq$ *spec.return v*
$\longleftrightarrow$ *trace.steps* σ = {} $\wedge$ (*case-option True* ((=) *v*) (*trace.term* σ))
by (*fastforce simp*: *spec.return-def spec.singleton.action-le-conv trace.steps'.step-conv*
split: *option.split*)

setup ‹*Sign.parent-path*›

setup ‹*Sign.mandatory-path action*›

lemma *mono-stronger*:
assumes $\bigwedge v\ a\ s\ s'.\ [\![(v,\ a,\ s,\ s') \in F;\ s \neq s']\!] \Longrightarrow (v,\ a,\ s,\ s') \in F'$
assumes $\bigwedge v\ a\ s.\ (v,\ a,\ s,\ s) \in F \Longrightarrow \exists\, a'.\ (v,\ a',\ s,\ s) \in F'$
shows *spec.action F* $\leq$ *spec.action F'*
proof (*induct rule*: *spec.action-le*[*OF spec.idle.action-le*, *case-names step*])
case (*step v a s s'*) **then show** *?case*
by (*cases s = s'*)
(*auto dest*: *assms intro*: *spec.action.stutterI spec.action.stepI*)
qed

lemma *cong*:
assumes $\bigwedge v\ a\ s\ s'.\ s \neq s' \Longrightarrow (v,\ a,\ s,\ s') \in F \longleftrightarrow (v,\ a,\ s,\ s') \in F'$
assumes $\bigwedge v\ a\ s.\ (v,\ a,\ s,\ s) \in F \Longrightarrow \exists\, a'.\ (v,\ a',\ s,\ s) \in F'$
assumes $\bigwedge v\ a\ s.\ (v,\ a,\ s,\ s) \in F' \Longrightarrow \exists\, a'.\ (v,\ a',\ s,\ s) \in F$
shows *spec.action F = spec.action F'*
using *assms* **by** (*blast intro*!: *spec.antisym spec.action.mono-stronger*)

lemma *le-actionD*:
assumes *spec.action F* $\leq$ *spec.action F'*
shows $[\![(v,\ a,\ s,\ s') \in F;\ s \neq s']\!] \Longrightarrow (v,\ a,\ s,\ s') \in F'$
and $(v,\ a,\ s,\ s) \in F \Longrightarrow \exists\, a'.\ (v,\ a',\ s,\ s) \in F'$
proof −
fix *v a s s'*
show $(v,\ a,\ s,\ s') \in F'$ **if** $(v,\ a,\ s,\ s') \in F$ **and** $s \neq s'$
using *iffD1*[*OF spec.singleton-le-ext-conv assms*] *that*
by (*fastforce simp*: *spec.singleton.action-le-conv*
dest: *spec*[**where** *x*=*trace.T s* [(*a*, *s'*)] (*Some v*)])
show $\exists\, a'.\ (v,\ a',\ s,\ s) \in F'$ **if** $(v,\ a,\ s,\ s) \in F$
using *iffD1*[*OF spec.singleton-le-ext-conv assms*] *that*
by (*fastforce simp*: *spec.singleton.action-le-conv*
dest: *spec*[**where** *x*=*trace.T s* [] (*Some v*)])
qed

lemma *eq-action-conv*:
shows *spec.action F = spec.action F'*
$\longleftrightarrow (\forall\, v\ a\ s\ s'.\ s \neq s' \longrightarrow (v,\ a,\ s,\ s') \in F \longleftrightarrow (v,\ a,\ s,\ s') \in F')$
$\wedge\ (\forall\, v\ a\ s.\ (v,\ a,\ s,\ s) \in F \longrightarrow (\exists\, a'.\ (v,\ a',\ s,\ s) \in F'))$
$\wedge\ (\forall\, v\ a\ s.\ (v,\ a,\ s,\ s) \in F' \longrightarrow (\exists\, a'.\ (v,\ a',\ s,\ s) \in F))$
by (*rule iffI*, *metis order.refl spec.action.le-actionD*, *blast intro*!: *spec.action.cong*)

setup ‹*Sign.parent-path*›

lemma *return-alt-def*:
assumes $A \neq \{\}$
shows *spec.return v = spec.action* $(\{v\} \times A \times Id)$
unfolding *spec.return-def* **using** *assms* **by** (*blast intro*: *spec.action.cong*)

setup ‹*Sign.mandatory-path return*›

lemma *cong*:
assumes $\bigwedge v\ a\ s\ s'.\ (v,\ a,\ s,\ s') \in F \Longrightarrow s' = s$
assumes $\bigwedge v\ s.\ v \in fst\ `\ F \Longrightarrow \exists\, a.\ (v,\ a,\ s,\ s) \in F$
shows *spec.action F* $= \bigsqcup$ (*spec.return* ‘ *fst* ‘ *F*) $\sqcup$ *spec.idle*
by (*simp add*: *spec.return-def image-image flip*: *spec.action.SUP*)
(*rule spec.action.cong*; *auto intro*: *rev-bexI dest*: *assms*(*1*) *intro*: *assms*(*2*))

lemma *action-le*:
  assumes $Id \subseteq snd\ `\ snd\ `\ F$
  shows $spec.return\ () \leq spec.action\ F$
unfolding *spec.return-def*
proof(*induct rule*: *spec.action-le*)
  case (*step v a s s'*) **with** *subsetD*[*OF assms*, **where** *c*=(*s*, *s*)] **show** *?case*
    by (*force intro*: *spec.action.stutterI*)
qed (*simp add*: *spec.idle-le*)

setup ‹*Sign.parent-path*›

setup ‹*Sign.mandatory-path guard*›

lemma *alt-def*:
  assumes $A \neq \{\}$
  shows $spec.guard\ g = spec.action\ (\{()\} \times A \times Diag\ g)$
unfolding *spec.guard-def* **using** *assms* **by** (*fastforce simp*: *intro*: *spec.action.cong*)

lemma *bot*:
  shows $spec.guard\ \bot = spec.idle$
    and $spec.guard\ \langle False\rangle = spec.idle$
by (*simp-all add*: *spec.guard-def spec.action.empty*)

lemma *top*:
  shows $spec.guard\ \top = spec.return\ ()$
    and $spec.guard\ \langle True\rangle = spec.return\ ()$
by (*simp-all add*: *spec.guard-def spec.return-def flip*: *Id-def*)

lemma *monotone*:
  shows *mono spec.guard*
proof(*rule monotoneI*)
  show $spec.guard\ g \leq spec.guard\ g'$ **if** $g \leq g'$ **for** $g\ g' :: 's\ pred$
    unfolding *spec.guard-def* **by** (*strengthen ord-to-strengthen*(*1*)[*OF* ‹$g \leq g'$›]) *simp*
qed

lemmas *strengthen*[*strg*] = *st-monotone*[*OF spec.guard.monotone*]
lemmas *mono* = *monotoneD*[*OF spec.guard.monotone*]
lemmas *mono2mono*[*cont-intro*, *partial-function-mono*] = *monotone2monotone*[*OF spec.guard.monotone*, *simplified*]

lemma *Sup*:
  shows $spec.guard\ (\bigsqcup X) = \bigsqcup (spec.guard\ `\ X) \sqcup spec.idle$
by (*auto simp*: *spec.guard-def Diag-Sup*
    simp flip: *spec.action.Sup*[**where** X=$(\lambda x.\ \{()\} \times UNIV \times Diag\ x)\ `\ X$, *simplified image-image*]
      intro: *arg-cong*[**where** *f*=*spec.action*])

lemma *sup*:
  shows $spec.guard\ (g \sqcup h) = spec.guard\ g \sqcup spec.guard\ h$
by (*simp add*: *spec.guard.Sup*[**where** X=$\{g, h\}$ **for** *g h*, *simplified*]
          ac-simps sup-absorb2 le-supI2 spec.idle-le)

lemma *return-le*:
  shows $spec.guard\ g \leq spec.return\ ()$
by (*simp add*: *spec.guard-def spec.return-def Sigma-mono spec.action.mono*)

lemma *guard-less*: — Non-triviality
  assumes $g < g'$

shows *spec.guard g* < *spec.guard g′*
proof(*rule le-neq-trans*)
show *spec.guard g* ≤ *spec.guard g′*
by (*strengthen ord-to-strengthen*(*1*)[*OF order-less-imp-le*[*OF assms*]]) *simp*
from *assms* **obtain** *s* **where** *g′ s* ¬*g s* **by** (*metis leD predicate1I*)
from ‹¬*g s*› **have** ¬⟪*s*, [], *Some* ()⟫ ≤ *spec.guard g*
by (*clarsimp simp*: *spec.guard-def spec.action-def*
spec.singleton-le-conv spec.singleton.le-conv trace.natural.simps)
moreover
from ‹*g′ s*› **have** ⟪*s*, [], *Some* ()⟫ ≤ *spec.guard g′*
by (*simp add*: *spec.guard-def spec.action.stutterI*)
ultimately show *spec.guard g* ≠ *spec.guard g′* **by** *metis*
qed

lemma *cong*:
assumes ⋀*v a s s′*. (*v*, *a*, *s*, *s′*) ∈ *F* ⟹ *s′* = *s*
shows *spec.action F* = *spec.guard* (λ*s*. *s* ∈ *fst* ‘ *snd* ‘ *snd* ‘ *F*) (**is** *?lhs* = *?rhs*)
proof(*rule antisym*)
show *?lhs* ≤ *?rhs*
by (*force simp*: *spec.guard-def intro*: *spec.action.mono dest*: *assms*)
show *?rhs* ≤ *?lhs*
unfolding *spec.guard-def*
by (*rule spec.action-le*;
clarsimp simp: *spec.idle-le*; *blast intro*: *spec.action.stutterI dest*: *assms*)
qed

lemma *action-le*:
assumes *Diag g* ⊆ *snd* ‘ *snd* ‘ *F*
shows *spec.guard g* ≤ *spec.action F*
unfolding *spec.guard-def*
proof(*induct rule*: *spec.action-le*)
case (*step v a s s′*) **with** *subsetD*[*OF assms*, **where** *c*=(*s*, *s*)] **show** *?case*
by (*force intro*: *spec.action.stutterI*)
qed (*simp add*: *spec.idle-le*)

setup ‹*Sign.parent-path*›

setup ‹*Sign.parent-path*›

8.7 Operations on return values

For various purposes, including defining a history-respecting sequential composition (bind, see §8.8), we use a Galois pair of operations that saturate or eradicate return values.

setup ‹*Sign.mandatory-path spec*›

setup ‹*Sign.mandatory-path term*›

definition *none* :: (′*a*, ′*s*, ′*v*) *spec* ⇒ (′*a*, ′*s*, ′*w*) *spec* **where**
none P = ⨆{⟪*s*, *xs*, *None*⟫ |*s xs v*. ⟪*s*, *xs*, *v*⟫ ≤ *P*}

definition *all* :: (′*a*, ′*s*, ′*v*) *spec* ⇒ (′*a*, ′*s*, ′*w*) *spec* **where**
all P = ⨆{⟪*s*, *xs*, *v*⟫ |*s xs v*. ⟪*s*, *xs*, *None*⟫ ≤ *P*}

setup ‹*Sign.parent-path*›

interpretation *term*: *galois.complete-lattice-distributive-class spec.term.none spec.term.all*
proof *standard*
show *spec.term.none P* ≤ *Q* ⟷ *P* ≤ *spec.term.all Q* (**is** *?lhs* ⟷ *?rhs*)

for $P :: ('a, 'b, 'c)$ *spec*
and $Q :: ('a, 'b, 'f)$ *spec*
proof(*rule iffI*)
show *?lhs* $\Longrightarrow$ *?rhs*
by (*fastforce simp*: *spec.term.none-def spec.term.all-def trace.split-all*
intro: *spec.singleton-le-extI*)
show *?rhs* $\Longrightarrow$ *?lhs*
by (*fastforce simp*: *spec.term.none-def spec.term.all-def*
spec.singleton-le-conv trace.natural-def trace.less-eq-None
elim: *trace.less-eqE order.trans*[*rotated*]
dest: *order.trans*[*of - P*])
qed
show *spec.term.all* ($\bigsqcup$ *X*) $\leq \bigsqcup$ (*spec.term.all* ‘ *X*) **for** *X* :: $('a, 'b, 'f)$ *spec set*
by (*auto 0 5 simp*: *spec.term.all-def*)
qed

setup ‹*Sign.mandatory-path singleton.term*›

lemma *none-le-conv*[*spec.singleton.le-conv*]:
shows $\langle\!|\sigma|\!\rangle \leq$ *spec.term.none P* $\longleftrightarrow$ *trace.term* σ = *None* $\wedge$ $\langle\!|$*trace.init* σ, *trace.rest* σ, *None*$|\!\rangle \leq P$ (**is** *?lhs* $\longleftrightarrow$ *?rhs*)
proof(*rule iffI*)
show *?lhs* $\Longrightarrow$ *?rhs*
by (*fastforce simp*: *spec.term.none-def trace.natural-def spec.singleton-le-conv trace.less-eq-None*
intro: *order.trans*[*rotated*])
show *?rhs* $\Longrightarrow$ *?lhs*
by (*cases* σ) (*fastforce simp*: *spec.term.none-def*)
qed

lemma *all-le-conv*[*spec.singleton.le-conv*]:
shows $\langle\!|\sigma|\!\rangle \leq$ *spec.term.all P* $\longleftrightarrow$ ($\exists$ *w*. $\langle\!|$*trace.init* σ, *trace.rest* σ, *w*$|\!\rangle \leq P$) (**is** *?lhs* $\longleftrightarrow$ *?rhs*)
proof(*rule iffI*)
show *?lhs* $\Longrightarrow$ *?rhs*
by (*cases* σ) (*fastforce simp*: *spec.term.none-def simp flip*: *spec.term.galois*)
show *?rhs* $\Longrightarrow$ *?lhs*
by (*cases* σ) (*fastforce simp*: *spec.term.all-def intro*: *order.trans*[*rotated*])
qed

setup ‹*Sign.parent-path*›

setup ‹*Sign.mandatory-path term*›

setup ‹*Sign.mandatory-path none*›

lemma *singleton*:
shows *spec.term.none* $\langle\!|\sigma|\!\rangle$ = $\langle\!|$*trace.init* σ, *trace.rest* σ, *None*$|\!\rangle$
by (*force simp*: *spec.eq-iff spec.term.galois spec.singleton.le-conv spec.singleton.mono trace.less-eq-None*)

lemmas *bot*[*simp*] = *spec.term.lower-bot*

lemmas *monotone* = *spec.term.monotone-lower*
lemmas *mono* = *monotoneD*[*OF spec.term.none.monotone*]

lemmas *Sup* = *spec.term.lower-Sup*
lemmas *sup* = *spec.term.lower-sup*

lemmas *Inf-le* = *spec.term.lower-Inf-le*

lemma *Inf-not-empty*:
assumes $X \neq \{\}$
shows *spec.term.none* $(\bigsqcap X) = (\bigsqcap x{\in}X.$ *spec.term.none* $x)$
by (*rule antisym*[*OF spec.term.lower-Inf-le*])
(*use assms* **in** ‹*auto intro*: *spec.singleton-le-extI simp*: *spec.singleton.term.none-le-conv le-Inf-iff*›)

lemma *inf*:
shows *spec.term.none* $(P \sqcap Q) =$ *spec.term.none* $P \sqcap$ *spec.term.none* Q
and *spec.term.none* $(Q \sqcap P) =$ *spec.term.none* $Q \sqcap$ *spec.term.none* P
using *spec.term.none.Inf-not-empty*[**where** $X{=}\{P, Q\}$] **by** (*simp-all add*: *ac-simps*)

lemma *inf-unit*:
fixes P Q :: (-, -, *unit*) *spec*
shows *spec.term.none* $(P \sqcap Q) =$ *spec.term.none* $P \sqcap Q$ (**is** *?thesis1 P Q*)
and *spec.term.none* $(P \sqcap Q) = P \sqcap$ *spec.term.none* Q (**is** *?thesis2*)
proof −
show $*$: *?thesis1 P Q* **for** P Q
by (*rule spec.singleton.antisym*; *metis le-inf-iff spec.singleton.term.none-le-conv trace.t.collapse*)
from $*$[**where** $P{=}Q$ **and** $Q{=}P$] **show** *?thesis2*
by (*simp add*: *ac-simps*)
qed

lemma *idempotent*[*simp*]:
shows *spec.term.none* (*spec.term.none* P) = *spec.term.none* P
by (*rule spec.singleton.exhaust*[*of P*])
(*simp add*: *spec.term.none.Sup spec.term.none.singleton image-image*)

lemma *contractive*[*iff*]:
shows *spec.term.none* $P \leq P$
by (*rule spec.singleton-le-extI*) (*simp add*: *spec.singleton.le-conv trace.split-all*)

lemma *map-gen*:
fixes *vf* :: $'v \Rightarrow 'w$
fixes *vf'* :: $'a \Rightarrow 'b$ — arbitrary type
shows *spec.term.none* (*spec.map af sf vf P*) = *spec.map af sf vf'* (*spec.term.none P*) (**is** *?lhs* = *?rhs*)
by (*fastforce simp*: *spec.map-def spec.eq-iff image-image trace.split-all trace.split-Ex*
spec.term.none.Sup spec.term.none.singleton spec.singleton.term.none-le-conv
elim: *order.trans*[*rotated*])

lemmas *map* = *spec.term.none.map-gen*[**where** *vf'*=*id*] — *simp*-friendly

lemma *invmap-gen*:
fixes *vf* :: $'v \Rightarrow 'w$
fixes *vf'* :: $'a \Rightarrow 'b$ — arbitrary type
shows *spec.term.none* (*spec.invmap af sf vf P*) = *spec.invmap af sf vf'* (*spec.term.none P*) (**is** *?lhs* = *?rhs*)
proof(*rule antisym*)
show *?lhs* ≤ *?rhs*
by (*simp add*: *spec.map-invmap.lower-upper-contractive spec.term.none.mono*
flip: *spec.map-invmap.galois spec.term.none.map-gen*[**where** *vf*=*vf*])
show *?rhs* ≤ *?lhs*
by (*rule spec.singleton-le-extI*)
(*clarsimp simp*: *spec.singleton.invmap-le-conv spec.singleton.term.none-le-conv*)
qed

lemmas *invmap* = *spec.term.none.invmap-gen*[**where** *vf'*=*id*] — *simp*-friendly

lemma *idle*:
shows *spec.term.none spec.idle* = *spec.idle*

by (*simp add*: *spec.idle-def spec.term.none.Sup spec.term.none.singleton image-image*)

lemma *return*:
  shows *spec.term.none* (*spec.return v*) = *spec.idle*
by (*auto simp*: *spec.eq-iff spec.return-def spec.action-def spec.term.none.idle spec.singleton.idle-le-conv*
    spec.term.none.sup spec.term.none.Sup spec.term.none.singleton)

lemma *guard*:
  shows *spec.term.none* (*spec.guard g*) = *spec.idle*
by (*rule antisym*[*OF spec.term.none.mono*[*OF spec.guard.return-le*, *simplified spec.term.none.return*]
      spec.term.none.mono[*OF spec.idle.guard-le*, *simplified spec.term.none.idle*]])

setup ‹*Sign.parent-path*›

lemma *none-all-le*:
  shows *spec.term.none P* ≤ *spec.term.all P*
using *spec.term.galois* **by** *fastforce*

lemma *none-all*[*simp*]:
  shows *spec.term.none* (*spec.term.all P*) = *spec.term.none P*
by (*metis spec.eq-iff spec.term.lower-upper-contractive*
    spec.term.none.idempotent spec.term.none.mono spec.term.none-all-le)

lemma *all-none*[*simp*]:
  shows *spec.term.all* (*spec.term.none P*) = *spec.term.all P*
by (*metis spec.eq-iff spec.term.galois spec.term.none-all*)

setup ‹*Sign.mandatory-path all*›

lemmas *bot*[*simp*] = *spec.term.upper-bot*

lemmas *top* = *spec.term.upper-top*

lemmas *monotone* = *spec.term.monotone-upper*
lemmas *mono* = *monotoneD*[*OF spec.term.all.monotone*]

lemma *expansive*:
  shows *P* ≤ *spec.term.all P*
using *spec.term.galois* **by** *blast*

lemmas *Sup* = *spec.term.upper-Sup*
lemmas *sup* = *spec.term.upper-sup*

lemmas *Inf* = *spec.term.upper-Inf*
lemmas *inf* = *spec.term.upper-inf*

lemmas *singleton* = *spec.term.all-def*[**where** *P*=⟪*σ*⟫] **for** *σ*

lemma *monomorphic*:
  shows *spec.term.cl -* = *spec.term.all*
unfolding *spec.term.cl-def* **by** *simp*

lemma *closed-conv*:
  assumes *P* ∈ *spec.term.closed -*
  shows *P* = *spec.term.all P*
using *assms spec.term.closed-conv* **by** (*auto simp*: *spec.term.all.monomorphic*)

lemma *closed*[*iff*]:

shows *spec.term.all P* ∈ *spec.term.closed* -
using *spec.term.closed-upper* **by** (*auto simp*: *spec.term.all.monomorphic*)

lemma *idempotent*[*simp*]:
shows *spec.term.all* (*spec.term.all P*) = *spec.term.all P*
by (*metis antisym spec.term.galois spec.term.lower-upper-contractive spec.term.none.idempotent*)

lemma *map*: — *vf* = *id* on the RHS
fixes *vf* :: *'v* ⇒ *'w*
shows *spec.term.all* (*spec.map af sf vf P*) = *spec.map af sf id* (*spec.term.all P*) (**is** *?lhs* = *?rhs*)
proof(*rule antisym*[*OF spec.singleton-le-extI*])
fix *σ*
assume ⟪*σ*⟫ ≤ *?lhs*
then obtain *σ' i w*
where ⟪*σ'*⟫ ≤ *P*
and *trace.T* (*trace.init σ*) (*trace.rest σ*) *w* ≃*S trace.map af sf vf* (*trace.take i σ'*)
using *that* **by** (*fastforce elim*!: *trace.less-eq-takeE trace.take.naturalE*
simp: *trace.take.map spec.singleton.le-conv spec.singleton-le-conv*)
then show ⟪*σ*⟫ ≤ *?rhs*
by (*simp add*: *spec.singleton.le-conv spec.singleton-le-conv*)
(*fastforce intro*: *exI*[**where** *x*=*trace.T* (*trace.init σ'*) (*trace.rest* (*trace.take i σ'*)) (*trace.term σ*)]
exI[**where** *x*=*None*]
elim: *order.trans*[*rotated*]
simp: *trace.natural-def spec.singleton.mono trace.less-eq-None take-is-prefix*)
next
show *?rhs* ≤ *?lhs*
by (*simp add*: *spec.term.none.map flip*: *spec.term.galois*)
(*simp flip*: *spec.term.none.map*[**where** *vf*=*vf*])
qed

lemma *invmap*: — *vf* = *id* on the RHS
fixes *vf* :: *'v* ⇒ *'w*
shows *spec.term.all* (*spec.invmap af sf vf P*) = *spec.invmap af sf id* (*spec.term.all P*) (**is** *?lhs* = *?rhs*)
proof(*rule antisym*)
show *?lhs* ≤ *?rhs*
by (*simp add*: *order.trans*[*OF spec.term.none.contractive spec.map-invmap.lower-upper-contractive*]
flip: *spec.map-invmap.galois spec.term.galois spec.term.all.map*[**where** *vf*=*vf*])
show *?rhs* ≤ *?lhs*
by (*simp add*: *spec.term.none.invmap flip*: *spec.term.galois*)
(*simp flip*: *spec.term.none.invmap-gen*[**where** *vf*=*vf*])
qed

lemma *vmap-unit-absorb*:
shows *spec.vmap* ⟨()⟩ (*spec.term.all P*) = *spec.term.all P* (**is** *?lhs* = *?rhs*)
proof(*rule antisym*[*OF - spec.singleton-le-extI*])
show *?lhs* ≤ *?rhs*
by (*simp add*: *spec.term.none.map spec.map.id flip*: *spec.term.galois*)
show ⟪*σ*⟫ ≤ *?lhs* **if** ⟪*σ*⟫ ≤ *?rhs* **for** *σ*
using *that*
by (*clarsimp simp*: *spec.singleton.le-conv*
intro!: *exI*[**where** *x*=*trace.map id id* ⟨*undefined*⟩ *σ*])
(*metis* (*mono-tags*) *order.refl fun-unit-id trace.t.map-ident*)
qed

lemma *vmap-unit*:
shows *spec.vmap* ⟨()⟩ (*spec.term.all P*) = *spec.term.all* (*spec.vmap* ⟨()⟩ *P*)
by (*simp add*: *spec.map.id spec.term.all.map spec.term.all.vmap-unit-absorb*)

lemma *idle*:
  shows *spec.term.all spec.idle* = (⨆ *v*. *spec.return v*) (**is** *?lhs* = *?rhs*)
proof(*rule antisym*)
  show *?lhs* ≤ *?rhs*
    by (*clarsimp simp*: *spec.term.all-def spec.singleton.le-conv option.case-eq-if*)
  show *?rhs* ≤ *?lhs*
    by (*simp add*: *spec.term.none.Sup spec.term.none.return flip*: *spec.term.galois*)
qed

lemma *action*:
  fixes *F* :: ('*v* × '*a* × '*s* × '*s*) *set*
  shows *spec.term.all* (*spec.action F*) = *spec.action* (*UNIV* × *snd* ‘ *F*) ⊔ (⨆ *v*. *spec.return v*) (**is** *?lhs* = *?rhs*)
proof(*rule antisym*)
  show *?lhs* ≤ *?rhs*
    by (*clarsimp simp*: *spec.term.all-def spec.singleton.le-conv spec.singleton.action-le-conv*
        split: *option.split*)
      meson
  show *?rhs* ≤ *?lhs*
    by (*force simp*: *spec.action-def spec.idle-le spec.term.none.idle spec.term.none.return*
        spec.term.none.Sup spec.term.none.sup spec.term.none.singleton
      simp flip: *spec.term.galois*)
qed

lemma *return*:
  shows *spec.term.all* (*spec.return v*) = (⨆ *v*. *spec.return v*)
by (*auto simp*: *spec.return-def spec.term.all.action*
    simp flip: *spec.action.SUP-not-empty spec.action.sup*
      intro: *arg-cong*[**where** *f*=*spec.action*])

lemma *guard*:
  shows *spec.term.all* (*spec.guard g*) = (⨆ *v*. *spec.return v*)
by (*simp add*: *spec.eq-iff spec.idle.guard-le spec.term.none.Sup spec.term.none.return*
      spec.term.all.mono[*OF spec.guard.return-le*, *unfolded spec.term.all.return*]
      flip: *spec.term.galois*)

setup ‹*Sign.parent-path*›

setup ‹*Sign.parent-path*›

setup ‹*Sign.mandatory-path idle.term*›

lemma *none-le-conv*[*spec.idle-le*]:
  shows *spec.idle* ≤ *spec.term.none P* ⟷ *spec.idle* ≤ *P*
by (*metis spec.term.all.monomorphic spec.term.cl-def spec.term.galois spec.term.none.idle*)

lemma *all-le-conv*[*spec.idle-le*]:
  shows *spec.idle* ≤ *spec.term.all P* ⟷ *spec.idle* ≤ *P*
by (*simp add*: *spec.term.none.idle flip*: *spec.term.galois*)

setup ‹*Sign.parent-path*›

setup ‹*Sign.mandatory-path term.closed*›

lemma *return-unit*:
  shows *spec.return* () ∈ *spec.term.closed* -
by (*rule spec.term.closed-clI*) (*simp add*: *spec.term.all.return spec.term.all.monomorphic*)

lemma *none-inf*:

fixes $P :: ('a, 's, 'v)\ spec$
fixes $Q :: ('a, 's, 'w)\ spec$
assumes $P \in spec.term.closed$ -
shows $P \sqcap spec.term.none\ Q = spec.term.none\ (spec.term.none\ P \sqcap Q)$ (**is** *?lhs* = *?rhs*)
and $spec.term.none\ Q \sqcap P = spec.term.none\ (Q \sqcap spec.term.none\ P)$ (**is** *?thesis1*)
proof −
show *?lhs* = *?rhs*
proof(*rule antisym*[*OF spec.singleton-le-extI*])
show $\langle\!|\sigma|\!\rangle \leq$ *?rhs* **if** $\langle\!|\sigma|\!\rangle \leq$ *?lhs* **for** σ
using *that* **by** (*cases* σ) (*simp add*: *spec.singleton.le-conv*)
show *?rhs* ≤ *?lhs*
by (*auto simp*: *spec.term.galois intro*: *le-infI1 le-infI2 spec.term.none-all-le spec.term.all.expansive*)
qed
then show *?thesis1*
by (*simp add*: *ac-simps*)
qed

lemma *none-inf-monomorphic*:
fixes $P :: ('a, 's, 'v)\ spec$
fixes $Q :: ('a, 's, 'v)\ spec$
assumes $P \in spec.term.closed$ -
shows $P \sqcap spec.term.none\ Q = spec.term.none\ (P \sqcap Q)$ (**is** *?thesis1*)
and $spec.term.none\ Q \sqcap P = spec.term.none\ (Q \sqcap P)$ (**is** *?thesis2*)
by (*simp-all add*: *spec.term.closed.none-inf*[*OF assms*, *simplified*] *spec.term.none.inf*)

lemma *singleton-le-extI*:
assumes $Q \in spec.term.closed$ -
assumes $\bigwedge s\ xs.\ \langle\!|s, xs, None|\!\rangle \leq P \Longrightarrow \langle\!|s, xs, None|\!\rangle \leq Q$
shows $P \leq Q$
by (*subst spec.term.closed-conv*[*OF assms*(*1*)], *rule spec.singleton-le-extI*)
(*auto simp*: *trace.split-all spec.term.none.singleton spec.term.all.monomorphic*
simp flip: *spec.term.galois*
intro: *assms*(*2*)
elim: *order.trans*[*rotated*])

setup ‹*Sign.parent-path*›

setup ‹*Sign.parent-path*›

8.8 Bind

We define monadic *bind* in terms of bi-strict *continue*. The latter supports left and right residuals (see, amongst many others, Hoare and He (1987); Hoare, He, and Sanders (1987b); Pratt (1990)), whereas *bind* encodes the non-retractability of observable actions, i.e., $spec.term.none\ f \leq f \gg\!= g$, which defeats a general right residual. It is tempting to write this in a more direct style (using *case-option*) but the set comprehension syntax is not friendly to strengthen/monotonicity facts.

setup ‹*Sign.mandatory-path spec*›

definition *continue* :: $('a, 's, 'v)\ spec \Rightarrow ('v \Rightarrow ('a, 's, 'w)\ spec) \Rightarrow ('a, 's, 'w)\ spec$ **where**
continue f g =
$\bigsqcup \{\langle\!|trace.init\ \sigma_f,\ trace.rest\ \sigma_f\ @\ trace.rest\ \sigma_g,\ trace.term\ \sigma_g|\!\rangle$
$|\sigma_f\ \sigma_g\ v.\ \langle\!|\sigma_f|\!\rangle \leq f \wedge trace.init\ \sigma_g = trace.final\ \sigma_f \wedge trace.term\ \sigma_f = Some\ v \wedge \langle\!|\sigma_g|\!\rangle \leq g\ v\}$

definition *bind* :: $('a, 's, 'v)\ spec \Rightarrow ('v \Rightarrow ('a, 's, 'w)\ spec) \Rightarrow ('a, 's, 'w)\ spec$ **where**
bind f g = $spec.term.none\ f \sqcup spec.continue\ f\ g$

adhoc-overloading

```
  Monad-Syntax.bind ⇌ spec.bind

setup ‹Sign.mandatory-path singleton›

lemma continue-le-conv:
  shows ⦇σ⦈ ≤ spec.continue f g
    ⟷ (∃ xs ys v w. ⦇trace.init σ, xs, Some v⦈ ≤ f
                  ∧ ⦇trace.final' (trace.init σ) xs, ys, w⦈ ≤ g v
                  ∧ σ ≤ trace.T (trace.init σ) (xs @ ys) w) (is ?lhs ⟷ ?rhs)
proof(rule iffI)
  assume ?lhs
  then obtain s xs ys v w
        where σ: ♮σ ≤ ♮(trace.T s (xs @ ys) w)
          and f: ⦇s, xs, Some v⦈ ≤ f
          and g: ⦇trace.final' s xs, ys, w⦈ ≤ g v
    by (clarsimp simp: spec.continue-def trace.split-all spec.singleton-le-conv)
  from σ show ?rhs
  proof(cases rule: trace.less-eqE)
    case prefix
    from prefix(3)[simplified, simplified trace.natural'.append] show ?thesis
    proof(cases rule: prefix-append-not-NilE)
      case incomplete
      then obtain zs where zs: trace.natural' s xs = trace.natural' (trace.init σ) (trace.rest σ) @ zs
        by (rule prefixE)
      from f prefix(2) zs
      have ⦇trace.init σ, trace.rest σ @ zs, Some v⦈ ≤ f
        by (clarsimp elim!: order.trans[rotated])
           (metis trace.natural'.append trace.final'.natural' trace.natural'.natural')
      moreover
      from g prefix(2) zs
      have ⦇trace.final' (trace.init σ) (trace.rest σ @ zs), ys, None⦈ ≤ g v
        by (clarsimp elim!: order.trans[rotated])
           (metis spec.singleton.less-eq-None trace.final'.natural' trace.final'.simps(3))
      moreover note ‹trace.term (♮σ) = None›
      ultimately show ?thesis
        by (fastforce simp: trace.less-eq-None)
    next
      case (continue us)
      from continue(1)
      obtain ys' zs'
        where trace.rest σ = ys' @ zs'
          and trace.natural' (trace.init σ) ys' = trace.natural' s xs
          and trace.natural' (trace.final' (trace.init σ) (trace.natural' s xs)) zs' = us
        by (clarsimp simp: trace.natural'.eq-append-conv)
      with f g prefix(1,2) continue(2−) show ?thesis
        by − (rule exI[where x=ys'];
              force simp: spec.singleton-le-conv trace.less-eq-None trace.natural-def
                    cong: trace.final'.natural'-cong
                    elim!: order.trans[rotated]
                    intro: exI[where x=None])
    qed
  next
    case (maximal x) with f g show ?thesis
      by (fastforce simp: trace.stuttering.equiv.append-conv
                    cong: trace.final'.natural'-cong
                    elim!: order.trans[rotated])
  qed
next
```

show *?rhs* $\Longrightarrow$ *?lhs*
using *spec.singleton.mono* **by** (*auto 10 0 simp*: *spec.continue-def trace.split-Ex*)
qed

setup ‹*Sign.parent-path*›

setup ‹*Sign.mandatory-path continue*›

lemma *mono*:
assumes $f \leq f'$
assumes $\bigwedge v.\ g\ v \leq g'\ v$
shows *spec.continue f g* $\leq$ *spec.continue f' g'*
unfolding *spec.continue-def*
apply (*strengthen ord-to-strengthen*(*1*)[*OF assms*(*1*)])
apply (*strengthen ord-to-strengthen*(*1*)[*OF assms*(*2*)])
apply (*rule order.refl*)
done

lemma *strengthen*[*strg*]:
assumes *st-ord F f f'*
assumes $\bigwedge x.$ *st-ord F* (*g x*) (*g' x*)
shows *st-ord F* (*spec.continue f g*) (*spec.continue f' g'*)
using *assms* **by** (*cases F*; *simp add*: *spec.continue.mono*)

lemma *mono2mono*[*cont-intro*, *partial-function-mono*]:
assumes *monotone orda* ($\leq$) *f*
assumes $\bigwedge x.$ *monotone orda* ($\leq$) ($\lambda y.$ *g y x*)
shows *monotone orda* ($\leq$) ($\lambda x.$ *spec.continue* (*f x*) (*g x*))
using *assms* **by** (*simp add*: *monotone-def spec.continue.mono*)

definition *resL* :: (*'v* $\Rightarrow$ (*'a*, *'s*, *'w*) *spec*) $\Rightarrow$ (*'a*, *'s*, *'w*) *spec* $\Rightarrow$ (*'a*, *'s*, *'v*) *spec* **where**
resL g P = $\bigsqcup$ {*f*. *spec.continue f g* $\leq$ *P*}

definition *resR* :: (*'a*, *'s*, *'v*) *spec* $\Rightarrow$ (*'a*, *'s*, *'w*) *spec* $\Rightarrow$ (*'v* $\Rightarrow$ (*'a*, *'s*, *'w*) *spec*) **where**
resR f P = $\bigsqcup$ {*g*. *spec.continue f g* $\leq$ *P*}

interpretation *L*: *galois.complete-lattice-class* $\lambda f.$ *spec.continue f g spec.continue.resL g* **for** *g*
proof
show *spec.continue f g* $\leq$ *P* $\longleftrightarrow$ *f* $\leq$ *spec.continue.resL g P* (**is** *?lhs* $\longleftrightarrow$ *?rhs*) **for** *f P*
proof(*rule iffI*)
assume *?rhs*
then have *spec.continue f g* $\leq$ *spec.continue* (*spec.continue.resL g P*) *g*
by (*simp add*: *spec.continue.mono*)
also have ... $\leq$ *P*
by (*auto simp*: *spec.continue.resL-def spec.continue-def*)
finally show *?lhs* **.**
qed (*simp add*: *spec.continue.resL-def Sup-upper*)
qed

interpretation *R*: *galois.complete-lattice-class* $\lambda g.$ *spec.continue f g spec.continue.resR f*
for *f* :: (*'a*, *'s*, *'v*) *spec*
proof
show *spec.continue f g* $\leq$ *P* $\longleftrightarrow$ *g* $\leq$ *spec.continue.resR f P* (**is** *?lhs* $\longleftrightarrow$ *?rhs*)
for *g* :: *'v* $\Rightarrow$ (*'a*, *'s*, *'w*) *spec*
and *P* :: (*'a*, *'s*, *'w*) *spec*
proof(*rule iffI*)
assume *?rhs*
then have *spec.continue f g* $\leq$ *spec.continue f* (*spec.continue.resR f P*)

```
    by (simp add: le-fun-def spec.continue.mono)
  also have ... ≤ P
    by (auto simp: spec.continue.resR-def spec.continue-def)
  finally show ?lhs .
 qed (simp add: spec.continue.resR-def Sup-upper)
qed

setup ‹Sign.parent-path›

setup ‹Sign.mandatory-path singleton›

lemma bind-le-conv:
 shows ⟪σ⟫ ≤ spec.bind f g ⟷ ⟪σ⟫ ≤ spec.term.none f ∨ ⟪σ⟫ ≤ spec.continue f g
by (simp add: spec.bind-def)

lemma bind-le[consumes 1]:
 assumes ⟪σ⟫ ≤ f ⤜ g
 obtains
   (incomplete) ⟪σ⟫ ≤ spec.term.none f
 | (continue) σ_f σ_g v_f
   where ⟪σ_f⟫ ≤ f and trace.final σ_f = trace.init σ_g and trace.term σ_f = Some v_f
     and ⟪σ_g⟫ ≤ g v_f and ♮σ_g ≠ trace.T (trace.init σ_g) [] None
     and σ = trace.T (trace.init σ_f) (trace.rest σ_f @ trace.rest σ_g) (trace.term σ_g)
using assms[unfolded spec.singleton.bind-le-conv]
proof(atomize-elim, induct rule: stronger-disjE[consumes 1, case-names incomplete continue])
 case continue
 from ‹⟪σ⟫ ≤ spec.continue f g› obtain xs ys v w
   where f: ⟪trace.init σ, xs, Some v⟫ ≤ f
     and g: ⟪trace.final' (trace.init σ) xs, ys, w⟫ ≤ g v
     and σ: σ ≤ trace.T (trace.init σ) (xs @ ys) w
   by (clarsimp simp: spec.singleton.continue-le-conv)
 with ‹¬ ⟪σ⟫ ≤ spec.term.none f› obtain ys'
   where ⟪trace.final' (trace.init σ) xs, ys', trace.term σ⟫ ≤ g v
     and trace.rest σ = xs @ ys'
     and trace.natural' (trace.final' (trace.init σ) xs) ys' = [] ⟶ (∃ y. trace.term σ = Some y)
   by (atomize-elim, cases σ)
      (auto elim!: trace.less-eqE prefix-append-not-NilE
            elim: order.trans[OF spec.singleton.mono, rotated]
              dest: spec.singleton.mono[OF iffD2[OF trace.less-eq-None(2)[where s=trace.init σ and σ=trace.T
(trace.init σ) xs (Some v)], simplified]]
                    order.trans[OF spec.singleton.less-eq-None]
            simp: trace.less-eq-None spec.singleton.le-conv trace.natural'.eq-Nil-conv)+
 with f show ?case
   by (cases σ) (force simp: trace.natural-def)
qed blast

setup ‹Sign.parent-path›

lemma bind-le[case-names incomplete continue]:
 assumes spec.term.none f ≤ P
 assumes ⋀σ_f σ_g v. ⟦⟪σ_f⟫ ≤ f; trace.init σ_g = trace.final σ_f; trace.term σ_f = Some v; ⟪σ_g⟫ ≤ g v;
                    ♮σ_g ≠ trace.T (trace.init σ_g) [] None⟧
                 ⟹ ⟪trace.init σ_f, trace.rest σ_f @ trace.rest σ_g, trace.term σ_g⟫ ≤ P
 shows f ⤜ g ≤ P
by (rule spec.singleton-le-extI) (use assms in ‹fastforce elim: spec.singleton.bind-le›)

setup ‹Sign.mandatory-path bind›
```

definition *resL* :: $('v \Rightarrow ('a, 's, 'w)\ spec) \Rightarrow ('a, 's, 'w)\ spec \Rightarrow ('a, 's, 'v)\ spec$ **where**
  $resL\ g\ P = \bigsqcup\{f.\ f \gg\!= g \leq P\}$

lemma *incompleteI*:
  assumes $\langle s, xs, None\rangle \leq f$
  shows $\langle s, xs, None\rangle \leq f \gg\!= g$
using *assms* **by** (*auto simp*: *spec.bind-def spec.singleton.term.none-le-conv*)

lemma *continueI*:
  assumes *f*: $\langle s, xs, Some\ v\rangle \leq f$
  assumes *g*: $\langle trace.final'\ s\ xs, ys, w\rangle \leq g\ v$
  shows $\langle s, xs\ @\ ys, w\rangle \leq f \gg\!= g$
using *assms* **by** (*force simp*: *spec.bind-def spec.continue-def intro*!: *disjI2*)

lemma *singletonL*:
  shows $\langle\sigma\rangle \gg\!= g$
    $= spec.term.none\ \langle\sigma\rangle$
      $\sqcup \bigsqcup\{\langle trace.init\ \sigma, trace.rest\ \sigma\ @\ trace.rest\ \sigma_g, trace.term\ \sigma_g\rangle\ |\sigma_g.$
          $trace.final\ \sigma = trace.init\ \sigma_g \wedge (\exists v.\ trace.term\ \sigma = Some\ v \wedge \langle\sigma_g\rangle \leq g\ v)\}$ (**is** *?lhs* = *?rhs*)
proof(*rule antisym*)
  show *?lhs* $\leq$ *?rhs*
  proof(*induct rule*: *spec.bind-le*)
    case (*continue* $\sigma_f\ \sigma_g\ v$) **then show** *?case*
      by (*cases* σ_f; *cases* σ_g)
        (*simp add*: *trace.split-Ex*;
          metis order.refl spec.singleton.simps(*1*) *trace.final'.natural' trace.stuttering.equiv.append-cong*)
  qed *force*
  show *?rhs* $\leq$ *?lhs*
    by (*cases* σ)
      (*force simp*: *spec.term.none.singleton spec.singleton.bind-le-conv spec.singleton.continue-le-conv*)
qed

lemma *mono*:
  assumes $f \leq f'$
  assumes $\bigwedge v.\ g\ v \leq g'\ v$
  shows $spec.bind\ f\ g \leq spec.bind\ f'\ g'$
unfolding *spec.bind-def*
apply (*strengthen ord-to-strengthen*(*1*)[*OF assms*(*1*)])
apply (*strengthen ord-to-strengthen*(*1*)[*OF assms*(*2*)])
apply (*rule order.refl*)
done

lemma *strengthen*[*strg*]:
  assumes $st\text{-}ord\ F\ f\ f'$
  assumes $\bigwedge x.\ st\text{-}ord\ F\ (g\ x)\ (g'\ x)$
  shows $st\text{-}ord\ F\ (spec.bind\ f\ g)\ (spec.bind\ f'\ g')$
using *assms* **by** (*cases F*; *simp add*: *spec.bind.mono*)

lemma *mono2mono*[*cont-intro*, *partial-function-mono*]:
  assumes $monotone\ orda\ (\leq)\ f$
  assumes $\bigwedge x.\ monotone\ orda\ (\leq)\ (\lambda y.\ g\ y\ x)$
  shows $monotone\ orda\ (\leq)\ (\lambda x.\ spec.bind\ (f\ x)\ (g\ x))$
using *assms* **by** (*simp add*: *monotone-def spec.bind.mono*)

interpretation *L*: *galois.complete-lattice-class* $\lambda f.\ f \gg\!= g$ *spec.bind.resL g* **for** *g*
proof
  show $f \gg\!= g \leq P \longleftrightarrow f \leq spec.bind.resL\ g\ P$ (**is** *?lhs* $\longleftrightarrow$ *?rhs*) **for** *f P*
  proof(*rule iffI*)

assume *?rhs*
then have $f \gg\!= g \leq$ *spec.bind.resL g P* $\gg\!= g$
by (*simp add*: *spec.bind.mono*)
also have $\ldots \leq P$
by (*simp add*: *spec.bind.resL-def spec.bind-def spec.term.none.Sup spec.continue.L.lower-Sup*)
finally show *?lhs* **.**
qed (*simp add*: *spec.bind.resL-def Sup-upper*)
qed

lemmas *SUPL* = *spec.bind.L.lower-SUP*
lemmas *SupL* = *spec.bind.L.lower-Sup*
lemmas *supL* = *spec.bind.L.lower-sup*[*of* f_1 f_2 g] **for** f_1 f_2 g

lemmas *INFL-le* = *spec.bind.L.lower-INF-le*
lemmas *InfL-le* = *spec.bind.L.lower-Inf-le*
lemmas *infL-le* = *spec.bind.L.lower-inf-le*[*of* f_1 f_2 g] **for** f_1 f_2 g

lemma *SUPR*:
shows *spec.bind f* $(\lambda v.\ \bigsqcup x{\in}X.\ g\ x\ v) = (\bigsqcup x{\in}X.\ f \gg\!= g\ x) \sqcup (f \gg\!= \bot)$ (**is** *?thesis1*) — *Sup* over $('a, 's, 'v)$ *spec*
and *spec.bind f* $(\bigsqcup x{\in}X.\ g\ x) = (\bigsqcup x{\in}X.\ f \gg\!= g\ x) \sqcup (f \gg\!= \bot)$ (**is** *?thesis2*) — *Sup* over functions
proof −
show *?thesis1*
by (*cases* $X = \{\}$)
(*simp-all add*: *spec.bind-def spec.continue.R.lower-bot sup-SUP ac-simps*
spec.continue.R.lower-SUP[**where** $f{=}g$ **and** $X{=}X$, *unfolded Sup-fun-def image-image*]
flip: *bot-fun-def*)
then show *?thesis2*
by (*simp add*: *Sup-fun-def image-image*)
qed

lemma *SUPR-not-empty*:
assumes $X \neq \{\}$
shows *spec.bind f* $(\lambda v.\ \bigsqcup x{\in}X.\ g\ x\ v) = (\bigsqcup x{\in}X.\ f \gg\!= g\ x)$
using *assms* **by** (*clarsimp simp*: *spec.bind.SUPR spec.bind.mono sup.absorb1 SUPI simp flip*: *ex-in-conv*)

lemmas *supR* = *spec.bind.SUPR-not-empty*[**where** $g{=}id$ **and** $X{=}\{g_1, g_2\}$ **for** g_1 g_2, *simplified*]

lemma *InfR-le*:
shows *spec.bind f* $(\lambda v.\ \bigsqcap x{\in}X.\ g\ x\ v) \leq (\bigsqcap x{\in}X.\ f \gg\!= g\ x)$
by (*meson INF-lower order.refl le-INF-iff spec.bind.mono*)

lemma *infR-le*:
shows *spec.bind f* $(g_1 \sqcap g_2) \leq (f \gg\!= g_1) \sqcap (f \gg\!= g_2)$
and *spec.bind f* $(\lambda v.\ g_1\ v \sqcap g_2\ v) \leq (f \gg\!= g_1) \sqcap (f \gg\!= g_2)$
by (*simp-all add*: *spec.bind.mono*)

lemma *Inf-le*:
shows *spec.bind* $(\bigsqcap x{\in}X.\ f\ x)\ (\lambda v.\ (\bigsqcap x{\in}X.\ g\ x\ v)) \leq (\bigsqcap x{\in}X.\ spec.bind\ (f\ x)\ (g\ x))$
by (*auto simp*: *le-INF-iff intro*: *spec.bind.mono*)

lemma *inf-le*:
shows *spec.bind* $(f_1 \sqcap f_2)\ (\lambda v.\ g_1\ v \sqcap g_2\ v) \leq$ *spec.bind* $f_1\ g_1 \sqcap$ *spec.bind* $f_2\ g_2$
by (*simp add*: *spec.bind.mono*)

lemma *mcont2mcont*[*cont-intro*]:
assumes *mcont luba orda Sup* $(\leq)$ f
assumes $\bigwedge v.$ *mcont luba orda Sup* $(\leq)$ $(\lambda x.\ g\ x\ v)$

shows *mcont luba orda Sup* $(\leq)$ $(\lambda x.\ spec.bind\ (f\ x)\ (g\ x))$
proof(*rule ccpo.mcont2mcont'*[*OF complete-lattice-ccpo - - assms*(*1*)])
show *mcont Sup* $(\leq)$ *Sup* $(\leq)$ $(\lambda f.\ bind\ f\ (g\ x))$ **for** *x*
by (*intro mcontI contI monotoneI*) (*simp-all add*: *spec.bind.mono flip*: *spec.bind.SUPL*)
show *mcont luba orda Sup* $(\leq)$ $(\lambda x.\ bind\ f\ (g\ x))$ **for** *f*
by (*intro mcontI monotoneI contI*)
(*simp-all add*: *mcont-monoD*[*OF assms*(*2*)] *spec.bind.mono*
flip: *spec.bind.SUPR-not-empty contD*[*OF mcont-cont*[*OF assms*(*2*)]])
qed

lemmas *botL*[*simp*] = *spec.bind.L.lower-bot*

lemma *botR*:
shows $f \gg\!= \bot = spec.term.none\ f$
by (*simp add*: *spec.bind-def spec.continue.R.lower-bot*)

lemma *eq-bot-conv*:
shows $spec.bind\ f\ g = \bot \longleftrightarrow f = \bot$
by (*fastforce simp*: *spec.continue.L.lower-bot spec.bind-def spec.term.galois simp flip*: *bot.extremum-unique*)

lemma *idleL*[*simp*]:
shows $spec.idle \gg\!= g = spec.idle$
by (*simp add*: *spec.idle-def spec.bind.SupL image-image spec.bind.singletonL spec.term.none.singleton*)

lemma *idleR*:
shows $f \gg spec.idle = f \gg\!= \bot$ (**is** *?lhs* = *?rhs*)
proof(*rule antisym*)
show *?lhs* $\leq$ *?rhs*
by (*fastforce simp*: *spec.bind.botR trace.split-all spec.singleton.le-conv*
intro!: *spec.bind-le*
intro: *spec.bind.incompleteI order.trans*[*rotated*])
show *?rhs* $\leq$ *?lhs*
by (*simp add*: *spec.bind.mono*)
qed

lemmas *ifL* = *if-distrib*[**where** $f{=}\lambda f.\ spec.bind\ f\ g$ **for** *g*]

setup ‹*Sign.parent-path*›

setup ‹*Sign.mandatory-path idle*›

lemma *bind-le-conv*[*spec.idle-le*]:
shows $spec.idle \leq f \gg\!= g \longleftrightarrow spec.idle \leq f$ (**is** *?lhs* $\longleftrightarrow$ *?rhs*)
proof(*rule iffI*)
show *?lhs* $\Longrightarrow$ *?rhs*
by (*fastforce simp*: *spec.idle-def spec.singleton.mono trace.less-eq-None spec.singleton.bind-le-conv*
spec.singleton.term.none-le-conv spec.singleton.continue-le-conv
elim: *order.trans*[*rotated*])
show *?rhs* $\Longrightarrow$ *?lhs*
by (*simp add*: *spec.bind-def spec.idle.term.none-le-conv le-supI1*)
qed

setup ‹*Sign.parent-path*›

setup ‹*Sign.mandatory-path term*›

setup ‹*Sign.mandatory-path none*›

lemma *bindL-le*[*iff*]:
shows *spec.term.none f* ≤ *f* ⤜ *g*
by (*simp add*: *spec.bind-def*)

lemma *bind*:
shows *spec.term.none* (*f* ⤜ *g*) = *f* ⤜ (λ*v*. *spec.term.none* (*g v*))
by (*rule spec.singleton.antisym*)
(*auto elim*: *spec.singleton.bind-le*
simp: *trace.split-all spec.bind.incompleteI spec.bind.continueI spec.singleton.term.none-le-conv*)

setup ‹*Sign.parent-path*›

setup ‹*Sign.mandatory-path all*›

lemma *bind*:
shows *spec.term.all* (*f* ⤜ *g*) = *spec.term.all f* ⊔ (*f* ⤜ (λ*v*. *spec.term.all* (*g v*))) (**is** *?lhs* = *?rhs*)
proof(*rule antisym*[*OF spec.singleton-le-extI*])
show ⟪*σ*⟫ ≤ *?rhs* **if** ⟪*σ*⟫ ≤ *?lhs* **for** *σ*
using *that*
by (*cases σ*)
(*fastforce simp*: *trace.split-all spec.singleton.le-conv*
elim!: *spec.singleton.bind-le*
intro: *spec.bind.continueI*)
show *?rhs* ≤ *?lhs*
by (*simp add*: *spec.term.none.sup spec.term.none.bind spec.bind.mono flip*: *spec.term.galois*)
qed

setup ‹*Sign.parent-path*›

setup ‹*Sign.parent-path*›

The monad laws for (⤜)**.** **setup** ‹*Sign.mandatory-path bind*›

lemma *bind*:
fixes *f* :: (-, -, -) *spec*
shows *f* ⤜ *g* ⤜ *h* = *f* ⤜ (λ*v*. *g v* ⤜ *h*) (**is** *?lhs* = *?rhs*)
proof(*rule antisym*)
show *?lhs* ≤ *?rhs*
proof(*induct rule*: *spec.bind-le*)
case *incomplete* **show** *?case*
by (*simp add*: *spec.bind.mono spec.term.none.bind*)
next
case (*continue* σ_{fg} σ_h *v*) **then show** *?case*
by (*cases* σ_h)
(*fastforce elim*: *spec.singleton.bind-le spec.bind.continueI*
simp: *spec.singleton.le-conv trace.split-all*)
qed
show *?rhs* ≤ *?lhs*
proof(*induct rule*: *spec.bind-le*)
case *incomplete* **show** *?case*
by (*strengthen ord-to-strengthen*(*2*)[*OF spec.term.none.bindL-le*])
(*simp add*: *spec.term.none.bind*)
next
case (*continue* σ_f σ_{gh} *v*)
note ∗ = *continue.hyps*(*1*−*3*)
from ‹⟪σ_{gh}⟫ ≤ *g v* ⤜ *h*› **show** *?case*
proof(*cases rule*: *spec.singleton.bind-le*)
case *incomplete* **with** ∗ **show** *?thesis*

```
      by (cases σ_f)
        (clarsimp simp: spec.singleton.le-conv spec.bind.incompleteI spec.bind.continueI)
    next
      case (continue σ_g σ_h v_g) with ∗ show ?thesis
      by (cases σ_f; cases σ_g)
        (simp flip: append-assoc; fastforce intro!: spec.bind.continueI)
    qed
  qed
qed

lemmas assoc = spec.bind.bind

lemma returnL-le:
  shows g v ≤ spec.return v ≫= g (is ?lhs ≤ ?rhs)
proof(rule spec.singleton-le-extI)
  show ⟪σ⟫ ≤ ?rhs if ⟪σ⟫ ≤ ?lhs for σ
    by (rule spec.bind.continueI[where xs=[] and s=trace.init σ and ys=trace.rest σ and w=trace.term σ and
v=v, simplified])
       (simp-all add: spec.return-def spec.action.stutterI that)
qed

lemma returnL:
  assumes spec.idle ≤ g v
  shows spec.return v ≫= g = g v
by (rule antisym[OF spec.bind-le spec.bind.returnL-le])
   (simp-all add: assms spec.term.none.return spec.singleton.return-le-conv trace.split-all)

lemma returnR[simp]:
  shows f ≫= spec.return = f (is ?lhs = ?rhs)
proof(rule antisym[OF - spec.singleton-le-extI])
  show ?lhs ≤ ?rhs
    by (auto intro: spec.bind-le
             simp: trace.split-all spec.singleton.return-le-conv order.trans[OF spec.singleton.less-eq-None(1)]
            split: option.split-asm)
  show ⟪σ⟫ ≤ ?lhs if ⟪σ⟫ ≤ ?rhs for σ
    using that
    by (cases σ; cases trace.term σ;
        clarsimp simp: spec.bind.incompleteI spec.bind.continueI[where ys=[], simplified] spec.singleton.le-conv)
qed

lemma return: — Does not require spec.idle ≤ g v
  fixes f :: ('a, 's, 'v) spec
  fixes g :: 'v ⇒ 'x ⇒ ('a, 's, 'w) spec
  shows f ≫= (λv. spec.return x ≫= g v) = f ≫= (λv. g v x) (is ?lhs = ?rhs)
proof(rule antisym)
  show ?lhs ≤ ?rhs
  proof(induct rule: spec.bind-le)
    case (continue σ_f σ_rg v)
    from ‹⟪σ_rg⟫ ≤ spec.return x ≫= g v› show ?case
    proof(induct rule: spec.singleton.bind-le[case-names incomplete continue2])
      case incomplete with ‹⟪σ_f⟫ ≤ f› ‹trace.init σ_rg = trace.final σ_f› show ?thesis
        by (cases σ_f)
          (auto simp: spec.term.none.return spec.singleton.le-conv
               intro: spec.bind.incompleteI order.trans[rotated])
    next
      case (continue2 σ_r σ_g v_r) with continue show ?case
        by (cases σ_f) (simp add: trace.split-all spec.singleton.le-conv spec.bind.continueI)
    qed
```

qed *simp*
show *?rhs* ≤ *?lhs*
by (*simp add*: *spec.bind.mono spec.bind.returnL-le*)
qed

setup ‹*Sign.mandatory-path term*›

lemma *noneL*[*simp*]:
shows *spec.term.none f* ⤜ *g* = *spec.term.none f*
by (*simp add*: *spec.bind.bind flip*: *spec.bind.botR bot-fun-def*)

setup ‹*Sign.parent-path*›

setup ‹*Sign.parent-path*›

setup ‹*Sign.mandatory-path map*›

lemma *bind-le*: — Converse does not hold: it may be that no final states of *f* satisfy *g*
fixes *f* :: ('*a*, '*s*, '*v*) *spec*
fixes *g* :: '*v* ⇒ ('*a*, '*s*, '*w*) *spec*
fixes *af* :: '*a* ⇒ '*b*
fixes *sf* :: '*s* ⇒ '*t*
fixes *vf* :: '*w* ⇒ '*x*
shows *spec.map af sf vf* (*f* ⤜ *g*) ≤ *spec.map af sf id f* ⤜ (λ*v*. *spec.map af sf vf* (*g v*))
by (*subst* (*1*) *spec.map-def*)
(*force simp*: *spec.singleton.le-conv trace.split-all trace.final'.map*
intro: *spec.bind.incompleteI spec.bind.continueI*
elim: *spec.singleton.bind-le*)

lemma *bind-inj-sf*:
fixes *f* :: ('*a*, '*s*, '*x*) *spec*
fixes *g* :: '*x* ⇒ ('*a*, '*s*, '*v*) *spec*
assumes *inj sf*
shows *spec.map af sf vf* (*f* ⤜ *g*) = *spec.map af sf id f* ⤜ (λ*v*. *spec.map af sf vf* (*g v*)) (**is** *?lhs* = *?rhs*)
proof(*rule antisym*[*OF spec.map.bind-le*])
show *?rhs* ≤ *?lhs*
proof(*induct rule*: *spec.bind-le*)
case *incomplete* **show** *?case*
by (*metis spec.map.mono spec.term.none.bindL-le spec.term.none.map-gen*)
next
case (*continue* σ_f σ_g *v*)
from *continue*(*1,4*) **obtain** σ_f' σ_g'
where ∗: ⟪σ_f'⟫ ≤ *f* ⟪σ_f⟫ ≤ ⟪*trace.map af sf id* σ_f'⟫
⟪σ_g'⟫ ≤ *g v* ⟪σ_g⟫ ≤ ⟪*trace.map af sf vf* σ_g'⟫
by (*clarsimp simp*: *spec.singleton.le-conv*)
with *continue*(*2,3*)
have *sf* (*trace.init* σ_g') = *sf* (*trace.final* σ_f')
by (*cases* σ_f; *cases* σ_g; *cases* σ_f'; *cases* σ_g'; *clarsimp*)
(*clarsimp simp*: *spec.singleton-le-conv simp flip*: *trace.final'.map*[**where** *af*=*af* **and** *sf*=*sf*];
erule trace.less-eqE; *simp add*: *trace.natural.trace-conv*; *metis trace.final'.natural'*)
with *continue*(*2,3*) ∗ **show** *?case*
by (*cases* σ_f; *cases* σ_g; *cases* σ_f'; *cases* σ_g')
(*fastforce dest*: *inj-onD*[*OF assms*, *simplified*]
elim: *trace.less-eqE spec.bind.continueI*
simp: *spec.singleton.le-conv trace.final'.map trace.less-eq-None*
spec.singleton-le-conv trace.natural-def trace.natural'.append)
qed
qed

setup ‹*Sign.parent-path*›

setup ‹*Sign.mandatory-path vmap*›

lemma *eq-return*: — generalizes *spec.bind.returnR*
shows *spec.vmap vf P* = *P* ⤜ *spec.return* ∘ *vf* (**is** *?thesis1*)
and *spec.vmap vf P* = *P* ⤜ (λ*v*. *spec.return* (*vf v*)) (**is** *?lhs* = *?rhs*) — useful for flip/symmetric
proof −
show *?lhs* = *?rhs*
proof(*rule antisym*)
show *?lhs* ≤ *?rhs*
by (*rule spec.singleton.exhaust*[*of P*])
(*fastforce simp*: *trace.split-all spec.singleton.le-conv spec.map.Sup spec.map.singleton map-option-case*
intro: *spec.bind.incompleteI spec.bind.continueI*[**where** *ys*=[], *simplified*]
split: *option.split*)
next
show *?rhs* ≤ *?lhs*
by (*rule spec.bind-le*)
(*force simp*: *trace.split-all spec.singleton.le-conv trace.less-eq-None trace.natural.mono*
spec.term.galois spec.term.all.expansive spec.term.all.map spec.map.id
split: *option.split-asm*)+
qed
then show *?thesis1*
by (*simp add*: *comp-def*)
qed

lemma *unitL*: — monomorphise ignored return values
shows *f* ⟫ *g* = *spec.vmap* ⟨()⟩ *f* ⟫ *g*
by (*simp add*: *spec.vmap.eq-return comp-def spec.bind.bind spec.bind.return*)

setup ‹*Sign.parent-path*›

setup ‹*Sign.mandatory-path invmap*›

lemma *bind*:
fixes *f* :: ('*b*, '*t*, '*v*) *spec*
fixes *g* :: '*v* ⇒ ('*b*, '*t*, '*x*) *spec*
fixes *af* :: '*a* ⇒ '*b*
fixes *sf* :: '*s* ⇒ '*t*
fixes *vf* :: '*w* ⇒ '*x*
shows *spec.invmap af sf vf* (*f* ⤜ *g*) = *spec.invmap af sf id f* ⤜ (λ*v*. *spec.invmap af sf vf* (*g v*)) (**is** *?lhs* = *?rhs*)
proof(*rule antisym*[*OF spec.singleton-le-extI*])
fix σ **assume** ⟪σ⟫ ≤ *?lhs*
then have ⟪*trace.map af sf vf* σ⟫ ≤ *f* ⤜ *g* **by** (*simp add*: *spec.singleton.le-conv*)
then show ⟪σ⟫ ≤ *?rhs*
proof(*induct rule*: *spec.singleton.bind-le*)
case *incomplete* **then show** *?case*
by (*cases* σ) (*clarsimp simp*: *spec.singleton.le-conv spec.bind.incompleteI*)
next
case (*continue* σ_f σ_g v_f) **then show** *?case*
by (*cases* σ; *cases* σ_f; *cases* σ_g)
(*clarsimp simp*: *spec.bind.continueI map-eq-append-conv spec.singleton.le-conv trace.final'.map*)
qed
next
show *?rhs* ≤ *?lhs*
by (*simp add*: *order.trans*[*OF spec.map.bind-le*] *spec.bind.mono spec.map-invmap.lower-upper-contractive*

```
        flip: spec.map-invmap.galois)
qed

lemma split-vinvmap:
  shows spec.invmap af sf vf P = spec.invmap af sf id P ⤜ (λv. ⨆v'∈vf − ‘ {v}. spec.return v') (is ?lhs = ?rhs)
proof(rule antisym[OF spec.singleton-le-extI])
  show ⟪σ⟫ ≤ ?rhs if ⟪σ⟫ ≤ ?lhs for σ
    using that
    by (cases σ; cases trace.term σ)
       (auto simp: spec.singleton.le-conv
            intro: spec.bind.incompleteI spec.bind.continueI[where ys=[], simplified])
  show ?rhs ≤ ?lhs
  proof(induct rule: spec.bind-le)
    case (continue σ_f σ_g v) then show ?case
      by (cases σ_f; cases trace.term σ_g)
         (auto simp: spec.singleton.le-conv split: option.split-asm elim: order.trans[rotated])
  qed (simp add: spec.term.none.invmap-gen[where vf'=vf] spec.invmap.mono)
qed

setup ‹Sign.parent-path›

setup ‹Sign.mandatory-path action›

lemma return-const:
  assumes V ≠ {}
  assumes W ≠ {}
  shows spec.action (V × F) = spec.action (W × F) ≫ (⨆v∈V. spec.return v) (is ?lhs = ?rhs)
proof(rule antisym)
  from ‹W ≠ {}› show ?lhs ≤ ?rhs
    by − (rule spec.action-le;
          fastforce intro: spec.bind.continueI[where xs=[x] and v=SOME w. w ∈ W for x, simplified]
                           spec.action.stepI
                   simp: some-in-eq spec.singleton.le-conv spec.singleton.action-le-conv
                           spec.idle.action-le spec.idle.bind-le-conv)
  from ‹V ≠ {}› show ?rhs ≤ ?lhs
    by − (rule spec.bind-le,
          fastforce simp: spec.term.galois spec.term.all.action intro: le-supI1 spec.action.mono,
          auto 0 3 simp: spec.singleton.le-conv spec.singleton.action-le-conv
              simp flip: trace.steps'.empty-conv
               simp del: trace.steps'.simps split: option.splits)
qed

setup ‹Sign.parent-path›

setup ‹Sign.mandatory-path term.closed›

lemma bind-all-return:
  assumes f ∈ spec.term.closed -
  shows f ≫ (⨆range spec.return) = spec.term.all f (is ?lhs = ?rhs)
proof(rule antisym[OF - spec.singleton-le-extI])
  show ?lhs ≤ ?rhs
    by (subst (2) spec.term.closed-conv[OF assms])
       (simp add: spec.term.none.bind spec.term.none.Sup image-image spec.term.none.return
                  spec.bind.botR spec.bind.idleR
                  spec.term.all.monomorphic
           flip: spec.term.galois)
next
  fix σ
```

assume $\langle\!|\sigma|\!\rangle \leq$ *?rhs*
then obtain *v* **where** $\langle\!|$*trace.init* σ, *trace.rest* σ, *Some v*$|\!\rangle \leq f$
by (*subst* (*asm*) *spec.term.closed-conv*[*OF assms*])
(*force simp*: *spec.singleton.le-conv spec.term.all.monomorphic*)
then show $\langle\!|\sigma|\!\rangle \leq$ *?lhs*
by (*cases* σ; *cases trace.term* σ)
(*auto simp*: *spec.singleton.le-conv spec.bind.continueI*[**where** *ys*=[], *simplified*]
split: *option.split*)
qed

setup ‹*Sign.parent-path*›

setup ‹*Sign.parent-path*›

8.9 Kleene star

We instantiate the generic Kleene locale with monomorphic *spec.return* (). The polymorphic ($\bigsqcup v$. *spec.return v*) fails the *comp-unitR* axiom ($\varepsilon \leq x \Longrightarrow x \cdot \varepsilon = x$).

setup ‹*Sign.mandatory-path spec*›

interpretation *kleene*: *weak-kleene spec.return* () $\lambda x\ y$. *spec.bind x* $\langle y \rangle$
by *standard* (*simp-all add*: *spec.bind.bind spec.bind.supL spec.bind.supR*
spec.bind.returnL order.trans[*OF spec.idle.return-le*])

setup ‹*Sign.mandatory-path idle.kleene*›

lemmas *star-le*[*spec.idle-le*] = *order.trans*[*OF spec.idle.return-le spec.kleene.epsilon-star-le*]

lemmas *rev-star-le*[*spec.idle-le*] = *spec.idle.kleene.star-le*[*unfolded spec.kleene.star-rev-star*]

setup ‹*Sign.parent-path*›

setup ‹*Sign.mandatory-path return.kleene*›

lemmas *star-le* = *spec.kleene.epsilon-star-le*

lemmas *rev-star-le* = *spec.return.kleene.star-le*[*unfolded spec.kleene.star-rev-star*]

setup ‹*Sign.parent-path*›

setup ‹*Sign.mandatory-path kleene*›

lemma *star-idle*:
shows *spec.kleene.star spec.idle* = *spec.return* ()
by (*subst spec.kleene.star.simps*) (*simp add*: *sup.absorb2 spec.idle.return-le*)

lemmas *rev-star-idle* = *spec.kleene.star-idle*[*unfolded spec.kleene.star-rev-star*]

setup ‹*Sign.parent-path*›

setup ‹*Sign.mandatory-path term.all.kleene*›

lemma *star-closed-le*:
fixes *P* :: (-, -, *unit*) *spec*
assumes *P* ∈ *spec.term.closed* -
shows *spec.term.all* (*spec.kleene.star P*) ≤ *spec.kleene.star P* (**is** - ≤ *?rhs*)
proof(*induct rule*: *spec.kleene.star.fixp-induct*[**where** *P*=λR. *spec.term.all* (*R P*) ≤ *?rhs*, *case-names adm bot step*])

case (*step R*) **show** *?case*
by (*auto simp*: *spec.term.all.sup spec.term.all.bind spec.kleene.expansive-star spec.term.all.return*
simp flip: *spec.term.all.closed-conv*[*OF assms*]
intro: *spec.kleene.epsilon-star-le*
order.trans[*OF spec.bind.mono*[*OF order.refl step*] *spec.kleene.fold-starL*])
qed *simp-all*

setup ‹*Sign.parent-path*›

setup ‹*Sign.mandatory-path term.closed.kleene*›

lemma *star*:
assumes $P \in$ *spec.term.closed -*
shows *spec.kleene.star* $P \in$ *spec.term.closed -*
by (*rule spec.term.closed-clI*)
(*simp add*: *spec.term.all.kleene.star-closed-le*[*OF assms*] *spec.term.all.monomorphic*)

setup ‹*Sign.parent-path*›

setup ‹*Sign.parent-path*›

8.10 Transition relations

Using *spec.kleene.star* we can specify the transitions each agent is allowed to perform. These constraints ((⊓) *spec.rel r*) distribute through all program constructs (for suitable r).
Observations:

- the Galois connection between *spec.rel* and *spec.steps* is much easier to show in the powerset model
  - see van Staden (2015, Footnote 2)
- most useful facts about *spec.steps* depend on the model

setup ‹*Sign.mandatory-path spec*›

setup ‹*Sign.mandatory-path rel*›

definition *act* :: ('*a*, '*s*) *steps* ⇒ ('*a*, '*s*, *unit*) *spec* **where** — lift above *spec.return* to ease some proofs
act r = *spec.action* ({()} × (*r* ∪ *UNIV* × *Id*))

abbreviation *monomorphic* :: ('*a*, '*s*) *steps* ⇒ ('*a*, '*s*, *unit*) *spec* **where**
monomorphic r ≡ *spec.kleene.star* (*spec.rel.act r*)

lemma *act-alt-def*:
shows *spec.rel.act r* = *spec.action* ({()} × *r*) ⊔ *spec.return* ()
by (*simp add*: *spec.rel.act-def spec.return-def Sigma-Un-distrib2 flip*: *spec.action.sup*)

setup ‹*Sign.parent-path*›

definition *rel* :: ('*a*, '*s*) *steps* ⇒ ('*a*, '*s*, '*v*) *spec* **where**
rel r = *spec.term.all* (*spec.rel.monomorphic r*)

definition *steps* :: ('*a*, '*s*, '*v*) *spec* ⇒ ('*a*, '*s*) *steps* **where**
steps P = ⋂{*r*. *P* ≤ *spec.rel r*}

setup ‹*Sign.mandatory-path rel.act*›

lemma *monotone*:

shows *mono spec.rel.act*
proof(*rule monotoneI*)
show *spec.rel.act r* ≤ *spec.rel.act r'* **if** *r* ⊆ *r'* **for** *r r'* :: (*'a*, *'s*) *steps*
using *that* **unfolding** *spec.rel.act-def* **by** (*strengthen ord-to-strengthen*(*1*)[*OF* ‹*r* ≤ *r'*›]) *simp*
qed

lemmas *strengthen*[*strg*] = *st-monotone*[*OF spec.rel.act.monotone*]
lemmas *mono* = *monotoneD*[*OF spec.rel.act.monotone*]

lemma *empty*:
shows *spec.rel.act* {} = *spec.return* ()
by (*simp add*: *spec.rel.act-def spec.return-def spec.action.empty*)

lemma *UNIV*:
shows *spec.rel.act UNIV* = *spec.action* ({()} × *UNIV*)
by (*simp add*: *spec.rel.act-def*)

lemma *sup*:
shows *spec.rel.act* (*r* ∪ *s*) = *spec.rel.act r* ⊔ *spec.rel.act s*
by (*fastforce simp*: *spec.rel.act-def simp flip*: *spec.action.sup intro*: *arg-cong*[**where** *f*=*spec.action*])

lemma *stutter*:
shows *spec.rel.act* (*UNIV* × *Id*) = *spec.return* ()
by (*simp add*: *spec.rel.act-def spec.return-def*)

setup ‹*Sign.parent-path*›

setup ‹*Sign.mandatory-path term*›

setup ‹*Sign.mandatory-path all*›

setup ‹*Sign.mandatory-path rel*›

lemma *act-mono*:
shows *spec.term.all* (*spec.rel.act r*) = *spec.rel.act r*
by (*simp add*: *spec.rel.act-alt-def spec.term.all.sup spec.term.all.action spec.term.all.return UNIV-unit*)

setup ‹*Sign.parent-path*›

lemma *rel*:
shows *spec.term.all* (*spec.rel r*) = *spec.rel r*
by (*simp add*: *spec.rel-def*)

setup ‹*Sign.parent-path*›

setup ‹*Sign.mandatory-path closed*›

setup ‹*Sign.mandatory-path rel*›

lemma *act*:
shows *spec.rel.act r* ∈ *spec.term.closed* -
by (*metis spec.term.all.rel.act-mono spec.term.all.closed*)

setup ‹*Sign.parent-path*›

lemma *rel*:
shows *spec.rel r* ∈ *spec.term.closed* -
by (*metis spec.term.all.closed spec.term.all.rel*)

setup ‹*Sign.parent-path*›

setup ‹*Sign.mandatory-path none*›

lemma *inf-none-rel*: — polymorphic constants
  shows *spec.term.none* (*spec.rel r* :: ('*a*, '*s*, '*w*) *spec*) ⊓ *spec.term.none P*
      = *spec.rel r* ⊓ (*spec.term.none P* :: ('*a*, '*s*, '*v*) *spec*) (**is** *?thesis1*)
    and *spec.term.none P* ⊓ *spec.term.none* (*spec.rel r* :: ('*a*, '*s*, '*w*) *spec*)
      = *spec.term.none P* ⊓ (*spec.rel r* :: ('*a*, '*s*, '*v*) *spec*) (**is** *?thesis2*)
proof −
  show *?thesis1*
    by (*metis spec.term.closed.rel spec.term.closed.none-inf*(*1*)
          spec.term.none.idempotent spec.term.none.inf(*2*) *spec.term.none-all spec.term.all.rel*)
  then show *?thesis2*
    by (*simp add*: *ac-simps*)
qed

lemma *inf-rel*:
  shows *spec.term.none P* ⊓ *spec.rel r* = *spec.term.none* (*P* ⊓ *spec.rel r*) (**is** *?thesis1*)
    and *spec.rel r* ⊓ *spec.term.none P* = *spec.term.none* (*spec.rel r* ⊓ *P*) (**is** *?thesis2*)
by (*simp-all add*: *ac-simps spec.term.none.inf*(*2*) *spec.term.none.inf-none-rel*(*2*))

setup ‹*Sign.parent-path*›

setup ‹*Sign.parent-path*›

setup ‹*Sign.mandatory-path return*›

setup ‹*Sign.mandatory-path rel*›

lemma *act-le*:
  shows *spec.return* () ≤ *spec.rel.act r*
by (*simp add*: *spec.rel.act.mono flip*: *spec.rel.act.empty*)

setup ‹*Sign.parent-path*›

lemma *rel-le*:
  shows *spec.return v* ≤ *spec.rel r*
by (*simp add*: *spec.rel-def spec.term.none.return spec.idle.kleene.star-le flip*: *spec.term.galois*)

lemma *Sup-rel-le*:
  shows ⨆ *range spec.return* ≤ *spec.rel r*
by (*simp add*: *spec.return.rel-le*)

setup ‹*Sign.parent-path*›

setup ‹*Sign.mandatory-path idle*›

setup ‹*Sign.mandatory-path rel*›

lemmas *act-le*[*spec.idle-le*] = *order.trans*[*OF spec.idle.return-le spec.return.rel.act-le*]

setup ‹*Sign.parent-path*›

lemmas *rel-le*[*spec.idle-le*] = *order.trans*[*OF spec.idle.return-le spec.return.rel-le*]

setup ‹*Sign.parent-path*›

setup ‹*Sign.mandatory-path singleton*›

setup ‹*Sign.mandatory-path rel*›

setup ‹*Sign.mandatory-path act*›

lemma *le-conv*[*spec.singleton.le-conv*]:
  shows $\langle\!|\sigma|\!\rangle \leq$ *spec.rel.act r* $\longleftrightarrow$ *trace.steps* $\sigma = \{\} \vee (\exists\, x{\in}r.$ *trace.steps* $\sigma = \{x\})$
by (*auto simp*: *spec.rel.act-def spec.singleton.le-conv spec.singleton.action-le-conv trace.steps′.step-conv*
      split: *option.split*)

setup ‹*Sign.parent-path*›

setup ‹*Sign.mandatory-path monomorphic*›

lemma *le-steps*:
  assumes *trace.steps* $\sigma \subseteq r$
  shows $\langle\!|\sigma|\!\rangle \leq$ *spec.rel.monomorphic r*
using *assms*
proof(*induct trace.rest* σ *arbitrary*: σ *rule*: *rev-induct*)
  case *Nil* **then show** *?case*
    by (*simp add*: *spec.singleton.rel.act.le-conv order.trans*[*OF - spec.kleene.expansive-star*])
next
  case (*snoc x xs* σ)
  from *snoc*(*2*,*3*)
  have $*$: $\langle\!|$*trace.init* σ, *xs*, *Some* ()$|\!\rangle \leq$ *spec.rel.monomorphic r*
    by (*cases* σ) (*fastforce intro*: *snoc*(*1*) *simp*: *trace.steps′.append*)
  have $**$: $\langle\!|$*trace.final′* (*trace.init* σ) *xs*, [*x*], *trace.term* $\sigma|\!\rangle \leq$ *spec.rel.act r*
  proof(*cases trace.final′* (*trace.init* σ) *xs* = *snd x*)
    case *True* **with** *snoc.prems snoc.hyps*(*2*) **show** *?thesis*
      by (*simp add*: *spec.singleton.le-conv*)
  next
    case *False* **with** *snoc.prems snoc.hyps*(*2*) **show** *?thesis*
      by (*cases* σ) (*clarsimp simp*: *spec.singleton.le-conv trace.steps′.append*)
  qed
  show *?case*
    by (*rule order.trans*[*OF spec.bind.continueI*[*OF* $*$ $**$, *simplified snoc.hyps*(*2*) *trace.t.collapse*]
                  spec.kleene.fold-starR])
qed

setup ‹*Sign.parent-path*›

setup ‹*Sign.parent-path*›

setup ‹*Sign.parent-path*›

setup ‹*Sign.mandatory-path rel.act*›

lemmas *mono-le* = *spec.kleene.expansive-star*

setup ‹*Sign.parent-path*›

setup ‹*Sign.mandatory-path rel.monomorphic*›

lemma *alt-def*:
  shows *spec.rel.monomorphic r* = $\bigsqcup$(*spec.singleton* ‘ $\{\sigma.$ *trace.steps* $\sigma \subseteq r\}$) (**is** *?lhs* = *?rhs*)
proof(*rule antisym*)

```
  show ?lhs ≤ ?rhs
  proof(induct rule: spec.kleene.star.fixp-induct[case-names adm bot step])
    case (step R)
    have spec.return () ≤ ?rhs
      by (force intro: spec.singleton-le-extI simp: spec.singleton.le-conv
                dest: trace.steps'.simps(5))
    moreover
    have spec.rel.act r ≫ ?rhs ≤ ?rhs
    proof(induct rule: spec.bind-le)
      case incomplete show ?case
        by (rule spec.singleton-le-extI)
           (clarsimp simp: spec.singleton.le-conv;
            metis order.refl empty-subsetI insert-subsetI trace.steps'.empty-conv(1))
    next
      case (continue σ_f σ_g v) then show ?case
        by (fastforce intro!: exI[where x=trace.T (trace.init σ_f) (trace.rest σ_f @ trace.rest σ_g) (trace.term σ_g)]
                  simp: trace.steps'.append spec.singleton.rel.act.le-conv
                  dest: trace.steps.mono[OF iffD1[OF spec.singleton-le-conv], simplified,
                                         simplified trace.steps'.natural'])
    qed
    ultimately show ?case
      by − (strengthen ord-to-strengthen(1)[OF step]; simp)
  qed simp-all
  show ?rhs ≤ ?lhs
    by (simp add: spec.singleton.rel.monomorphic.le-steps)
qed

setup ‹Sign.parent-path›

setup ‹Sign.mandatory-path singleton›

setup ‹Sign.mandatory-path rel›

lemma monomorphic-le-conv[spec.singleton.le-conv]:
  shows ⟪σ⟫ ≤ spec.rel.monomorphic r ⟷ trace.steps σ ⊆ r
by (fastforce simp: spec.rel.monomorphic.alt-def spec.singleton-le-conv trace.steps'.natural'
          dest: trace.steps.mono)

setup ‹Sign.parent-path›

lemma rel-le-conv[spec.singleton.le-conv]:
  shows ⟪σ⟫ ≤ spec.rel r ⟷ trace.steps σ ⊆ r
by (cases σ) (auto simp add: spec.rel-def spec.singleton.le-conv)

setup ‹Sign.parent-path›

interpretation rel: galois.complete-lattice-class spec.steps spec.rel
proof(rule galois.upper-preserves-InfI)
  show mono spec.rel
    by (simp add: monoD monotoneI spec.kleene.monotone-star spec.rel.act.mono spec.rel-def)
  show (⨅x∈X. spec.rel x) ≤ spec.rel (⋂X) for X :: ('a, 'b) steps set
    by (fastforce intro: spec.singleton-le-extI simp: le-INF-iff spec.singleton.le-conv)
qed (simp add: spec.steps-def)

lemma rel-alt-def:
  shows spec.rel r = ⨆(spec.singleton ‘ {σ. trace.steps σ ⊆ r})
by (simp flip: spec.singleton.rel-le-conv)
```

setup ‹*Sign.mandatory-path vmap*›

lemma *unit-rel*:
  shows *spec.vmap* ⟨()⟩ (*spec.rel r*) = *spec.rel r*
by (*simp add*: *spec.rel-def spec.term.all.vmap-unit-absorb*)

setup ‹*Sign.parent-path*›

setup ‹*Sign.mandatory-path rel*›

lemma *monomorphic-conv*: — if the return type is *unit*
  shows *spec.rel r* = *spec.rel.monomorphic r*
by (*simp add*: *spec.rel-def*
      flip: *spec.term.all.closed-conv*[*OF spec.term.closed.kleene.star*[*OF spec.term.closed.rel.act*]])

lemma *monomorphic-act-le*: — *unit* return type
  shows *spec.rel.act r* ≤ *spec.rel r*
by (*simp add*: *spec.rel.monomorphic-conv spec.rel.act.mono-le*)

lemma *empty*:
  shows *spec.rel* {} = (⨆ *v*. *spec.return v*)
by (*simp add*: *spec.rel-def spec.kleene.star-epsilon spec.rel.act.empty spec.term.all.return*)

lemmas *UNIV* = *spec.rel.upper-top*
lemmas *top* = *spec.rel.UNIV*

lemmas *INF* = *spec.rel.upper-INF*
lemmas *Inf* = *spec.rel.upper-Inf*
lemmas *inf* = *spec.rel.upper-inf*

lemmas *Sup-le* = *spec.rel.Sup-upper-le*
lemmas *sup-le* = *spec.rel.sup-upper-le* — Converse does not hold: the RHS allows interleaving of *r* and *s* steps

lemma *reflcl*:
  shows *spec.rel* (*r* ∪ *A* × *Id*) = *spec.rel r*
    and *spec.rel* (*A* × *Id* ∪ *r*) = *spec.rel r*
by (*simp-all add*: *spec.rel-def spec.rel.act-def ac-simps flip*: *Times-Un-distrib1*)

lemma *minus-Id*:
  shows *spec.rel* (*r* − *A* × *Id*) = *spec.rel r*
by (*metis Un-Diff-cancel spec.rel.reflcl*(*2*))

lemma *Id*:
  shows *spec.rel* (*A* × *Id*) = (⨆ *v*. *spec.return v*)
by (*subst spec.rel.minus-Id*[**where** *A*=*A*, *symmetric*]) (*simp add*: *spec.rel.empty*)

lemmas *monotone* = *spec.rel.monotone-upper*
lemmas *mono* = *monotoneD*[*OF spec.rel.monotone*, *of r r'* **for** *r r'*]

lemma *mono-reflcl*:
  assumes *r* ⊆ *s* ∪ *UNIV* × *Id*
  shows *spec.rel r* ≤ *spec.rel s*
by (*metis assms spec.rel.mono spec.rel.reflcl*(*1*))

lemma *unfoldL*:
  shows *spec.rel r* = *spec.rel.act r* ≫ *spec.rel r* (**is** *?lhs* = *?rhs*)
proof(*rule antisym*)
  show *?lhs* ≤ *?rhs*

```
    by (rule order.trans[OF spec.bind.returnL-le spec.bind.mono[OF spec.return.rel.act-le order.refl]])
  show ?rhs ≤ ?lhs
    by (subst (2) spec.rel-def, subst spec.kleene.star-unfoldL)
      (simp add: spec.term.all.sup spec.term.all.bind le-supI1 flip: spec.rel-def)
qed

lemma foldR: — arbitrary interstitial return type
  shows spec.rel r ≫ spec.rel.act r = spec.rel r (is ?lhs = ?rhs)
proof −
  have ?lhs = spec.rel.monomorphic r ≫ spec.rel.act r
    by (subst spec.vmap.unitL) (simp add: spec.vmap.unit-rel spec.rel.monomorphic-conv)
  also have ... = ?rhs
  proof(rule antisym)
    show spec.rel.monomorphic r ≫ spec.rel.act r ≤ ?rhs
      by (simp add: spec.kleene.fold-starR spec.rel.monomorphic-conv)
    show ?rhs ≤spec.rel.monomorphic r ≫ spec.rel.act r
      by (simp add: spec.rel.monomorphic-conv)
        (rule spec.bind.mono[OF order.refl spec.return.rel.act-le, where 'c=unit, simplified])
  qed
  finally show ?thesis .
qed

lemma wind-bind: — arbitrary interstitial return type
  shows spec.rel r ≫ spec.rel r = spec.rel r (is ?lhs = ?rhs)
proof(rule antisym)
  show ?lhs ≤ ?rhs
  proof(induct rule: spec.bind-le)
    case incomplete show ?case
      by (simp add: spec.term.all.rel spec.term.galois)
  next
    case (continue σf σg v) then show ?case
      by (simp add: spec.singleton.rel-le-conv trace.steps'.append)
  qed
  show ?rhs ≤ ?lhs
    by (meson order.trans order.refl spec.bind.mono spec.bind.returnL-le spec.return.rel-le)
qed

lemma wind-bind-leading: — arbitrary interstitial return type
  assumes r' ⊆ r
  shows spec.rel r' ≫ spec.rel r = spec.rel r (is ?lhs = ?rhs)
proof(rule antisym)
  from assms show ?lhs ≤ ?rhs
    by (metis order.refl spec.bind.mono spec.rel.mono spec.rel.wind-bind)
  show ?rhs ≤ ?lhs
    by (meson order.trans spec.eq-iff spec.bind.mono spec.bind.returnL-le spec.return.rel-le)
qed

lemma wind-bind-trailing: — arbitrary interstitial return type
  assumes r' ⊆ r
  shows spec.rel r ≫ spec.rel r' = spec.rel r (is ?lhs = ?rhs)
proof(rule antisym[OF - spec.singleton-le-extI])
  from assms show ?lhs ≤ ?rhs
    by (metis order-refl spec.bind.mono spec.rel.mono spec.rel.wind-bind)
  show ⟪σ⟫ ≤ ?lhs if ⟪σ⟫ ≤ ?rhs for σ
    using that
    by (cases σ)
      (force simp: spec.singleton.le-conv spec.singleton.bind-le-conv spec.singleton.continue-le-conv)
qed
```

Interstitial unit, for unfolding

lemmas *unwind-bind* = *spec.rel.wind-bind*[**where** *'d=unit*, *symmetric*]
lemmas *unwind-bind-leading* = *spec.rel.wind-bind-leading*[**where** *'d=unit*, *symmetric*]
lemmas *unwind-bind-trailing* = *spec.rel.wind-bind-trailing*[**where** *'d=unit*, *symmetric*]

setup ‹*Sign.parent-path*›

setup ‹*Sign.mandatory-path invmap*›

lemma *rel*:
  shows *spec.invmap af sf vf* (*spec.rel r*) = *spec.rel* (*map-prod af* (*map-prod sf sf*) −‘ (*r* ∪ *UNIV* × *Id*))
by (*fastforce intro*: *antisym spec.singleton-le-extI simp*: *spec.singleton.invmap-le-conv spec.singleton.rel-le-conv trace.steps'.map*)

lemma *range*:
  shows *spec.invmap af sf vf P* = *spec.invmap af sf vf* (*P* ⊓ *spec.rel* (*range af* × *range sf* × *range sf*))
by (*rule antisym*[*OF spec.singleton-le-extI*])
   (*auto simp*: *spec.singleton.le-conv trace.steps'.map spec.invmap.mono*)

setup ‹*Sign.parent-path*›

setup ‹*Sign.mandatory-path map*›

lemma *inf-rel*:
  shows *spec.map af sf vf P* ⊓ *spec.rel r*
      = *spec.map af sf vf* (*P* ⊓ *spec.rel* (*map-prod af* (*map-prod sf sf*) −‘ (*r* ∪ *UNIV* × *Id*)))
    and *spec.rel r* ⊓ *spec.map af sf vf P*
      = *spec.map af sf vf* (*spec.rel* (*map-prod af* (*map-prod sf sf*) −‘ (*r* ∪ *UNIV* × *Id*)) ⊓ *P*)
by (*simp-all add*: *spec.invmap.rel spec.map.inf-distr ac-simps*)

setup ‹*Sign.parent-path*›

setup ‹*Sign.mandatory-path action*›

lemma *rel-le*:
  fixes *F* :: (*'v* × *'a* × *'s* × *'s*) *set*
  fixes *r* :: (*'a*, *'s*) *steps*
  assumes ⋀*v a s s'*. (*v*, *a*, *s*, *s'*) ∈ *F* ⟹ (*a*, *s*, *s'*) ∈ *r* ∨ *s* = *s'*
  shows *spec.action F* ≤ *spec.rel r*
unfolding *spec.rel-def*
by (*strengthen ord-to-strengthen*(*2*)[*OF spec.kleene.expansive-star*])
   (*fastforce simp*: *spec.rel.act-def spec.term.all.action intro*: *le-supI1 spec.action.mono dest*: *assms*)

setup ‹*Sign.parent-path*›

setup ‹*Sign.mandatory-path kleene*›

lemma *star-le*:
  assumes *S* ≤ *spec.rel r*
  shows *spec.kleene.star S* ≤ *spec.rel r*
by (*strengthen ord-to-strengthen*(*1*)[*OF assms*])
   (*simp add*: *spec.rel-def spec.kleene.idempotent-star*
       flip: *spec.term.all.closed-conv*[*OF spec.term.closed.kleene.star*[*OF spec.term.closed.rel.act*]])

setup ‹*Sign.parent-path*›

setup ‹*Sign.mandatory-path bind*›

lemma *relL-le*:
  shows $g\ x \leq$ *spec.rel* $r \gg\!= g$
by (*rule order.trans*[*OF spec.bind.returnL-le spec.bind.mono*[*OF spec.return.rel-le order.refl*]])

lemma *relR-le*:
  shows $f \leq f \gg$ *spec.rel r*
by (*rule order.trans*[*OF eq-refl*[*OF spec.bind.returnR*[*symmetric*]]
                    spec.bind.mono[*OF order.refl spec.return.rel-le*]])

lemma *inf-rel*:
  shows $(f \gg\!= g) \sqcap$ *spec.rel r* $=$ (*spec.rel r* $\sqcap f$) $\gg\!=$ ($\lambda x.$ *spec.rel r* $\sqcap\ g\ x$) (**is** *?thesis1*)
    and *spec.rel r* $\sqcap\ (f \gg\!= g) =$ (*spec.rel r* $\sqcap f$) $\gg\!=$ ($\lambda x.$ *spec.rel r* $\sqcap\ g\ x$) (**is** *?lhs = ?rhs*)
proof −
  show *?lhs = ?rhs*
  proof(*rule antisym*[*OF spec.singleton-le-extI*])
    fix σ **assume** *lhs*: $\langle\!|\sigma|\!\rangle \leq$ *?lhs*
    then have $\langle\!|\sigma|\!\rangle \leq f \gg\!= g$ **by** *simp*
    then show $\langle\!|\sigma|\!\rangle \leq$ *?rhs*
    proof(*cases rule*: *spec.singleton.bind-le*)
      case *incomplete* **with** *lhs* **show** *?thesis*
        by (*cases* σ) (*simp add*: *spec.singleton.le-conv spec.bind.incompleteI*)
    next
      case (*continue* $\sigma_f\ \sigma_g\ v_f$) **with** *lhs* **show** *?thesis*
        by (*cases* σ_f) (*simp add*: *spec.singleton.le-conv trace.steps'.append spec.bind.continueI*)
    qed
  next
    show *?rhs* $\leq$ *?lhs*
    proof(*induct rule*: *spec.bind-le*)
      case *incomplete* **show** *?case*
        by (*auto simp*: *spec.term.none.inf spec.term.galois spec.term.all.rel intro*: *le-infI1 le-infI2*)
    next
      case (*continue* $\sigma_f\ \sigma_g\ v$) **then show** *?case*
        by (*cases* σ_f; *cases* σ_g) (*simp add*: *spec.singleton.rel-le-conv spec.bind.continueI trace.steps'.append*)
    qed
  qed
  then show *?thesis1*
    by (*simp add*: *ac-simps*)
qed

lemma *inf-rel-distr-le*:
  shows (*f* $\sqcap$ *spec.rel r*) $\gg\!=$ ($\lambda v.\ g_1\ v \sqcap g_2$) $\leq$ (*f* $\gg\!=\ g_1$) $\sqcap$ (*spec.rel r* $\gg\!=$ (λ-::*unit*. g_2))
by (*rule spec.bind-le*;
    force simp: *trace.split-all spec.singleton.le-conv spec.term.galois spec.term.none.inf*
          spec.term.all.bind spec.term.all.rel
      intro: *le-infI1 le-infI2 spec.bind.continueI*)

setup ‹*Sign.parent-path*›

setup ‹*Sign.mandatory-path singleton*›

lemma *inf-rel*:
  shows $\langle\!|\sigma|\!\rangle \sqcap$ *spec.rel r* $= \bigsqcup$(*spec.singleton* ‘ $\{\sigma'.\ \sigma' \leq \sigma \wedge$ *trace.steps* $\sigma' \subseteq r\}$) (**is** *?lhs = ?rhs*)
    and *spec.rel r* $\sqcap\ \langle\!|\sigma|\!\rangle = \bigsqcup$(*spec.singleton* ‘ $\{\sigma'.\ \sigma' \leq \sigma \wedge$ *trace.steps* $\sigma' \subseteq r\}$) (**is** *?thesis2*)
proof −
  show *?lhs = ?rhs*
  proof(*rule antisym*[*OF spec.singleton-le-extI*])
    show $\langle\!|\sigma'|\!\rangle \leq$ *?rhs* **if** $\langle\!|\sigma'|\!\rangle \leq$ *?lhs* **for** σ'

```
    using that
    by (fastforce simp: spec.singleton.le-conv spec.singleton-le-conv
                  elim: trace.natural.less-eqE[where u=♮σ and u'=σ, simplified]
                  dest: trace.stuttering.equiv.steps)
  show ?rhs ≤ ?lhs
    by (clarsimp simp: spec.singleton.le-conv spec.singleton.mono)
 qed
 then show ?thesis2
  by (rule inf-commute-conv)
qed

setup ‹Sign.parent-path›

setup ‹Sign.mandatory-path action›

lemma inf-rel:
 fixes F :: ('v × 'a × 's × 's) set
 fixes r :: ('a, 's) steps
 assumes ⋀a. refl (r `` {a})
 shows spec.action F ⊓ spec.rel r = spec.action (F ∩ UNIV × r) (is ?lhs = ?rhs)
  and spec.rel r ⊓ spec.action F = spec.action (F ∩ UNIV × r) (is ?thesis1)
proof −
 show ?lhs = ?rhs
 proof(rule antisym[OF spec.singleton-le-extI])
  from reflD[OF assms] show ⟪σ⟫ ≤ ?rhs if ⟪σ⟫ ≤ ?lhs for σ
   using that
   by (auto 0 2 simp: spec.singleton.le-conv spec.singleton.action-le-conv
                split: option.split-asm)
  show ?rhs ≤ ?lhs
   by (rule order.trans[OF spec.action.inf-le inf.mono[OF order.refl spec.action.rel-le]]) simp
 qed
 then show ?thesis1
  by (rule inf-commute-conv)
qed

lemma inf-rel-reflcl:
 shows spec.action F ⊓ spec.rel r = spec.action (F ∩ UNIV × (r ∪ UNIV × Id))
  and spec.rel r ⊓ spec.action F = spec.action (F ∩ UNIV × (r ∪ UNIV × Id))
by (simp-all add: refl-on-def spec.rel.reflcl ac-simps flip: spec.action.inf-rel)

setup ‹Sign.parent-path›

setup ‹Sign.mandatory-path return›

lemma inf-rel:
 shows spec.rel r ⊓ spec.return v = spec.return v
  and spec.return v ⊓ spec.rel r = spec.return v
by (simp-all add: spec.return-def ac-simps spec.action.inf-rel-reflcl
            Sigma-Un-distrib2 Int-Un-distrib Times-Int-Times
      flip: Sigma-Un-distrib2)

setup ‹Sign.parent-path›

setup ‹Sign.mandatory-path kleene.star›

lemma inf-rel:
 shows spec.kleene.star P ⊓ spec.rel r = spec.kleene.star (P ⊓ spec.rel r) (is ?lhs = ?rhs)
proof(rule antisym)
```

show *?lhs* ≤ *?rhs*
by (*induct rule*: *spec.kleene.star.fixp-induct*)
(*simp-all add*: *ac-simps inf-sup-distrib1 spec.bind.inf-rel le-supI1 le-supI2 spec.bind.mono*)
show *?rhs* ≤ *?lhs*
by (*induct rule*: *spec.kleene.star.fixp-induct*)
(*simp-all add*: *ac-simps le-supI2 inf-sup-distrib1*
spec.bind.inf-rel spec.bind.mono spec.return.inf-rel)
qed

setup ‹*Sign.parent-path*›

setup ‹*Sign.mandatory-path steps*›

lemma *simps*[*simp*]:
shows (*a*, *s*, *s*) ∉ *spec.steps P*
by (*simp add*: *spec.steps-def exI*[**where** *x*=*UNIV* × −*Id*]
spec.rel.minus-Id[**where** *r*=*UNIV* **and** *A*=*UNIV*, *simplified*] *spec.rel.UNIV*)

lemma *member-conv*:
shows *x* ∈ *spec.steps P* ⟷ (∃ σ. ⟨σ⟩ ≤ *P* ∧ *x* ∈ *trace.steps* σ)
by (*meson spec.rel.galois spec.singleton.rel-le-conv spec.singleton-le-ext-conv subset-Compl-singleton*)

setup ‹*Sign.mandatory-path term*›

lemma *none*:
shows *spec.steps* (*spec.term.none P*) = *spec.steps P*
by (*metis order.eq-iff spec.rel.galois spec.term.all.rel spec.term.galois*)

lemma *all*:
shows *spec.steps* (*spec.term.all P*) = *spec.steps P*
by (*metis spec.steps.term.none spec.term.none-all*)

setup ‹*Sign.parent-path*›

lemmas *bot* = *spec.rel.lower-bot*

lemmas *monotone* = *spec.rel.monotone-lower*
lemmas *mono* = *monotoneD*[*OF spec.steps.monotone*]

lemmas *Sup* = *spec.rel.lower-Sup*
lemmas *sup* = *spec.rel.lower-sup*
lemmas *Inf-le* = *spec.rel.lower-Inf-le*
lemmas *inf-le* = *spec.rel.lower-inf-le*

lemma *singleton*:
shows *spec.steps* ⟨σ⟩ = *trace.steps* σ
by (*meson subset-antisym order.refl spec.rel.galois spec.singleton.rel-le-conv*)

lemma *idle*:
shows *spec.steps spec.idle* = {}
by (*simp add*: *spec.steps-def spec.idle.rel-le*)

lemma *action*:
shows *spec.steps* (*spec.action F*) = *snd* ‘ *F* − *UNIV* × *Id*
by (*force simp*: *spec.action-def split-def*
spec.steps.Sup spec.steps.sup spec.steps.singleton spec.steps.idle)

lemma *return*:

shows *spec.steps* (*spec.return v*) = {}
by (*simp add*: *spec.return-def spec.steps.action*)

lemma *bind-le*: — see *spec.steps.bind*
shows *spec.steps* (*f* ⤜ *g*) ⊆ *spec.steps f* ∪ (⋃ *v*. *spec.steps* (*g v*))
by (*force simp*: *spec.steps.member-conv spec.singleton.le-conv trace.split-all trace.steps'.append*
elim!: *spec.singleton.bind-le*)

lemma *kleene-star*:
shows *spec.steps* (*spec.kleene.star P*) = *spec.steps P* (**is** *?lhs* = *?rhs*)
proof(*rule antisym*)
show *?lhs* ≤ *?rhs*
proof(*induct rule*: *spec.kleene.star.fixp-induct*[*case-names adm bot step*])
case (*step S*) **then show** *?case*
by (*simp add*: *spec.steps.sup spec.steps.return order.trans*[*OF spec.steps.bind-le*])
qed (*simp-all add*: *spec.steps.bot*)
show *?rhs* ≤ *?lhs*
by (*simp add*: *spec.steps.mono spec.kleene.expansive-star*)
qed

lemma *map*:
shows *spec.steps* (*spec.map af sf vf P*)
= *map-prod af* (*map-prod sf sf*) ‘ *spec.steps P* − *UNIV* × *Id*
by (*rule spec.singleton.exhaust*[*of P*])
(*force simp*: *spec.map.Sup spec.map.singleton spec.steps.Sup spec.steps.singleton trace.steps'.map image-Union*)

lemma *invmap-le*:
shows *spec.steps* (*spec.invmap af sf vf P*)
⊆ *map-prod af* (*map-prod sf sf*) −‘ (*spec.steps* (*P* ⊓ *spec.rel* (*range af* × *range sf* × *range sf*)) ∪ *UNIV* × *Id*) − *UNIV* × *Id*
by (*simp add*: *spec.rel.galois spec.rel.minus-Id*
order.trans[*OF - spec.invmap.mono*[*OF spec.rel.upper-lower-expansive*]]
flip: *vimage-Un spec.invmap.rel*[**where** *vf*=*vf*] *spec.invmap.range*)

setup ‹*Sign.mandatory-path rel*›

lemma *monomorphic*:
fixes *r* :: (′*a*, ′*s*) *steps*
shows *spec.steps* (*spec.rel.monomorphic r*) = *r* − *UNIV* × *Id* (**is** *?lhs* = *?rhs*)
proof(*rule antisym*)
show *?lhs* ⊆ *?rhs*
by (*simp add*: *spec.rel.galois spec.rel.minus-Id flip*: *spec.rel.monomorphic-conv*)
show *?rhs* ⊆ *?lhs*
by (*force simp*: *spec.rel.monomorphic.alt-def spec.term.all.Sup spec.term.all.singleton*
spec.steps.Sup spec.steps.singleton
dest: *spec*[**where** *x*=*trace.T s* [(*a*, *s′*)] *None* **for** *a s s′*]
split: *if-splits*)
qed

setup ‹*Sign.parent-path*›

lemma *rel*:
fixes *r* :: (′*a*, ′*s*) *steps*
shows *spec.steps* (*spec.rel r*) = *r* − *UNIV* × *Id*
by (*simp add*: *spec.rel-def spec.steps.term.all spec.steps.rel.monomorphic*)

lemma *top*:
shows *spec.steps* ⊤ = *UNIV* × − *Id*

using *spec.steps.rel*[**where** *r*=*UNIV*] **by** (*simp add*: *spec.rel.UNIV*)

setup ‹*Sign.parent-path*›

setup ‹*Sign.parent-path*›

8.11 Sequential assertions

We specify sequential behavior with preconditions and postconditions.

8.11.1 Preconditions

setup ‹*Sign.mandatory-path spec*›

definition *pre* :: *'s pred* ⇒ (*'a*, *'s*, *'v*) *spec* **where**
pre P = ⨆(*spec.singleton* ‘ {*σ*. *P* (*trace.init σ*)})

setup ‹*Sign.mandatory-path singleton*›

lemma *pre-le-conv*[*spec.singleton.le-conv*]:
shows ⟪*σ*⟫ ≤ *spec.pre P* ⟷ *P* (*trace.init σ*)
by (*auto simp add*: *spec.pre-def spec.singleton-le-conv trace.natural-def elim*: *trace.less-eqE*)

lemma *inf-pre*:
shows *spec.pre P* ⊓ ⟪*σ*⟫ = (*if P* (*trace.init σ*) *then* ⟪*σ*⟫ *else* ⊥) (**is** *?thesis1*)
and ⟪*σ*⟫ ⊓ *spec.pre P* = (*if P* (*trace.init σ*) *then* ⟪*σ*⟫ *else* ⊥) (**is** *?thesis2*)
proof −
show *?thesis1*
by (*cases σ*; *rule spec.singleton.antisym*)
(*auto simp*: *spec.singleton.le-conv spec.singleton-le-conv spec.singleton.not-bot trace.natural.trace-conv*
split: *if-split-asm*
elim: *trace.less-eqE*)
then show *?thesis2*
by (*rule inf-commute-conv*)
qed

setup ‹*Sign.parent-path*›

setup ‹*Sign.mandatory-path idle*›

lemma *pre-le-conv*[*spec.idle-le*]:
shows *spec.idle* ≤ (*spec.pre P* :: (*'a*, *'s*, *'v*) *spec*) ⟷ *P* = ⊤ (**is** *?lhs* ⟷ *?rhs*)
proof(*rule iffI*)
show *?lhs* ⟹ *?rhs*
by (*rule ccontr*)
(*simp add*: *fun-eq-iff spec.pre-def spec.idle-def trace.split-Ex*
spec.singleton-le-conv trace.less-eq-None trace.natural.simps)
show *?rhs* ⟹ *?lhs*
by (*rule spec.singleton-le-extI*) (*simp add*: *spec.singleton.le-conv*)
qed

setup ‹*Sign.parent-path*›

setup ‹*Sign.mandatory-path term*›

setup ‹*Sign.mandatory-path all*›

lemma *pre*:

shows *spec.term.all* (*spec.pre P*) = *spec.pre P*
by (*rule spec.singleton.antisym*; *simp add*: *spec.singleton.le-conv*)

setup ‹*Sign.parent-path*›

setup ‹*Sign.mandatory-path closed*›

lemma *pre*:
shows *spec.pre P* ∈ *spec.term.closed* -
by (*rule spec.term.closed-upper*[*of spec.pre P*, *simplified spec.term.all.pre*])

lemma *none-inf-pre*:
fixes *P* :: *'s pred*
fixes *Q* :: (*'a*, *'s*, *'v*) *spec*
shows *spec.term.none* (*Q* ⊓ *spec.pre P*) = (*spec.term.none Q* ⊓ *spec.pre P* :: (*'a*, *'s*, *'w*) *spec*) (**is** *?lhs* = *?rhs*)
and *spec.term.none* (*spec.pre P* ⊓ *Q*) = (*spec.pre P* ⊓ *spec.term.none Q* :: (*'a*, *'s*, *'w*) *spec*) (**is** *?thesis2*)
proof −
show *?lhs* = *?rhs*
apply (*subst spec.term.none-all*[*symmetric*])
apply (*subst spec.term.all.inf*)
apply (*subst spec.term.closed.none-inf-monomorphic*(*2*)[*symmetric*])
apply (*simp-all add*: *spec.term.all.pre spec.term.closed.pre*)
done
then show *?thesis2*
by (*simp add*: *ac-simps*)
qed

setup ‹*Sign.parent-path*›

setup ‹*Sign.parent-path*›

setup ‹*Sign.mandatory-path pre*›

lemma *bot*[*iff*]:
shows *spec.pre* ⟨*False*⟩ = ⊥
and *spec.pre* ⊥ = ⊥
by (*simp-all add*: *spec.pre-def*)

lemma *top*[*iff*]:
shows *spec.pre* ⟨*True*⟩ = ⊤
and *spec.pre* ⊤ = ⊤
by (*simp-all add*: *spec.pre-def full-SetCompr-eq spec.singleton.top*)

lemma *top-conv*:
shows *spec.pre P* = (⊤ :: (*'a*, *'s*, *'v*) *spec*) ⟷ *P* = ⊤
by (*auto intro*: *iffD1*[*OF spec.idle.pre-le-conv*[**where** *'a*=*'a* **and** *'s*=*'s* **and** *'v*=*'v*]])

lemma *K*:
shows *spec.pre* ⟨*P*⟩ = (*if P then* ⊤ *else* ⊥)
by (*simp add*: *spec.pre-def full-SetCompr-eq spec.singleton.top*)

lemma *monotone*:
shows *mono spec.pre*
by (*fastforce simp*: *spec.pre-def intro*: *monoI*)

lemmas *strengthen*[*strg*] = *st-monotone*[*OF spec.pre.monotone*]
lemmas *mono* = *monotoneD*[*OF spec.pre.monotone*]

lemma *SUP*:
  shows *spec.pre* $(\bigsqcup x{\in}X.\ P\ x) = (\bigsqcup x{\in}X.\ spec.pre\ (P\ x))$
by (*auto simp*: *spec.pre-def spec.eq-iff intro*: *rev-SUPI*)

lemma *Sup*:
  shows *spec.pre* $(\bigsqcup X) = (\bigsqcup x{\in}X.\ spec.pre\ x)$
by (*metis image-ident image-image spec.pre.SUP*)

lemma *Bex*:
  shows *spec.pre* $(\lambda s.\ \exists\, x{\in}X.\ P\ x\ s) = (\bigsqcup x{\in}X.\ spec.pre\ (P\ x))$
by (*simp add*: *Sup-fun-def flip*: *spec.pre.SUP*)

lemma *Ex*:
  shows *spec.pre* $(\lambda s.\ \exists\, x.\ P\ x\ s) = (\bigsqcup x.\ spec.pre\ (P\ x))$
by (*simp add*: *Sup-fun-def flip*: *spec.pre.SUP*)

lemma
  shows *disj*: *spec.pre* $(P \vee Q) = spec.pre\ P \sqcup spec.pre\ Q$
    and *sup*: *spec.pre* $(P \sqcup Q) = spec.pre\ P \sqcup spec.pre\ Q$
using *spec.pre.Sup*[**where** $X{=}\{P,\ Q\}$] **by** (*simp-all add*: *sup-fun-def*)

lemma *INF*:
  shows *spec.pre* $(\bigsqcap x{\in}X.\ P\ x) = (\bigsqcap x{\in}X.\ spec.pre\ (P\ x))$
by (*auto simp*: *spec.eq-iff spec.singleton.pre-le-conv le-INF-iff intro*: *spec.singleton-le-extI*)

lemma *Inf*:
  shows *spec.pre* $(\bigsqcap X) = (\bigsqcap x{\in}X.\ spec.pre\ x)$
by (*metis image-ident image-image spec.pre.INF*)

lemma *Ball*:
  shows *spec.pre* $(\lambda s.\ \forall\, x{\in}X.\ P\ x\ s) = (\bigsqcap x{\in}X.\ spec.pre\ (P\ x))$
by (*simp add*: *Inf-fun-def flip*: *spec.pre.INF*)

lemma *All*:
  shows *spec.pre* $(\lambda s.\ \forall\, x.\ P\ x\ s) = (\bigsqcap x.\ spec.pre\ (P\ x))$
by (*simp add*: *Inf-fun-def flip*: *spec.pre.INF*)

lemma *inf*:
  shows *conj*: *spec.pre* $(P \wedge Q) = spec.pre\ P \sqcap spec.pre\ Q$
    and *spec.pre* $(P \sqcap Q) = spec.pre\ P \sqcap spec.pre\ Q$
using *spec.pre.Inf*[**where** $X{=}\{P,\ Q\}$] **by** (*simp-all add*: *inf-fun-def*)

lemma *inf-action-le*: — Converse does not hold
  shows *spec.pre* $P \sqcap$ *spec.action* $F \le$ *spec.action* $(UNIV \times UNIV \times Collect\ P \times UNIV \cap F)$ (**is** *?lhs* $\le$ *?rhs*)
    and *spec.action* $F \sqcap$ *spec.pre* $P \le$ *spec.action* $(F \cap UNIV \times UNIV \times Collect\ P \times UNIV)$ (**is** *?thesis2*)
proof −
  show *?lhs* $\le$ *?rhs*
  proof(*rule spec.singleton-le-extI*)
    show $\langle\!|\sigma|\!\rangle \le$ *?rhs* **if** $\langle\!|\sigma|\!\rangle \le$ *?lhs* **for** σ
      using *that*[*simplified*, *unfolded spec.singleton.action-le-conv spec.singleton.le-conv*]
      by (*cases* σ;
          safe; *clarsimp simp*: *trace.steps'.step-conv spec.action.idleI spec.action.stutter-stepsI*
                  split: *option.split-asm*;
          subst spec.singleton.Cons; *blast intro*: *spec.action.stepI*)
  qed
  then show *?thesis2*
    by (*simp add*: *ac-simps*)
qed

setup ‹*Sign.parent-path*›

setup ‹*Sign.mandatory-path invmap*›

lemma *pre*:
  shows *spec.invmap af sf vf* (*spec.pre P*) = *spec.pre* (λ*s*. *P* (*sf s*))
by (*rule spec.singleton.antisym*) (*simp-all add*: *spec.singleton.le-conv*)

setup ‹*Sign.parent-path*›

setup ‹*Sign.mandatory-path bind*›

lemma *inf-pre*:
  shows *spec.pre P* ⊓ (*f* ⤜ *g*) = (*spec.pre P* ⊓ *f*) ⤜ *g* (**is** *?lhs* = *?rhs*)
    and (*f* ⤜ *g*) ⊓ *spec.pre P* = (*f* ⊓ *spec.pre P*) ⤜ *g* (**is** *?thesis1*)
proof −
  show *?lhs* = *?rhs*
  proof(*rule antisym*)
    show *?lhs* ≤ *?rhs*
      by (*fastforce simp*: *spec.bind-def spec.continue-def inf-sup-distrib1 inf-Sup spec.singleton.inf-pre*
          simp flip: *spec.term.closed.none-inf-pre spec.singleton.pre-le-conv*)
    show *?rhs* ≤ *?lhs*
      by (*fastforce simp*: *spec.bind-def spec.continue-def inf-sup-distrib1 spec.singleton.pre-le-conv*
          simp flip: *spec.term.closed.none-inf-pre*)
  qed
  then show *?thesis1*
    by (*simp add*: *ac-simps*)
qed

setup ‹*Sign.parent-path*›

setup ‹*Sign.mandatory-path steps*›

lemma *pre*:
  assumes *P* s_0
  shows *spec.steps* (*spec.pre P* :: ('*a*, '*s*, '*v*) *spec*) = *UNIV* × − *Id*
proof −
  have (*a*, *s*, *s'*) ∈ *spec.steps* (*spec.pre P*) **if** *s* ≠ *s'* **for** *a* :: '*a* **and** *s s'*
    using *assms that*
    by (*simp add*: *spec.singleton.le-conv spec.steps.member-conv trace.steps'.Cons-eq-if*
        exI[**where** *x*=*trace.T* s_0 [(*undefined*, *s*), (*a*, *s'*)] *None*])
  then show *?thesis*
    by *auto*
qed

setup ‹*Sign.parent-path*›

setup ‹*Sign.parent-path*›

8.11.2 Postconditions

Unlike *spec.pre spec.post* can be expressed in terms of other constants.

setup ‹*Sign.mandatory-path spec*›

setup ‹*Sign.mandatory-path post*›

definition *act* :: ('*v* ⇒ '*s pred*) ⇒ ('*v* × '*a* × '*s* × '*s*) *set* **where**

act $Q = \{(v,\ a,\ s,\ s')\ |v\ a\ s\ s'.\ Q\ v\ s'\}$

setup ‹*Sign.mandatory-path act*›

lemma *simps*[*simp*]:
  shows *spec.post.act* $\langle\langle False\rangle\rangle = \{\}$
    and *spec.post.act* $\langle\bot\rangle = \{\}$
    and *spec.post.act* $\bot = \{\}$
    and *spec.post.act* $\langle\langle True\rangle\rangle = UNIV$
    and *spec.post.act* $\langle\top\rangle = UNIV$
    and *spec.post.act* $\top = UNIV$
    and *spec.post.act* $(Q \sqcup Q') =$ *spec.post.act* $Q \cup$ *spec.post.act* Q'
    and *spec.post.act* $(\bigsqcup X) = (\bigcup x{\in}X.$ *spec.post.act* $x)$
    and *spec.post.act* $(\lambda v.\ \bigsqcup x{\in}Y.\ R\ x\ v) = (\bigcup x{\in}Y.$ *spec.post.act* $(R\ x))$
by (*auto 0 2 simp*: *spec.post.act-def*)

lemma *monotone*:
  shows *mono spec.post.act*
proof(*rule monotoneI*)
  show *spec.post.act* $Q \le$ *spec.post.act* Q' **if** $Q \le Q'$ **for** $Q\ Q' :: 'v \Rightarrow 's\ pred$
    using *that* **unfolding** *spec.post.act-def* **by** *blast*
qed

lemmas *strengthen*[*strg*] = *st-monotone*[*OF spec.post.act.monotone*]
lemmas *mono* = *monotoneD*[*OF spec.post.act.monotone*]

setup ‹*Sign.parent-path*›

setup ‹*Sign.parent-path*›

definition *post* :: $('v \Rightarrow 's\ pred) \Rightarrow ('a,\ 's,\ 'v)$ *spec* **where**
  post $Q = \top \gg\!= (\lambda$-::*unit*. *spec.action* (*spec.post.act* $Q))$

setup ‹*Sign.mandatory-path singleton*›

lemma *post-le-conv*[*spec.singleton.le-conv*]:
  fixes $Q :: 'v \Rightarrow 's\ pred$
  shows $\langle\!|\sigma|\!\rangle \le$ *spec.post* Q
    $\longleftrightarrow$ (*case trace.term* σ *of None* $\Rightarrow$ *True* | *Some* $v \Rightarrow Q\ v$ (*trace.final* σ)) (**is** *?lhs* $\longleftrightarrow$ *?rhs*)
proof(*rule iffI*)
  show *?lhs* $\Longrightarrow$ *?rhs*
    by (*fastforce simp*: *spec.post-def spec.post.act-def*
                  spec.singleton.le-conv spec.singleton.action-le-conv trace.steps'.step-conv
                split: *option.split*
                 elim: *spec.singleton.bind-le*)
  show *?rhs* $\Longrightarrow$ *?lhs*
    by (*cases* σ)
      (*simp add*: *spec.post-def spec.post.act-def spec.action.stutterI*
                spec.bind.incompleteI spec.bind.continueI[**where** *ys*=[], *simplified*]
          split: *option.splits*)
qed

setup ‹*Sign.parent-path*›

setup ‹*Sign.mandatory-path idle*›

lemma *post-le*[*spec.idle-le*]:
  shows *spec.idle* $\le$ *spec.post* Q

by (*rule spec.singleton-le-extI*) (*simp add*: *spec.singleton.le-conv*)

setup ‹*Sign.parent-path*›

setup ‹*Sign.mandatory-path term*›

setup ‹*Sign.mandatory-path none*›

lemma *post-le*:
  shows *spec.term.none P* ≤ *spec.post Q*
by (*rule spec.singleton-le-extI*) (*simp add*: *spec.singleton.le-conv*)

lemma *post*:
  shows *spec.term.none* (*spec.post Q* :: (*'a*, *'s*, *'v*) *spec*)
      = *spec.term.none* (⊤ :: (*'a*, *'s*, *unit*) *spec*)
by (*metis spec.eq-iff spec.term.galois spec.term.none.post-le spec.term.none-all top-greatest*)

setup ‹*Sign.parent-path*›

setup ‹*Sign.mandatory-path all*›

lemma *post*:
  shows *spec.term.all* (*spec.post Q*) = ⊤
by (*metis spec.term.all-none spec.term.none.post spec.term.upper-top*)

setup ‹*Sign.parent-path*›

setup ‹*Sign.parent-path*›

setup ‹*Sign.mandatory-path post*›

lemma *bot*[*iff*]:
  shows *spec.post* ⟨⟨*False*⟩⟩ = *spec.term.none* (⊤ :: (-, -, *unit*) *spec*)
    and *spec.post* ⟨⊥⟩ = *spec.term.none* (⊤ :: (-, -, *unit*) *spec*)
    and *spec.post* ⊥ = *spec.term.none* (⊤ :: (-, -, *unit*) *spec*)
by (*simp-all add*: *spec.post-def spec.action.empty spec.bind.idleR spec.bind.botR*)

lemma *monotone*:
  shows *mono spec.post*
by (*simp add*: *spec.post-def monoI spec.action.mono spec.bind.mono spec.post.act.mono*)

lemmas *strengthen*[*strg*] = *st-monotone*[*OF spec.post.monotone*]
lemmas *mono* = *monotoneD*[*OF spec.post.monotone*]

lemma *SUP-not-empty*:
  fixes *X* :: *'a set*
  fixes *Q* :: *'a* ⇒ *'v* ⇒ *'s pred*
  assumes *X* ≠ {}
  shows *spec.post* (λ*v*. ⨆*x*∈*X*. *Q x v*) = (⨆*x*∈*X*. *spec.post* (*Q x*))
by (*simp add*: *assms spec.post-def flip*: *spec.bind.SUPR-not-empty spec.action.SUP-not-empty*)

lemma *disj*:
  shows *spec.post* (*Q* ⊔ *Q'*) = *spec.post Q* ⊔ *spec.post Q'*
    and *spec.post* (λ*rv*. *Q rv* ⊔ *Q' rv*) = *spec.post Q* ⊔ *spec.post Q'*
    and *spec.post* (λ*rv*. *Q rv* ∨ *Q' rv*) = *spec.post Q* ⊔ *spec.post Q'*
using *spec.post.SUP-not-empty*[**where** *X*=*UNIV* **and** *Q*=λ*x*. *if x then Q' else Q*]
by (*simp-all add*: *UNIV-bool sup-fun-def*)

lemma *INF*:
  shows *spec.post* ($\bigsqcap x{\in}X.\ Q\ x$) = ($\bigsqcap x{\in}X.$ *spec.post* ($Q\ x$))
    and *spec.post* ($\lambda v.\ \bigsqcap x{\in}X.\ Q\ x\ v$) = ($\bigsqcap x{\in}X.$ *spec.post* ($Q\ x$))
    and *spec.post* ($\lambda v\ s.\ \bigsqcap x{\in}X.\ Q\ x\ v\ s$) = ($\bigsqcap x{\in}X.$ *spec.post* ($Q\ x$))
by (*fastforce intro*: *antisym spec.singleton-le-extI simp*: *spec.singleton.le-conv le-Inf-iff split*: *option.split*)+

lemma *Inf*:
  shows *spec.post* ($\bigsqcap X$) = ($\bigsqcap x{\in}X.$ *spec.post* x)
by (*metis image-ident image-image spec.post.INF*(*1*))

lemma *Ball*:
  shows *spec.post* ($\lambda v\ s.\ \forall x{\in}X.\ Q\ x\ v\ s$) = ($\bigsqcap x{\in}X.$ *spec.post* ($Q\ x$))
by (*simp add*: *Inf-fun-def flip*: *spec.post.INF*)

lemma *All*:
  shows *spec.post* ($\lambda v\ s.\ \forall x.\ Q\ x\ v\ s$) = ($\bigsqcap x.$ *spec.post* ($Q\ x$))
by (*simp add*: *Inf-fun-def flip*: *spec.post.INF*)

lemma *inf*:
  shows *spec.post* ($Q \sqcap Q'$) = *spec.post* $Q \sqcap$ *spec.post* Q'
    and *spec.post* ($\lambda rv.\ Q\ rv \sqcap Q'\ rv$) = *spec.post* $Q \sqcap$ *spec.post* Q'
    and *conj*: *spec.post* ($\lambda rv.\ Q\ rv \wedge Q'\ rv$) = *spec.post* $Q \sqcap$ *spec.post* Q'
by (*simp-all add*: *inf-fun-def flip*: *spec.post.INF*[**where** X=*UNIV* **and** Q=$\lambda x.$ *if* x *then* Q' *else* Q, *simplified UNIV-bool*, *simplified*])

lemma *top*[*iff*]:
  shows *spec.post* $\langle\langle True\rangle\rangle = \top$
    and *spec.post* $\langle\top\rangle = \top$
    and *spec.post* $\top = \top$
by (*simp-all add*: *top-fun-def flip*: *spec.post.INF*[**where** X={}, *simplified*])

lemma *top-conv*:
  shows *spec.post* Q = ($\top$ :: ('*a*, '*s*, '*v*) *spec*) $\longleftrightarrow$ $Q = \top$
by (*fastforce simp*: *spec.singleton.post-le-conv*
        dest: *arg-cong*[**where** f=$\lambda x.\ \forall\sigma.\ \langle\sigma\rangle \le x$] *spec*[**where** x=*trace.T s* [] (*Some v*) **for** *s v*])

lemma *K*:
  shows *spec.post* (λ- -. Q) = (*if* Q *then* $\top$ *else* $\top \gg\!=$ (λ-::*unit*. $\bot$))
by (*auto simp flip*: *spec.bind.botR bot-fun-def*)

setup ‹*Sign.parent-path*›

lemma *bind-post-pre*:
  shows $f \sqcap$ *spec.post* $Q \gg\!= g = f \gg\!=$ ($\lambda v.\ g\ v \sqcap$ *spec.pre* ($Q\ v$)) (**is** *?lhs* = *?rhs*)
    and *spec.post* $Q \sqcap f \gg\!= g = f \gg\!=$ ($\lambda v.$ *spec.pre* ($Q\ v$) $\sqcap\ g\ v$) (**is** *?thesis1*)
proof −
  show *?lhs* = *?rhs*
  proof(*rule spec.singleton.antisym*)
    show $\langle\sigma\rangle \le$ *?rhs* **if** $\langle\sigma\rangle \le$ *?lhs* **for** σ
      using *that*
      by (*induct rule*: *spec.singleton.bind-le*)
        (*cases* σ; *force simp*: *trace.split-all spec.singleton.le-conv*
                intro: *spec.bind.incompleteI spec.bind.continueI*)+
    show $\langle\sigma\rangle \le$ *?lhs* **if** $\langle\sigma\rangle \le$ *?rhs* **for** σ
      using *that*
      by (*induct rule*: *spec.singleton.bind-le*)
        (*cases* σ; *force simp*: *trace.split-all spec.singleton.le-conv*
                intro: *spec.bind.incompleteI spec.bind.continueI*)+

qed
then show *?thesis1*
by (*simp add*: *ac-simps*)
qed

setup ‹*Sign.mandatory-path invmap*›

lemma *post*:
shows *spec.invmap af sf vf* (*spec.post Q*) = *spec.post* (λ*v s*. *Q* (*vf v*) (*sf s*))
by (*rule antisym*[*OF spec.singleton-le-extI spec.singleton-le-extI*])
(*simp-all add*: *spec.singleton.invmap-le-conv spec.singleton.post-le-conv trace.final'.map split*: *option.split-asm*)

setup ‹*Sign.parent-path*›

setup ‹*Sign.mandatory-path action*›

lemma *post-le-conv*:
shows *spec.action F* ≤ *spec.post Q* ⟷ (∀ *v a s s'*. (*v*, *a*, *s*, *s'*) ∈ *F* ⟶ *Q v s'*)
by (*fastforce simp*: *spec.action-def split-def spec.singleton.le-conv spec.idle.post-le*)

setup ‹*Sign.parent-path*›

setup ‹*Sign.mandatory-path bind*›

lemma *post-le*:
assumes ⋀*v*. *g v* ≤ *spec.post Q*
shows *f* ⤜ *g* ≤ *spec.post Q*
by (*strengthen ord-to-strengthen*(*1*)[*OF assms*])
(*simp add*: *spec.post-def spec.bind.mono flip*: *spec.bind.bind*)

lemma *inf-post*:
shows (*f* ⤜ *g*) ⊓ *spec.post Q* = *f* ⤜ (λ*v*. *g v* ⊓ *spec.post Q*) (**is** *?lhs* = *?rhs*)
and *spec.post Q* ⊓ (*f* ⤜ *g*) = *f* ⤜ (λ*v*. *spec.post Q* ⊓ *g v*) (**is** *?thesis2*)
proof −
show *?lhs* = *?rhs*
proof(*rule antisym*[*OF spec.singleton-le-extI*])
fix σ
assume *lhs*: ⟪σ⟫ ≤ *?lhs*
from *lhs*[*simplified*, *THEN conjunct1*] *lhs*[*simplified*, *THEN conjunct2*] **show** ⟪σ⟫ ≤ *?rhs*
proof(*induct rule*: *spec.singleton.bind-le*)
case (*continue* σ_f σ_g v_f) **then show** *?case*
by (*cases* σ_f)
(*force intro*: *spec.bind.continueI simp*: *spec.singleton.le-conv split*: *option.split*)
qed (*simp add*: *spec.singleton.bind-le-conv*)
qed (*simp add*: *spec.bind.mono spec.bind.post-le*)
then show *?thesis2*
by (*simp add*: *ac-simps*)
qed

lemma *mono-stronger*:
assumes *f*: *f* ≤ *f'* ⊓ *spec.post Q*
assumes *g*: ⋀*v*. *g v* ⊓ *spec.pre* (*Q v*) ≤ *g' v*
shows *spec.bind f g* ≤ *spec.bind f' g'*
by (*strengthen ord-to-strengthen*(*1*)[*OF f*]) (*simp add*: *g spec.bind.mono spec.bind-post-pre*)

setup ‹*Sign.parent-path*›

setup ‹*Sign.parent-path*›

8.11.3 Strongest postconditions

setup ‹*Sign.mandatory-path spec.post*›

definition *strongest* :: ('*a*, '*s*, '*v*) *spec* ⇒ '*v* ⇒ '*s pred* **where**
strongest P = ⨅{*Q*. *P* ≤ *spec.post Q*}

interpretation *strongest*: *galois.complete-lattice-class spec.post.strongest spec.post*
by (*simp add*: *spec.post.strongest-def galois.upper-preserves-InfI spec.post.Inf spec.post.monotone*)

lemma *strongest-alt-def*:
shows *spec.post.strongest P* = (λ*v s*. ∃ σ. ⟪σ⟫ ≤ *P* ∧ *trace.term* σ = *Some v* ∧ *trace.final* σ = *s*) (**is** *?lhs* = *?rhs*)
proof(*rule antisym*)
show *?lhs* ≤ *?rhs*
by (*rule spec.singleton.exhaust*[*of P*])
(*fastforce simp*: *spec.post.strongest-def spec.singleton.le-conv*
dest: *spec*[**where** *x*=λ*v s*. ∃ σ∈*X*. *trace.term* σ = *Some v* ∧ *trace.final* σ = *s* **for** *X*]
split: *option.split*)
show *?rhs* ≤ *?lhs*
by (*fastforce simp*: *spec.post.strongest-def spec.singleton.le-conv dest*: *order.trans*)
qed

setup ‹*Sign.mandatory-path strongest*›

lemma *singleton*:
shows *spec.post.strongest* ⟪σ⟫
= (λ*v s*. *case trace.term* σ *of None* ⇒ *False* | *Some v'* ⇒ *v'* = *v* ∧ *trace.final* σ = *s*)
by (*auto simp*: *spec.post.strongest-alt-def fun-eq-iff trace.split-all*
cong: *trace.final'.natural'-cong*
split: *option.split*)

lemmas *monotone* = *spec.post.strongest.monotone-lower*
lemmas *mono* = *monoD*[*OF spec.post.strongest.monotone*]
lemmas *Sup* = *spec.post.strongest.lower-Sup*
lemmas *sup* = *spec.post.strongest.lower-sup*

lemma *top*[*iff*]:
shows *spec.post.strongest* ⊤ = ⊤
by (*simp add*: *spec.post.strongest-def spec.post.top-conv top.extremum-unique*)

lemma *action*:
shows *spec.post.strongest* (*spec.action F*) = (λ*v s'*. ∃ *a s*. (*v*, *a*, *s*, *s'*) ∈ *F*)
by (*simp add*: *spec.post.strongest-def spec.action.post-le-conv*) *fast*

lemma *return*:
shows *spec.post.strongest* (*spec.return v*) = (λ*v'* *s*. *v'* = *v*)
by (*simp add*: *spec.return-def spec.post.strongest.action*)

setup ‹*Sign.mandatory-path term*›

lemma *none*:
shows *spec.post.strongest* (*spec.term.none P*) = ⊥
by (*clarsimp simp*: *spec.term.none-def spec.post.strongest.Sup spec.post.strongest.singleton fun-eq-iff*)

lemma *all*:
assumes *spec.idle* ≤ *P*
shows *spec.post.strongest* (*spec.term.all P*) = ⊤

by (*rule top-le*[*OF order.trans*[*OF - spec.post.strongest.mono*[*OF spec.term.all.mono*[*OF assms*]]]])
(*simp add*: *spec.term.all.idle spec.post.strongest.Sup spec.post.strongest.return Sup-fun-def top-fun-def*)

lemma *closed*:
assumes *spec.idle* $\leq P$
assumes $P \in$ *spec.term.closed -*
shows *spec.post.strongest* $P = \top$
by (*metis spec.post.strongest.term.all*[*OF assms*(*1*)] *spec.term.all.closed-conv*[*OF assms*(*2*)])

setup ‹*Sign.parent-path*›

lemma *bind*:
shows *spec.post.strongest* $(f \gg\!= g)$
$=$ *spec.post.strongest* $(\bigsqcup v.$ *spec.pre* $($*spec.post.strongest* $f\ v) \sqcap g\ v)$ (**is** *?lhs* $=$ *?rhs*)
proof(*rule antisym*)
show *?lhs* $\leq$ *?rhs*
by (*auto 0 4 simp*: *spec.post.strongest-alt-def spec.singleton.le-conv*
elim!: *spec.singleton.bind-le*)
show *?rhs* $\leq$ *?lhs*
by (*force simp*: *spec.post.strongest-alt-def spec.singleton.le-conv trace.split-all*
dest: *spec.bind.continueI*)
qed

lemma *rel*:
shows *spec.post.strongest* $($*spec.rel* $r) = \top$
by (*simp add*: *spec.rel-def spec.post.strongest.term.all spec.idle.kleene.star-le*)

lemma *pre*:
shows *spec.post.strongest* $($*spec.pre* $P) = (\lambda v\ s'.\ \exists s.\ P\ s)$
by (*auto simp*: *spec.post.strongest-alt-def spec.singleton.le-conv trace.split-Ex fun-eq-iff*
intro!: *exI*[**where** x=[(*undefined*, s)] **for** s])

lemma *post*:
shows *spec.post.strongest* $($*spec.post* $Q) = Q$
by (*auto simp*: *spec.post.strongest-alt-def spec.singleton.le-conv trace.split-Ex fun-eq-iff*
intro!: *exI*[**where** x=[]])

setup ‹*Sign.parent-path*›

setup ‹*Sign.parent-path*›

8.12 Initial steps

The initial transition of a process.

setup ‹*Sign.mandatory-path spec*›

definition *initial-steps* :: $('a, 's, 'v)$ *spec* $\Rightarrow$ $('a, 's)$ *steps* **where**
initial-steps $P = \{(a, s, s').\ \langle s, [(a, s')], None\rangle \leq P\}$

setup ‹*Sign.mandatory-path initial-steps*›

lemma *steps-le*:
shows *spec.initial-steps* $P \subseteq$ *spec.steps* $P \cup$ *UNIV* $\times$ *Id*
by (*fastforce simp*: *spec.initial-steps-def spec.steps.member-conv split*: *if-splits*)

lemma *galois*:
shows $r \subseteq$ *spec.initial-steps* $P \wedge$ *spec.idle* $\leq P \longleftrightarrow$ *spec.action* $(\{()\} \times r) \gg\!= \bot \leq P$ (**is** *?lhs* $\longleftrightarrow$ *?rhs*)
proof(*rule iffI*)

```
  show ?lhs ⟹ ?rhs
    by (auto simp: spec.action-def spec.initial-steps-def spec.bind.SupL spec.bind.supL
                   spec.bind.singletonL spec.singleton.not-bot spec.term.none.singleton)
  show ?rhs ⟹ ?lhs
    by (auto simp: spec.initial-steps-def spec.idle.action-le spec.idle.bind-le-conv
          elim!: order.trans[rotated]
          intro: spec.action.stepI spec.bind.incompleteI)
qed

lemma bot:
  shows spec.initial-steps ⊥ = {}
by (auto simp: spec.initial-steps-def spec.singleton.not-bot)

lemma top:
  shows spec.initial-steps ⊤ = UNIV
by (auto simp: spec.initial-steps-def spec.singleton.not-bot)

lemma monotone:
  shows mono spec.initial-steps
by (force intro: monoI simp: spec.initial-steps-def)

lemmas strengthen[strg] = st-monotone[OF spec.initial-steps.monotone]
lemmas mono = monotoneD[OF spec.initial-steps.monotone]

lemma Sup:
  shows spec.initial-steps (⨆ X) = ⋃ (spec.initial-steps ‘ X)
by (auto simp: spec.initial-steps-def)

lemma Inf:
  shows spec.initial-steps (⨅ X) = ⋂ (spec.initial-steps ‘ X)
by (auto simp: spec.initial-steps-def le-Inf-iff)

lemma idle:
  shows spec.initial-steps spec.idle = UNIV × Id
by (auto simp: spec.initial-steps-def spec.singleton.le-conv)

lemma action:
  shows spec.initial-steps (spec.action F) = snd ‘ F ∪ UNIV × Id
by (auto simp: spec.initial-steps-def spec.singleton.action-le-conv trace.steps′.step-conv)

lemma return:
  shows spec.initial-steps (spec.return v) = UNIV × Id
by (auto simp: spec.initial-steps-def spec.singleton.le-conv)

lemma bind:
  shows spec.initial-steps (f ⤜ g)
      = spec.initial-steps f
        ∪ spec.initial-steps (⨆ v. spec.pre (spec.post.strongest (f ⊓ spec.return v) v) ⊓ g v) (is ?lhs = ?rhs)
proof(rule antisym)
  show ?lhs ≤ ?rhs
    by (fastforce simp: spec.initial-steps-def spec.post.strongest-alt-def spec.singleton.le-conv
                        trace.split-all Cons-eq-append-conv trace.natural.simps
                  elim!: spec.singleton.bind-le)
  show ?rhs ≤ ?lhs
    by (auto simp: spec.initial-steps-def spec.post.strongest-alt-def spec.singleton.le-conv
                   trace.split-all spec.bind.incompleteI order.trans[OF - spec.bind.continueI, rotated])
qed
```

lemma *rel*:
  shows *spec.initial-steps* (*spec.rel r*) = *r* ∪ *UNIV* × *Id*
by (*auto simp*: *spec.initial-steps-def spec.singleton.le-conv*)

lemma *pre*:
  shows *spec.initial-steps* (*spec.pre P*) = *UNIV* × *Pre P*
by (*auto simp*: *spec.initial-steps-def spec.singleton.le-conv*)

lemma *post*:
  shows *spec.initial-steps* (*spec.post Q*) = *UNIV*
by (*auto simp*: *spec.initial-steps-def spec.singleton.le-conv*)

setup ‹*Sign.mandatory-path term*›

lemma *none*:
  shows *spec.initial-steps* (*spec.term.none P*) = *spec.initial-steps P*
by (*auto simp*: *spec.initial-steps-def spec.singleton.le-conv*)

lemma *all*:
  shows *spec.initial-steps* (*spec.term.all P*) = *spec.initial-steps P*
by (*auto simp*: *spec.initial-steps-def spec.singleton.le-conv order.trans*[*rotated*]
        spec.singleton.mono trace.less-eq-None)

setup ‹*Sign.parent-path*›

setup ‹*Sign.parent-path*›

setup ‹*Sign.parent-path*›

8.13 Heyting implication

setup ‹*Sign.mandatory-path spec*›

setup ‹*Sign.mandatory-path singleton*›

lemma *heyting-le-conv*:
  shows ⟪*σ*⟫ ≤ *P* ⟶$_H$ *Q* ⟷ (∀ *σ'*≤*σ*. ⟪*σ'*⟫ ≤ *P* ⟶ ⟪*σ'*⟫ ≤ *Q*) (**is** *?lhs* ⟷ *?rhs*)
proof(*rule iffI*)
  show *?lhs* ⟹ *?rhs*
    by (*meson order.trans heyting.mp spec.singleton.mono*)
  show *?rhs* ⟹ *?lhs*
    by (*rule spec.singleton.exhaust*[*of P*])
      (*clarsimp simp*: *heyting inf-Sup spec.singleton.inf trace.less-eq-take-def spec.singleton-le-conv*;
       metis spec.singleton.simps(*1*) *trace.take.naturalE*(*2*))
qed

setup ‹*Sign.parent-path*›

Connect the generic definition of Heyting implication to a concrete one in the model.

lift-definition *heyting* :: ('*a*, '*s*, '*v*) *spec* ⇒ ('*a*, '*s*, '*v*) *spec* ⇒ ('*a*, '*s*, '*v*) *spec* **is**
  downwards.imp
by (*rule raw.spec.closed.downwards-imp*)

lemma *heyting-alt-def*:
  shows (⟶$_H$) = (*spec.heyting* :: -⇒-⇒('*a*, '*s*, '*v*) *spec*)
proof −
  have *P* ≤ *spec.heyting Q R* ⟷ *P* ⊓ *Q* ≤ *R* **for** *P Q R* :: ('*a*, '*s*, '*v*) *spec*
    by *transfer* (*simp add*: *raw.spec.closed.heyting-downwards-imp*)

with *heyting* **show** *?thesis* **by** (*intro fun-eqI antisym*; *fast*)
qed

declare *spec.heyting.transfer*[*transfer-rule del*]

setup ‹*Sign.mandatory-path heyting*›

lemma *transfer-alt*[*transfer-rule*]:
shows *rel-fun* (*pcr-spec* (=) (=) (=)) (*rel-fun* (*pcr-spec* (=) (=) (=)) (*pcr-spec* (=) (=) (=))) *downwards.imp* ($\longrightarrow_H$)
by (*simp add*: *spec.heyting.transfer spec.heyting-alt-def*)

An example due to Abadi and Merz (1995, p504) where the (TLA) model validates a theorem that is not intuitionistically valid. This is "some kind of linearity" and intuitively encodes disjunction elimination.

lemma *linearity*:
fixes Q :: (-, -, -) *spec*
shows $((P \longrightarrow_H Q) \longrightarrow_H R) \sqcap ((Q \longrightarrow_H P) \longrightarrow_H R) \leq R$
by *transfer*
(*clarsimp simp*: *downwards.imp-def*;
meson downwards.closed-in[*OF - - raw.spec.closed.downwards-closed*] *trace.less-eq-same-cases order.refl*)

lemma *SupR*:
fixes P :: (-, -, -) *spec*
assumes $X \neq \{\}$
shows $P \longrightarrow_H (\bigsqcup x{\in}X.\ Q\ x) = (\bigsqcup x{\in}X.\ P \longrightarrow_H Q\ x)$ (**is** *?lhs* = *?rhs*)
proof(*rule antisym*[*OF spec.singleton-le-extI heyting.SUPR-le*])
show $\langle\!|\sigma|\!\rangle \leq$ *?rhs* **if** $\langle\!|\sigma|\!\rangle \leq$ *?lhs* **for** σ
proof(*cases* $\langle\!|\sigma|\!\rangle \leq P$)
case *True* **with** ‹$\langle\!|\sigma|\!\rangle \leq$ *?lhs*› **show** *?thesis* **by** (*simp add*: *heyting inf.absorb1*)
next
case *False* **show** *?thesis*
proof(*cases* $\langle\!|$*trace.init* σ, [], *None*$|\!\rangle \leq P$)
case *True* **with** ‹$\neg\ \langle\!|\sigma|\!\rangle \leq P$›
obtain j **where** $\forall\, i{\leq}j.\ \langle\!|$*trace.take* $i\ \sigma|\!\rangle \leq P$
and $\neg\ \langle\!|$*trace.take* (*Suc* j) $\sigma|\!\rangle \leq P$
using *ex-least-nat-less*[**where** $P{=}\lambda i.\ \neg\langle\!|$*trace.take* $i\ \sigma|\!\rangle \leq P$ **and** n=*Suc* (*length* (*trace.rest* σ))]
by (*clarsimp simp*: *less-Suc-eq-le simp flip*: *trace.take.Ex-all*)
with ‹$\langle\!|\sigma|\!\rangle \leq$ *?lhs*› **show** *?thesis*
by (*clarsimp simp*: *spec.singleton.heyting-le-conv dest*!: *spec*[**where** x=*trace.take* $j\ \sigma$])
(*metis not-less-eq-eq order-refl spec.singleton.mono spec.singleton-le-ext-conv*
trace.less-eq-takeE trace.take.mono)
next
case *False* **with** ‹$X \neq \{\}$› ‹$\langle\!|\sigma|\!\rangle \leq$ *?lhs*› **show** *?thesis*
by (*clarsimp simp*: *spec.singleton.heyting-le-conv simp flip*: *ex-in-conv*)
(*metis trace.take.0 spec.singleton.takeI trace.less-eq-take-def trace.take.sel*(*1*))
qed
qed
qed

lemma *cont*:
fixes P :: (-, -, -) *spec*
shows *cont Sup* ($\leq$) *Sup* ($\leq$) (($\longrightarrow_H$) P)
by (*rule contI*) (*simp add*: *spec.heyting.SupR*[**where** Q=*id*, *simplified*])

lemma *mcont*:
fixes P :: (-, -, -) *spec*
shows *mcont Sup* ($\leq$) *Sup* ($\leq$) (($\longrightarrow_H$) P)
by (*simp add*: *mcontI*[*OF - spec.heyting.cont*])

lemmas *mcont2mcont*[*cont-intro*] = *mcont2mcont*[*OF spec.heyting.mcont*, *of luba orda Q P*] **for** *luba orda Q P*

lemma *non-triv*:
shows $P \longrightarrow_H \bot \leq P \longleftrightarrow$ *spec.idle* $\leq P$ (**is** *?lhs* $\longleftrightarrow$ *?rhs*)
proof(*rule iffI*)
show *?lhs* $\Longrightarrow$ *?rhs*
by (*rule spec.singleton.exhaust*[*of P*])
(*fastforce dest*: *spec*[**where** x=$\langle x, [], None\rangle$ **for** x]
simp: *spec.idle-def heyting-def heyting.inf-Sup-distrib trace.split-all*
spec.singleton.inf spec.singleton-le-conv trace.less-eq-None trace.natural.simps)
have *spec.idle* $\longrightarrow_H \bot \leq$ *spec.idle*
by (*fastforce intro*: *spec.singleton-le-extI*
dest: *spec*[**where** x=*trace.T* (*trace.init* σ) [] *None* **for** σ]
simp: *spec.singleton.le-conv spec.singleton.heyting-le-conv spec.singleton.not-bot trace.less-eq-None*)
then show *?lhs* **if** *?rhs*
by − (*strengthen ord-to-strengthen*(*2*)[*OF* ‹*?rhs*›])
qed

lemma *post*:
shows *spec.post Q* $\longrightarrow_H$ *spec.post Q'* = *spec.post* ($\lambda v\ s.\ Q\ v\ s \longrightarrow Q'\ v\ s$) (**is** *?lhs* = *?rhs*)
proof(*rule antisym*)
show *?lhs* $\leq$ *?rhs*
by (*auto intro*: *spec.singleton-le-extI simp*: *spec.singleton.heyting-le-conv spec.singleton.le-conv split*: *option.splits*)
show *?rhs* $\leq$ *?lhs*
by (*auto simp add*: *heyting simp flip*: *spec.post.conj intro*: *spec.post.mono*)
qed

setup ‹*Sign.parent-path*›

setup ‹*Sign.mandatory-path invmap*›

lemma *heyting*:
shows *spec.invmap af sf vf* (*P* $\longrightarrow_H$ *Q*) = *spec.invmap af sf vf P* $\longrightarrow_H$ *spec.invmap af sf vf Q* (**is** *?lhs* = *?rhs*)
proof(*rule antisym*)
show *?lhs* $\leq$ *?rhs* **by** (*simp add*: *heyting heyting.detachment spec.invmap.mono flip*: *spec.invmap.inf*)
show *?rhs* $\leq$ *?lhs*
by (*simp add*: *heyting heyting.detachment spec.map.inf-distr flip*: *spec.map-invmap.galois spec.invmap.inf*)
(*simp add*: *spec.invmap.mono spec.map-invmap.galois*)
qed

setup ‹*Sign.parent-path*›

setup ‹*Sign.mandatory-path term*›

lemma *heyting-noneL-allR-mono*:
fixes *P* :: (-, -, *'v*) *spec*
fixes *Q* :: (-, -, *'v*) *spec*
shows *spec.term.none P* $\longrightarrow_H$ *Q* = *P* $\longrightarrow_H$ *spec.term.all Q* (**is** *?lhs* = *?rhs*)
proof(*rule antisym*)
show *?lhs* $\leq$ *?rhs*
by (*simp add*: *heyting spec.term.none.inf flip*: *spec.term.galois*) (*simp add*: *heyting.uncurry*)
show *?rhs* $\leq$ *?lhs*
by (*simp add*: *heyting heyting.discharge spec.term.closed.none-inf-monomorphic spec.term.galois*)
qed

setup ‹*Sign.mandatory-path all*›

lemma *heyting*: — polymorphic *spec.term.all*
fixes P :: (-, -, $'v$) *spec*
fixes Q :: (-, -, $'v$) *spec*
shows (*spec.term.all* ($P \longrightarrow_H Q$) :: (-, -, $'w$) *spec*)
= *spec.term.none* $P \longrightarrow_H$ *spec.term.all* Q (**is** *?lhs* = *?rhs*)
proof(*rule antisym*)
show *?lhs* ≤ *?rhs*
by (*simp add*: *heyting spec.term.none.inf flip*: *spec.term.galois*)
(*metis heyting.detachment*(*2*) *le-inf-iff spec.term.none.contractive spec.term.none.inf*(*2*))
have *spec.term.none* (*spec.term.none* $P \longrightarrow_H$ *spec.term.all* Q :: (-, -, $'w$) *spec*)
⊓ *spec.term.none* (*spec.term.none* P :: (-, -, $'w$) *spec*)
≤ Q
by (*metis heyting.detachment*(*2*) *inf-sup-ord*(*2*) *spec.term.galois spec.term.none.inf*(*2*))
then show *?rhs* ≤ *?lhs*
by (*simp add*: *heyting flip*: *spec.term.galois*)
(*metis spec.term.cl-def spec.term.all.monomorphic spec.term.none-all*
heyting.detachment(*2*) *spec.term.heyting-noneL-allR-mono*)
qed

setup ‹*Sign.parent-path*›

setup ‹*Sign.mandatory-path none*›

lemma *heyting-le*:
shows *spec.term.none* ($P \longrightarrow_H Q$) ≤ *spec.term.all* $P \longrightarrow_H$ *spec.term.none* Q
by (*simp add*: *spec.term.galois spec.term.all.heyting heyting.mono spec.term.all.expansive*)

setup ‹*Sign.parent-path*›

setup ‹*Sign.mandatory-path closed*›

lemma *heyting*:
assumes $Q \in$ *spec.term.closed* -
shows $P \longrightarrow_H Q \in$ *spec.term.closed* -
by (*rule spec.term.closed-clI*)
(*simp add*: *spec.term.all.heyting spec.term.heyting-noneL-allR-mono spec.term.all.monomorphic*
flip: *spec.term.all.closed-conv*[*OF assms*])

setup ‹*Sign.parent-path*›

setup ‹*Sign.parent-path*›

setup ‹*Sign.parent-path*›

8.14 Miscellaneous algebra

setup ‹*Sign.mandatory-path spec*›

setup ‹*Sign.mandatory-path steps*›

lemma *bind*:
shows *spec.steps* ($f \gg\!= g$)
= *spec.steps* $f \cup (\bigcup v.$ *spec.steps* (*spec.pre* (*spec.post.strongest* f v) ⊓ g v)) (**is** *?lhs* = *?rhs*)
proof(*rule antisym*)
show *?lhs* ≤ *?rhs*
unfolding *spec.rel.galois*
by (*rule spec.singleton-le-extI*)

```
    (fastforce elim!: spec.singleton.bind-le
              simp: spec.singleton.le-conv spec.steps.member-conv trace.steps'.append
                    spec.post.strongest-alt-def)
  show ?rhs ≤ ?lhs
    by (fastforce simp: spec.post.strongest-alt-def spec.bind-def spec.continue-def
                        spec.steps.term.none spec.steps.Sup spec.steps.sup spec.steps.singleton
                        spec.steps.member-conv spec.singleton.le-conv trace.split-Ex trace.steps'.append)
qed

setup ‹Sign.parent-path›

setup ‹Sign.mandatory-path map›

lemma idle:
  shows spec.map af sf vf spec.idle = spec.pre (λs. s ∈ range sf) ⊓ spec.idle (is ?lhs = ?rhs)
proof(rule antisym)
  show ?lhs ≤ ?rhs
    by (auto simp: spec.idle-def spec.map.Sup spec.map.singleton spec.singleton.pre-le-conv)
  show ?rhs ≤ ?lhs
    by (auto simp: spec.idle-def spec.pre-def trace.split-all image-image inf-Sup Sup-inf
                   spec.map.Sup spec.map.singleton spec.singleton.inf
             elim!: trace.less-eqE)
qed

lemma return:
  fixes F :: ('v × 'a × 's × 's) set
  shows spec.map af sf vf (spec.return v)
      = spec.pre (λs. s ∈ range sf) ⊓ spec.return (vf v) (is ?lhs = ?rhs)
proof(rule antisym[OF - spec.singleton-le-extI])
  show ?lhs ≤ ?rhs
    by (force simp: spec.return-def spec.action-def spec.idle-def
                    spec.map.Sup spec.map.sup spec.map.singleton spec.singleton.pre-le-conv)
  fix σ
  assume ⟪σ⟫ ≤ ?rhs
  then obtain s where trace.init σ = sf s by (clarsimp simp: spec.singleton.le-conv)
  with ‹⟪σ⟫ ≤ ?rhs› show ⟪σ⟫ ≤ ?lhs
    by (simp add: spec.singleton.le-conv exI[where x=trace.T s [] (Some v)]
                  spec.singleton-le-conv trace.natural-def trace.less-eq-None
             flip: trace.natural'.eq-Nil-conv
            split: option.split-asm)
qed

lemma kleene-star-le:
  fixes P :: ('a, 's, unit) spec
  fixes af :: 'a ⇒ 'b
  fixes sf :: 's ⇒ 't
  fixes vf :: unit ⇒ unit
  shows spec.map af sf vf (spec.kleene.star P) ≤ spec.kleene.star (spec.map af sf vf P) (is - ≤ ?rhs)
proof(induct rule: spec.kleene.star.fixp-induct[where P=λR. spec.map af sf vf (R P) ≤ ?rhs, case-names adm bot step])
  case (step R) show ?case
    apply (simp add: spec.map.sup spec.map.return order.trans[OF - spec.kleene.epsilon-star-le])
    apply (subst spec.kleene.star.simps)
    apply (strengthen ord-to-strengthen(1)[OF spec.map.bind-le])
    apply (strengthen ord-to-strengthen(1)[OF step])
    apply (simp add: fun-unit-id[where f=vf])
    done
qed (simp-all add: spec.map.bot)
```

lemma *rel-le*:
  shows *spec.map af sf vf* (*spec.rel r*) ≤ *spec.rel* (*map-prod af* (*map-prod sf sf*) ‘ *r*)
apply (*simp add*: *spec.rel-def spec.term.none.map flip*: *spec.term.galois*)
apply (*simp add*: *spec.rel.act-def flip*: *spec.term.none.map*[**where** *vf*=*id*])
apply (*strengthen ord-to-strengthen*(*1*)[*OF spec.map.kleene-star-le*])
apply (*strengthen ord-to-strengthen*(*1*)[*OF spec.map.action-le*])
apply (*strengthen ord-to-strengthen*(*1*)[*OF spec.term.none.contractive*])
apply (*auto intro*!: *monotoneD*[*OF spec.kleene.monotone-star*] *spec.action.mono*)
done

General lemmas for *spec.map* are elusive. We relate it to *spec.rel*, *spec.pre* and *spec.post* under a somewhat weak constraint. Intuitively we ask that, for distinct representations (s_0 and s_0') of an abstract state (*sf* s_0 where *sf* s_0' = *sf* s_0), if agent a can evolve s_0 to s_1 according to r ((a, s_0, s_1) ∈ r) then there is an agent a' where *af* a' = *af* a that can evolve s_0' to an s_1' which represents the same abstract state (*sf* s_1' = *sf* s_1).

All injective *sf* satisfy this condition.

context
  fixes *af* :: ′*a* ⇒ ′*b*
  fixes *sf* :: ′*s* ⇒ ′*t*
  fixes *vf* :: ′*v* ⇒ ′*w*
begin

context
  fixes *r* :: (′*a*, ′*s*) *steps*
  assumes *step-cong*: ∀ *a s*$_0$ *s*$_1$ *s*$_0$′. (*a*, *s*$_0$, *s*$_1$) ∈ *r* ∧ *sf s*$_1$ ≠ *sf s*$_0$ ∧ *sf s*$_0$′ = *sf s*$_0$
                ⟶ (∃ *a*′ *s*$_1$′. *af a*′ = *af a* ∧ *sf s*$_1$′ = *sf s*$_1$ ∧ (*a*′, *s*$_0$′, *s*$_1$′) ∈ *r*)
begin

private lemma *map-relE*[*consumes 1*]:
  fixes *xs* :: (′*b* × ′*t*) *list*
  assumes *trace.steps*′ *s xs* ⊆ *map-prod af* (*map-prod sf sf*) ‘ *r*
  obtains (*Idle*) *snd* ‘ *set xs* ⊆ {*s*}
      | (*Step*) *s*′ *xs*′
        where *sf s*′ = *s*
        and *trace.natural*′ *s xs* = *map* (*map-prod af sf*) *xs*′
        and *trace.steps*′ *s*′ *xs*′ ⊆ *r*
using *assms*
proof(*atomize-elim*, *induct xs rule*: *prefix-induct*)
  case (*snoc xs x*) **show** *?case*
  proof(*cases snd x* = *trace.final*′ *s xs*)
    case *True* **with** *snoc*(*2*,*3*) **show** *?thesis*
      by (*fastforce simp*: *trace.steps*′*.append trace.natural*′*.append*)
  next
    case *False*
    with *snoc*(*2*,*3*) **consider**
       (*idle*) *snd* ‘ *set xs* ⊆ {*s*}
     | (*step*) *s*′ *xs*′
       where *sf s*′ = *s*
         and *trace.natural*′ *s xs* = *map* (*map-prod af sf*) *xs*′
         and *trace.steps*′ *s*′ *xs*′ ⊆ *r*
      by (*auto 0 0 simp*: *trace.steps*′*.append*)
    then show *?thesis*
    proof *cases*
      case *idle* **with** *snoc*(*3*) **show** *?thesis*
        by (*cases x*)
          (*clarsimp simp*: *trace.steps*′*.append trace.natural*′*.append Cons-eq-map-conv*
              simp flip: *trace.natural*′*.eq-Nil-conv ex-simps*
                  split: *if-splits*;

```
        metis)
    next
      case (step s xs') with False snoc(3) step-cong show ?thesis
        by (cases x)
          (clarsimp simp: trace.steps'.append trace.natural'.append append-eq-map-conv Cons-eq-map-conv
              simp flip: ex-simps
                 intro!: exI[where x=s] exI[where x=xs'];
           metis trace.final'.map trace.final'.natural')
    qed
  qed
qed simp

lemma rel:
  shows spec.map af sf vf (spec.rel r)
      = spec.rel (map-prod af (map-prod sf sf) ‘ r)
       ⊓ spec.pre (λs. s ∈ range sf)
       ⊓ spec.post (λv s. v ∈ range vf) (is ?lhs = ?rhs)
proof(rule antisym[OF spec.singleton-le-extI spec.singleton-le-extI])
  show ⟪σ⟫ ≤ ?rhs if ⟪σ⟫ ≤ ?lhs for σ
  proof(intro le-infI)
    from that show ⟪σ⟫ ≤ spec.rel (map-prod af (map-prod sf sf) ‘ r)
      by (force simp: spec.singleton.le-conv spec.steps.singleton trace.steps'.map
              dest: spec.steps.mono)
    from that show ⟪σ⟫ ≤ spec.pre (λs. s ∈ range sf)
      by (fastforce simp: spec.singleton.le-conv spec.singleton-le-conv trace.natural-def
                  elim: trace.less-eqE)
    from that show ⟪σ⟫ ≤ spec.post (λv s. v ∈ range vf)
      by (cases σ) (clarsimp simp: spec.singleton.le-conv split: option.split)
  qed
  show ⟪σ⟫ ≤ ?lhs if ⟪σ⟫ ≤ ?rhs for σ
    using that[simplified, simplified spec.singleton.le-conv, THEN conjunct1]
          that[simplified, simplified spec.singleton.le-conv, THEN conjunct2]
  proof(induct rule: map-relE)
    case Idle then show ?case
      by (cases σ)
        (clarsimp simp: spec.singleton.le-conv;
         force simp: trace.natural.idle trace.natural.simps f-inv-into-f order-le-less
              split: option.split-asm
             intro!: exI[where x=trace.T s [] (map-option (inv vf) (trace.term σ)) for s])
  next
    case (Step s xs)
    from Step(1,3,4) Step(2)[symmetric] show ?case
      by (cases σ)
        (clarsimp simp: spec.singleton.le-conv f-inv-into-f[OF rangeI] trace.natural'.natural'
                        exI[where x=trace.T s xs (map-option (inv vf) (trace.term σ))]
                 split: option.split-asm)
  qed
qed

lemma pre:
  fixes P :: 't pred
  shows spec.map af sf vf (spec.pre (λs. P (sf s)))
      = spec.pre (λs. P s ∧ s ∈ range sf) ⊓ spec.post (λv s. s ∈ range sf ⟶ v ∈ range vf)
        ⊓ spec.rel (range af × range sf × range sf) (is ?lhs = ?rhs)
proof(rule antisym[OF - spec.singleton-le-extI])
  show ?lhs ≤ ?rhs
    by (simp add: spec.map-invmap.galois spec.invmap.pre spec.invmap.post spec.invmap.rel
                  map-prod-vimage-Times vimage-range spec.rel.UNIV)
```

fix σ
assume $\langle\!|\sigma|\!\rangle \leq$ *?rhs*
then obtain *s xs*
where *P* (*sf s*)
and *trace.init* σ = *sf s*
and *case trace.term* σ *of None* ⇒ *True*
| *Some v* ⇒ *trace.final′* (*trace.init* σ) (*trace.rest* σ) ∈ *range sf* ⟶ *v* ∈ *range vf*
and *map* (*map-prod af sf*) *xs* = *trace.natural′* (*sf s*) (*trace.rest* σ)
by (*clarsimp simp*: *spec.singleton.le-conv trace.steps′.map-range-conv*)
then show $\langle\!|\sigma|\!\rangle \leq$ *?lhs*
by (*cases* σ)
(*fastforce intro*: *exI*[**where** *x*=*trace.T s xs* (*Some* (*inv vf* (*the* (*trace.term* σ))))]
range-eqI[**where** *x*=*trace.final′ s xs*]
dest: *arg-cong*[**where** *f*=*trace.final′* (*sf s*)]
simp: *spec.singleton.le-conv trace.final′.map f-inv-into-f trace.natural′.natural′*
order.trans[*OF spec.singleton.less-eq-None spec.singleton.simps*(*2*)]
split: *option.split-asm*)
qed

lemma *post*:
fixes *Q* :: *′w* ⇒ *′t pred*
shows *spec.map af sf vf* (*spec.post* (λ*v s*. *Q* (*vf v*) (*sf s*)))
= *spec.pre* (λ*s*. *s* ∈ *range sf*) ⊓ *spec.post* (λ*v s*. *s* ∈ *range sf* ⟶ *Q v s* ∧ *v* ∈ *range vf*)
⊓ *spec.rel* (*range af* × *range sf* × *range sf*) (**is** *?lhs* = *?rhs*)
proof(*rule antisym*[*OF - spec.singleton-le-extI*])
show *?lhs* ≤ *?rhs*
by (*simp add*: *spec.map-invmap.galois spec.invmap.pre spec.invmap.post spec.invmap.rel*
map-prod-vimage-Times vimage-range spec.rel.UNIV)
fix σ
assume $\langle\!|\sigma|\!\rangle \leq$ *?rhs*
then obtain *s xs*
where *trace.init* σ = *sf s*
and *case trace.term* σ *of None* ⇒ *True* | *Some v* ⇒ *trace.final′* (*trace.init* σ) (*trace.rest* σ) ∈ *range sf* ⟶ *Q v* (*trace.final′* (*trace.init* σ) (*trace.rest* σ)) ∧ *v* ∈ *range vf*
and *map* (*map-prod af sf*) *xs* = *trace.natural′* (*sf s*) (*trace.rest* σ)
by (*clarsimp simp*: *spec.singleton.le-conv trace.steps′.map-range-conv*)
then show $\langle\!|\sigma|\!\rangle \leq$ *?lhs*
by (*cases* σ)
(*clarsimp simp*: *spec.singleton.le-conv trace.natural′.natural′*
intro!: *exI*[**where** *x*=*trace.T s xs* (*map-option* (*inv vf*) (*trace.term* σ))]
split: *option.split-asm*;
clarsimp dest!: *arg-cong*[**where** *f*=*trace.final′* (*sf s*)] *simp*: *trace.final′.map*;
metis f-inv-into-f rangeI)
qed

end

end

setup ‹*Sign.parent-path*›

setup ‹*Sign.mandatory-path invmap*›

lemma *idle*:
shows *spec.invmap af sf vf spec.idle*
= *spec.term.none* (*spec.rel* (*UNIV* × *map-prod sf sf* −‘ *Id*) :: (*′a*, *′s*, *unit*) *spec*) (**is** *?lhs* = *?rhs*)
proof(*rule antisym*[*OF spec.singleton-le-extI spec.singleton-le-extI*])
have *sf s* = *sf s′*

```
    if (a, s, s') ∈ trace.steps' s0 xs
   and (λx. sf (snd x)) ` set xs ⊆ {sf s0}
   for a :: 'a and s s' s0 :: 's and xs :: ('a × 's) list
    using that by (induct xs arbitrary: s0) (auto simp: trace.steps'.Cons-eq-if split: if-split-asm)
  then show ⟪σ⟫ ≤ ?rhs if ⟪σ⟫ ≤ ?lhs for σ
    using that by (clarsimp simp: spec.singleton.le-conv image-image)
  have sf s' = sf s
    if trace.steps' s xs ⊆ UNIV × map-prod sf sf −` Id
   and (a, s') ∈ set xs
   for a s s' and xs :: ('a × 's) list
    using that
    by (induct xs arbitrary: s)
       (auto simp: Diff-subset-conv trace.steps'.Cons-eq-if split: if-split-asm)
  then show ⟪σ⟫ ≤ ?lhs if ⟪σ⟫ ≤ ?rhs for σ
    using that by (clarsimp simp: spec.singleton.le-conv)
qed

lemma inf-rel:
  shows spec.rel (map-prod af (map-prod sf sf) −` (r ∪ UNIV × Id)) ⊓ spec.invmap af sf vf P = spec.invmap
af sf vf (spec.rel r ⊓ P)
    and spec.invmap af sf vf P ⊓ spec.rel (map-prod af (map-prod sf sf) −` (r ∪ UNIV × Id)) = spec.invmap af
sf vf (P ⊓ spec.rel r)
by (simp-all add: inf-commute spec.invmap.rel spec.invmap.inf)

lemma action: — (* could restrict the stuttering expansion to range af or an arbitrary element of that
  fixes af :: 'a ⇒ 'b
  fixes sf :: 's ⇒ 't
  fixes vf :: 'v ⇒ 'w
  fixes F :: ('w × 'b × 't × 't) set
  defines F' ≡ map-prod id (map-prod af (map-prod sf sf))
                −` (F ∪ {(v, a', s, s) |v a a' s. (v, a, s, s) ∈ F ∧ ¬surj af})
  shows spec.invmap af sf vf (spec.action F)
      = spec.rel (UNIV × map-prod sf sf −` Id)
        ⤜ (λ-::unit. spec.action F')
        ⤜ (λv. spec.rel (UNIV × map-prod sf sf −` Id)
        ⤜ (λ-::unit. ⨆v'∈vf −` {v}. spec.return v'))) (is ?lhs = ?rhs)
proof(rule antisym[OF spec.singleton-le-extI], unfold spec.singleton.invmap-le-conv)
  have ∗: sf x = sf y
    if (λx. sf (snd x)) ` set xs ⊆ {sf s}
   and (a, x, y) ∈ trace.steps' s xs
   for s a x y and v :: 'v and xs :: ('a × 's) list
   using that
   by (induct xs arbitrary: s; clarsimp simp: trace.steps'.Cons-eq-if split: if-split-asm; metis)
  show ⟪σ⟫ ≤ ?rhs if ⟪trace.map af sf vf σ⟫ ≤ spec.action F for σ
  proof(cases ↯ (trace.map af sf vf σ) = trace.T (trace.init (trace.map af sf vf σ)) [] None)
    case True then show ?thesis
      by (cases σ)
         (force simp: spec.singleton.le-conv trace.natural-def trace.natural'.eq-Nil-conv image-image
               dest: ∗
               intro: spec.bind.incompleteI)
  next
    case False with that show ?thesis
    proof(cases rule: spec.singleton.action-not-idle-le-splitE)
      case (return v a) then show ?thesis
        by (cases σ; clarsimp simp: image-image)
           ( rule spec.action.stutterI[where v=v and a=inv af a]
                 spec.bind.continueI[where ys=[], simplified]
           | ( fastforce simp: spec.singleton.le-conv F'-def trace.final'.map-idle surj-f-inv-f dest: ∗ )
```

```
        )+
    next
      case (step v a ys zs) then show ?thesis
        by (cases σ; clarsimp simp: map-eq-append-conv image-image split: option.split-asm)
          (rule spec.bind.continueI
                spec.bind.continueI[where xs=[x] for x, simplified]
                spec.bind.incompleteI[where g=⟨Sup X⟩ for X]
                spec.bind.continueI[where ys=[], simplified]
          | ( rule spec.action.stepI; force simp: F'-def trace.final'.map-idle )
          | ( fastforce simp: spec.singleton.le-conv trace.final'.map-idle dest: * )
          )+
    qed
  qed
  have *: map-prod af (map-prod sf sf) ` (UNIV × map-prod sf sf −` Id) − UNIV × Id = {} by blast
  have (v, a, s, s') ∈ F' ⟹ ⟪sf s, [(af a, sf s')], None⟫ ≤ spec.action F for v a s s'
    by (auto simp: F'-def spec.action.stepI intro: order.trans[OF spec.idle.minimal-le spec.idle.action-le])
  moreover
  have ⟦(vf v, a, s, s') ∈ F'; sf s' = trace.init σ; ⟪σ⟫ ≤ spec.pre (λs. s ∈ range sf); ⟪σ⟫ ≤ spec.return (vf v)⟧
    ⟹ ⟪sf s, (af a, trace.init σ) # trace.rest σ, trace.term σ⟫ ≤ spec.action F for v a s s' σ
    by (auto simp: F'-def spec.action.stepI spec.action.stutterI spec.singleton.le-conv
                   spec.singleton.Cons[where as=trace.rest σ]
          intro: order.trans[OF spec.idle.minimal-le spec.idle.action-le]
          split: option.split-asm)
  ultimately have spec.action (map-prod id (map-prod af (map-prod sf sf)) ` F')
                    ⤜ (λv. ⨆x∈vf −` {v}. spec.pre (λs. s ∈ range sf) ⊓ spec.return v)
                  ≤ spec.action F
    by (subst spec.action-def)
      (auto simp: spec.bind.SupL spec.bind.supL spec.bind.singletonL spec.idle-le split-def spec.term.none.singleton)
  then show ?rhs ≤ ?lhs
    apply (fold spec.map-invmap.galois)
    apply (strengthen ord-to-strengthen(1)[OF spec.map.bind-le])+
    apply (strengthen ord-to-strengthen[OF spec.map.rel-le])
    apply (strengthen ord-to-strengthen[OF spec.map.action-le])
    apply (subst (1 2) spec.rel.minus-Id[where A=UNIV, symmetric])
    apply (simp add: image-image * spec.map.return spec.rel.empty spec.bind.SupL spec.bind.returnL
                     spec.idle.action-le spec.idle.bind-le-conv spec.bind.SUPR-not-empty spec.bind.supR
                     spec.bind.return spec.map.Sup)
    done
qed

lemma return:
  fixes af :: 'a ⇒ 'b
  fixes sf :: 's ⇒ 't
  fixes vf :: 'v ⇒ 'w
  fixes F :: ('w × 'b × 't × 't) set
  shows spec.invmap af sf vf (spec.return v)
      = spec.rel (UNIV × map-prod sf sf −` Id) ⤜ (λ-::unit. ⨆v'∈vf −` {v}. spec.return v')
proof −
  have *: spec.action ({()} × UNIV × map-prod sf sf −` Id) = spec.rel.act (UNIV × map-prod sf sf −` Id)
    by (auto simp: spec.rel.act-def intro: spec.action.cong)
  show ?thesis
    apply (subst spec.return-def)
    apply (simp add: spec.invmap.action map-prod-vimage-Times)
    apply (subst sup.absorb1, force)
    apply (simp add: spec.action.return-const[where V={v} and W={()}] spec.bind.bind spec.bind.return *)
    apply (simp add: spec.rel.wind-bind flip: spec.bind.bind spec.rel.unfoldL)
    done
qed
```

setup ‹*Sign.parent-path*›

setup ‹*Sign.parent-path*›

9 Constructions in the *('a, 's, 'v) spec* lattice

9.1 Constrains-at-most

Abadi and Plotkin (1993, §3.1) require that processes to be composed in parallel *constrain at most* (CAM) distinct sets of agents: intuitively each process cannot block other processes from taking steps after any of its transitions. We model this as a closure.

See §9.2 for a discussion of their composition rules.

Observations:

- the sense of the relation r here is inverted wrt Abadi/Plotkin
- this is a key ingredient in interference closure (§9.3)
- this closure is antimatroidal

setup ‹*Sign.mandatory-path spec*›

setup ‹*Sign.mandatory-path cam*›

definition *cl* :: ('*a*, '*s*) *steps* ⇒ ('*a*, '*s*, '*v*) *spec* ⇒ ('*a*, '*s*, '*v*) *spec* **where**
  cl r P = *P* ⊔ *spec.term.none* (*spec.term.all P* ≫= (λ-::*unit*. *spec.rel r* :: (-, -, *unit*) *spec*))

setup ‹*Sign.parent-path*›

setup ‹*Sign.mandatory-path term*›

setup ‹*Sign.mandatory-path none.cam*›

lemma *cl*:
  shows *spec.term.none* (*spec.cam.cl r P*) = *spec.cam.cl r* (*spec.term.none P*)
by (*simp add*: *spec.cam.cl-def spec.bind.supL spec.bind.bind spec.term.all.bind ac-simps*
      flip: *spec.bind.botR bot-fun-def*)

lemma *cl-rel-wind*:
  fixes *P* :: ('*a*, '*s*, '*v*) *spec*
  shows *spec.cam.cl r P* ≫ *spec.term.none* (*spec.rel r* :: ('*a*, '*s*, '*w*) *spec*)
    = *spec.term.none* (*spec.cam.cl r P*)
by (*simp add*: *spec.cam.cl-def spec.term.none.sup spec.term.none.bind spec.bind.supL spec.bind.bind*
        bot-fun-def sup.absorb2
        spec.vmap.unitL[**where** *f*=*P*] *spec.vmap.unitL*[**where** *f*=*spec.term.all P*]
        spec.vmap.unitL[**where** *f*=*spec.rel r* :: ('*a*, '*s*, '*w*) *spec*]
        spec.term.all.vmap-unit spec.vmap.unit-rel spec.bind.mono spec.term.all.expansive
      flip: *spec.bind.botR*)

setup ‹*Sign.parent-path*›

setup ‹*Sign.mandatory-path all.cam*›

lemma *cl-le*: — Converse does not hold
  shows *spec.cam.cl r* (*spec.term.all P*) ≤ *spec.term.all* (*spec.cam.cl r P*)
by (*simp add*: *spec.term.none.cam.cl flip*: *spec.term.galois*) (*simp flip*: *spec.term.none.cam.cl*)

setup ‹*Sign.parent-path*›

setup ‹*Sign.parent-path*›

interpretation *cam*: *closure-complete-distrib-lattice-distributive-class spec.cam.cl r* **for** *r* :: ('*a*, '*s*) *steps*
proof *standard*
show *P* ≤ *spec.cam.cl r Q* ⟷ *spec.cam.cl r P* ≤ *spec.cam.cl r Q* (**is** *?lhs* ⟷ *?rhs*) **for** *P Q* :: ('*a*, '*s*, '*v*) *spec*
proof(*rule iffI*)
assume *?lhs* **show** *?rhs*
apply (*subst spec.cam.cl-def*)
apply (*strengthen ord-to-strengthen*(*1*)[*OF* ‹*?lhs*›])
apply (*simp add*: *spec.cam.cl-def spec.term.galois spec.term.all.sup spec.term.all.bind*
spec.bind.supL spec.term.all.rel spec.bind.bind spec.rel.wind-bind)
done
next
show *?rhs* ⟹ *?lhs*
by (*simp add*: *spec.cam.cl-def*)
qed
show *spec.cam.cl r* (⨆ *P*) ≤ ⨆(*spec.cam.cl r* ‘ *P*) ⊔ *spec.cam.cl r* ⊥ **for** *P* :: ('*a*, '*s*, '*v*) *spec set*
by (*simp add*: *spec.cam.cl-def spec.term.none.bind spec.term.all.Sup spec.bind.SupL*
spec.term.none.Sup SUP-upper2)
qed

setup ‹*Sign.mandatory-path cam*›

setup ‹*Sign.mandatory-path cl*›

lemma *bot*[*simp*]:
shows *spec.cam.cl r* ⊥ = ⊥
by (*simp add*: *spec.cam.cl-def*)

lemma *mono*:
fixes *r* :: ('*a*, '*s*) *steps*
assumes *r* ⊆ *r'*
assumes *P* ≤ *P'*
shows *spec.cam.cl r P* ≤ *spec.cam.cl r' P'*
unfolding *spec.cam.cl-def*
apply (*strengthen ord-to-strengthen*(*1*)[*OF* ‹*r* ≤ *r'*›])
apply (*strengthen ord-to-strengthen*(*1*)[*OF* ‹*P* ≤ *P'*›])
apply *blast*
done

declare *spec.cam.strengthen-cl*[*strg del*]

lemma *strengthen*[*strg*]:
assumes *st-ord F r r'*
assumes *st-ord F P P'*
shows *st-ord F* (*spec.cam.cl r P*) (*spec.cam.cl r' P'*)
using *assms* **by** (*cases F*; *simp add*: *spec.cam.cl.mono*)

lemma *Sup*:
shows *spec.cam.cl r* (⨆ *X*) = (⨆ *P*∈*X*. *spec.cam.cl r P*)
by (*simp add*: *spec.cam.cl-Sup*)

lemmas *sup* = *spec.cam.cl.Sup*[**where** *X*={*P*, *Q*} **for** *P Q*, *simplified*]

lemma *rel-empty*:
shows *spec.cam.cl* {} *P* = *P*

```
by (simp add: spec.cam.cl-def spec.rel.empty sup.absorb1 UNIV-unit)

lemma rel-reflcl:
  shows spec.cam.cl (r ∪ A × Id) P = spec.cam.cl r P
    and spec.cam.cl (A × Id ∪ r) P = spec.cam.cl r P
by (simp-all add: spec.cam.cl-def spec.rel.reflcl)

lemma rel-minus-Id:
  shows spec.cam.cl (r − UNIV × Id) P = spec.cam.cl r P
by (metis Un-Diff-cancel2 spec.cam.cl.rel-reflcl(1))

lemma Inf:
  shows spec.cam.cl r (⨅X) = ⨅(spec.cam.cl r ‘ X) (is ?lhs = ?rhs)
proof(rule antisym[OF spec.cam.cl-Inf-le spec.singleton-le-extI])
  show ⟪σ⟫ ≤ ?lhs if ⟪σ⟫ ≤ ?rhs for σ
  proof (cases trace.term σ)
    case None
    have ⟪σ⟫ ≤ ⨅ (spec.term.all ‘ X) ⤜ (λv. spec.term.none (spec.rel r))
      if x ∈ X and ¬ ⟪σ⟫ ≤ x
     for x
    proof −
      from ‹⟪σ⟫ ≤ ?rhs› that
      have ⟪σ⟫ ≤ spec.term.all x ⤜ (λ-::unit. spec.term.none (spec.rel r :: (-, -, unit) spec))
        by (auto simp: spec.cam.cl-def le-Inf-iff spec.term.none.bind)
      then show ?thesis
      proof(induct rule: spec.singleton.bind-le)
        case incomplete with ‹¬ ⟪σ⟫ ≤ x› show ?case
          using order-trans by auto
      next
        case (continue σ_f σ_g v_f)
        from None obtain xs ys
          where ∗: ∀xs′ zs. trace.rest σ = xs′ @ zs ∧ trace.steps′ (trace.final′ (trace.init σ) xs′) zs ⊆ r
                      ⟶ length xs ≤ length xs′
                xs @ ys = trace.rest σ
                trace.steps′ (trace.final′ (trace.init σ) xs) ys ⊆ r
          using ex-has-least-nat[where P=λxs. ∃ys. trace.rest σ = xs @ ys
                                        ∧ trace.steps′ (trace.final′ (trace.init σ) xs) ys ⊆ r
                                and k=trace.rest σ
                                and m=length]
          by clarsimp
        show ?case
        proof(induct rule: spec.bind.continueI[where s=trace.init σ and xs=xs and ys=ys
                                                  and v=undefined and w=trace.term σ,
                                               simplified ‹xs @ ys = trace.rest σ› trace.t.collapse,
                                               case-names f g])
          case f
          have ⟪trace.init σ, xs, None⟫ ≤ x
            if x ∈ X
           and ⟪σ⟫ ≤ spec.cam.cl r x
           for x
            using that(2)[unfolded spec.cam.cl-def, simplified]
          proof(induct rule: disjE[consumes 1, case-names expansive cam])
            case expansive with ‹xs @ ys = trace.rest σ› show ?case
              by (cases σ)
                 (fastforce elim: order.trans[rotated]
                            simp: spec.singleton.mono trace.less-eq-same-append-conv)

          next
```

```
      case cam from cam[unfolded spec.term.none.bind] show ?case
      proof(induct rule: spec.singleton.bind-le)
        case incomplete with ‹xs @ ys = trace.rest σ› show ?case
          by clarsimp
            (metis prefixI spec.singleton.mono spec.singleton-le-ext-conv
                spec.term.none.contractive trace.less-eq-None(2))
      next
        case (continue σ_f σ_g v_f) with *(1,2) show ?case
          by (clarsimp simp: spec.singleton.le-conv trace.less-eq-None
                elim!: order.trans[rotated]
                intro!: spec.singleton.mono)
            (metis prefixI prefix-length-prefix)
      qed
    qed
    with ‹⟪σ⟫ ≤ ?rhs› show ?case
      by (simp add: le-Inf-iff spec.singleton.le-conv exI[where x=None])
  next
    case g with None *(3) show ?case
      by (simp add: spec.singleton.le-conv)
  qed
 qed
 qed
 then show ⟪σ⟫ ≤ ?lhs
  by (auto simp: spec.cam.cl-def spec.term.none.bind spec.term.all.Inf le-Inf-iff)
 next
  case Some with ‹⟪σ⟫ ≤ ?rhs› show ?thesis
   by (simp add: le-Inf-iff spec.cam.cl-def spec.singleton.term.none-le-conv)
 qed
qed

lemmas inf = spec.cam.cl.Inf[where X={P, Q} for P Q, simplified]

lemma idle:
 shows spec.cam.cl r spec.idle = spec.term.none (spec.rel r :: (-, -, unit) spec)
by (simp add: spec.cam.cl-def spec.term.all.idle UNIV-unit spec.bind.returnL
          spec.idle-le sup-absorb2)

lemma bind:
 shows spec.cam.cl r (f ⤜ g) = spec.cam.cl r f ⤜ (λv. spec.cam.cl r (g v))
by (simp add: spec.cam.cl-def spec.bind.supL spec.bind.supR spec.bind.bind ac-simps spec.term.all.bind
      flip: spec.bind.botR bot-fun-def)

lemma action:
 fixes r :: ('a, 's) steps
 fixes F :: ('v × 'a × 's × 's) set
 shows spec.cam.cl r (spec.action F)
     = spec.action F
    ⊔ spec.term.none (spec.action F ≫ (spec.rel r :: (-, -, unit) spec))
    ⊔ spec.term.none (spec.rel r :: (-, -, unit) spec)
by (simp add: spec.cam.cl-def spec.term.all.action spec.term.none.bind spec.term.none.sup
          spec.bind.botR spec.bind.supL spec.bind.returnL spec.idle-le
          spec.vmap.unitL[where f=spec.action F] spec.map.surj-sf-action
          UNIV-unit map-prod-const-image ac-simps
      flip: spec.return-def)

lemma return:
 shows spec.cam.cl r (spec.return v) = spec.return v ⊔ spec.term.none (spec.rel r :: (-, -, unit) spec)
unfolding spec.return-def spec.cam.cl.action
```

by (*simp add*: *spec.bind.returnL spec.idle-le bot-fun-def*
flip: *spec.return-def bot-fun-def*)

lemma *rel-le*:
assumes $r \subseteq r' \vee r' \subseteq r$
shows *spec.cam.cl r* (*spec.rel r'*) ≤ *spec.rel* ($r \cup r'$)
using *assms*
by (*auto simp*: *spec.cam.cl-def spec.rel.mono spec.term.all.rel*
spec.rel.wind-bind-leading spec.rel.wind-bind-trailing spec.term.galois)

lemma *rel*:
assumes $r \subseteq r'$
shows *spec.cam.cl r* (*spec.rel r'*) = *spec.rel r'*
by (*simp add*: *assms spec.eq-iff spec.cam.expansive*
order.trans[*OF spec.cam.cl.rel-le*[*OF disjI1*] *spec.rel.mono*])

lemma *inf-rel*:
fixes *r* :: ('*a*, '*s*) *steps*
fixes *s* :: ('*a*, '*s*) *steps*
fixes *P* :: ('*a*, '*s*, '*v*) *spec*
shows *spec.rel r* ⊓ *spec.cam.cl r' P* = *spec.cam.cl* ($r \cap r'$) (*spec.rel r* ⊓ *P*) (**is** *?thesis1*)
and *spec.cam.cl r' P* ⊓ *spec.rel r* = *spec.cam.cl* ($r \cap r'$) (*spec.rel r* ⊓ *P*) (**is** *?thesis2*)
proof −
show *?thesis1*
by (*simp add*: *spec.cam.cl-def ac-simps inf-sup-distrib*
spec.term.none.bind spec.term.all.inf spec.term.all.rel
spec.bind.inf-rel spec.rel.inf spec.term.none.inf spec.term.none.inf-none-rel(*1*))
then show *?thesis2*
by (*rule inf-commute-conv*)
qed

lemma *bind-return*:
shows *spec.cam.cl r* (*f* ≫ *spec.return v*) = *spec.cam.cl r f* ≫ *spec.return v*
by (*simp add*: *spec.cam.cl.bind spec.cam.cl.return spec.bind.supR sup.absorb1 spec.term.none.cam.cl-rel-wind*)

lemma *heyting-le*:
shows *spec.cam.cl r* (*P* ⟶$_H$ *Q*) ≤ *P* ⟶$_H$ *spec.cam.cl r Q*
by (*force intro*!: *SupI*
dest: *spec.cam.mono-cl*[**where** *r*=*r*]
elim: *order.trans*[*rotated*]
simp: *heyting-def spec.cam.cl.Sup spec.cam.cl.inf le-infI1 spec.cam.expansive*)

lemma *pre*:
shows *spec.cam.cl r* (*spec.pre P*) = *spec.pre P*
by (*simp add*: *spec.cam.cl-def spec.term.none.bind spec.term.all.pre sup-iff-le spec.bind.inf-pre*
flip: *inf-iff-le*)

lemma *post*:
shows *spec.cam.cl r* (*spec.post Q*) = *spec.post Q*
by (*simp add*: *spec.cam.cl-def spec.term.none.post-le sup-iff-le*)

setup ‹*Sign.parent-path*›

setup ‹*Sign.mandatory-path closed*›

lemma *empty*:
shows *spec.cam.closed* {} = *UNIV*
by (*simp add*: *order.eq-iff spec.cam.cl.rel-empty spec.cam.closed-clI subsetI*)

lemma *antimonotone*:
shows *antimono spec.cam.closed*
by (*rule monotoneI*) (*auto intro*: *spec.cam.closed-clI elim*: *spec.cam.le-closedE*[*OF spec.cam.cl.mono*])

lemmas *strengthen*[*strg*] = *st-ord-antimono*[*OF spec.cam.closed.antimonotone*]
lemmas *antimono* = *antimonoD*[*OF spec.cam.closed.antimonotone*, *of r r'* **for** *r r'*]

lemma *reflcl*:
shows *spec.cam.closed* (*r* ∪ *A* × *Id*) = *spec.cam.closed r*
by (*simp add*: *spec.cam.cl.rel-reflcl*(*1*) *spec.cam.closed-def*)

setup ‹*Sign.mandatory-path term*›

lemma *none*:
assumes *P* ∈ *spec.cam.closed r*
shows *spec.term.none P* ∈ *spec.cam.closed r*
by (*simp add*: *assms spec.cam.closed-clI flip*: *spec.term.none.cam.cl spec.cam.closed-conv*[*OF assms*])

setup ‹*Sign.parent-path*›

lemma *bind*:
fixes *f* :: ('*a*, '*s*, '*v*) *spec*
fixes *g* :: '*v* ⇒ ('*a*, '*s*, '*w*) *spec*
assumes *f* ∈ *spec.cam.closed r*
assumes ⋀*x*. *g x* ∈ *spec.cam.closed r*
shows *f* ⤜ *g* ∈ *spec.cam.closed r*
by (*simp add*: *assms spec.cam.closed-clI spec.cam.least spec.cam.cl.bind spec.bind.mono*)

lemma *rel*[*intro*]:
assumes *r* ⊆ *r'*
shows *spec.rel r'* ∈ *spec.cam.closed r*
by (*simp add*: *assms spec.cam.closed-clI spec.cam.cl.rel*)

lemma *pre*[*intro*]:
shows *spec.pre P* ∈ *spec.cam.closed r*
by (*simp add*: *spec.cam.closed-clI spec.cam.cl.pre*)

lemma *post*[*intro*]:
shows *spec.post Q* ∈ *spec.cam.closed r*
by (*simp add*: *spec.cam.closed-clI spec.cam.cl.post*)

lemma *heyting*[*intro*]:
assumes *Q* ∈ *spec.cam.closed r*
shows *P* ⟶$_H$ *Q* ∈ *spec.cam.closed r*
by (*rule spec.cam.closed-clI*)
(*simp add*: *assms order.trans*[*OF spec.cam.cl.heyting-le*] *flip*: *spec.cam.closed-conv*)

lemma *snoc-conv*:
fixes *P* :: ('*a*, '*s*, '*v*) *spec*
assumes *P* ∈ *spec.cam.closed r*
assumes (*fst x*, *trace.final' s xs*, *snd x*) ∈ *r* ∪ *UNIV* × *Id*
shows ⟪*s*, *xs* @ [*x*], *None*⟫ ≤ *P* ⟷ ⟪*s*, *xs*, *None*⟫ ≤ *P* (**is** *?lhs* ⟷ *?rhs*)
proof(*rule iffI*)
show *?lhs* ⟹ *?rhs*
by (*erule order.trans*[*rotated*]) (*simp add*: *spec.singleton.mono trace.less-eq-same-append-conv*)
from *assms*(*2*) **show** *?rhs* ⟹ *?lhs*
by (*subst spec.cam.closed-conv*[*OF* ‹*P* ∈ *spec.cam.closed r*›])

(*auto simp*: *spec.cam.cl-def spec.singleton.term.none-le-conv*
spec.term.none.singleton spec.steps.singleton
simp flip: *spec.rel.galois spec.term.galois*
intro: *spec.bind.continueI*)
qed

setup ‹*Sign.parent-path*›

setup ‹*Sign.parent-path*›

setup ‹*Sign.mandatory-path invmap.cam*›

lemma *cl*:
fixes *af* :: $'a \Rightarrow 'b$
fixes *sf* :: $'s \Rightarrow 't$
fixes *vf* :: $'v \Rightarrow 'w$
fixes *r* :: ($'b$, $'t$) *steps*
fixes *P* :: ($'b$, $'t$, $'w$) *spec*
shows *spec.invmap af sf vf* (*spec.cam.cl r P*)
= *spec.cam.cl* (*map-prod af* (*map-prod sf sf*) −‘ (*r* ∪ *UNIV* × *Id*)) (*spec.invmap af sf vf P*)
by (*simp add*: *spec.cam.cl-def spec.invmap.sup spec.invmap.bind spec.invmap.rel spec.term.all.invmap*
flip: *spec.term.none.invmap-gen*[**where** *vf*=*id*])

setup ‹*Sign.parent-path*›

setup ‹*Sign.mandatory-path map.cam*›

lemma *cl-le*:
fixes *af* :: $'a \Rightarrow 'b$
fixes *sf* :: $'s \Rightarrow 't$
fixes *vf* :: $'v \Rightarrow 'w$
fixes *r* :: ($'a$, $'s$) *steps*
fixes *P* :: ($'a$, $'s$, $'v$) *spec*
shows *spec.map af sf vf* (*spec.cam.cl r P*)
≤ *spec.cam.cl* (*map-prod af* (*map-prod sf sf*) ‘ *r*) (*spec.map af sf vf P*)
by (*simp add*: *spec.map-invmap.galois spec.map-invmap.upper-lower-expansive*
spec.invmap.cam.cl spec.cam.cl.mono subset-vimage-iff le-supI1)

lemma *cl-inj-sf*:
fixes *af* :: $'a \Rightarrow 'b$
fixes *sf* :: $'s \Rightarrow 't$
fixes *vf* :: $'v \Rightarrow 'w$
fixes *r* :: ($'a$, $'s$) *steps*
fixes *P* :: ($'a$, $'s$, $'v$) *spec*
assumes *inj sf*
shows *spec.map af sf vf* (*spec.cam.cl r P*)
= *spec.cam.cl* (*map-prod af* (*map-prod sf sf*) ‘ *r*) (*spec.map af sf vf P*)
apply (*simp add*: *spec.cam.cl-def spec.map.sup spec.map.bind-inj-sf*[*OF* ‹*inj sf*›] *spec.term.all.map*
flip: *spec.term.none.map-gen*[**where** *vf*=*id*])
apply (*subst spec.map.rel*, *blast dest*: *injD*[*OF* ‹*inj sf*›])
apply (*simp add*: *inf.absorb1 spec.map-invmap.galois spec.invmap.post flip*: *spec.bind-post-pre*)
done

setup ‹*Sign.parent-path*›

setup ‹*Sign.parent-path*›

9.2 Abadi and Plotkin's composition principle

Abadi and Plotkin (1991, 1993) develop a theory of circular reasoning about Heyting implication for safety properties under the mild condition that each is CAM-closed with respect to the other.

setup ‹*Sign.mandatory-path spec*›

abbreviation *ap-cam-cl* :: *'a set* ⇒ (*'a*, *'s*, *'v*) *spec* ⇒ (*'a*, *'s*, *'v*) *spec* **where**
ap-cam-cl as ≡ *spec.cam.cl* ((−*as*) × *UNIV*)

abbreviation (*input*) *ap-cam-closed* :: *'a set* ⇒ (*'a*, *'s*, *'v*) *spec set* **where**
ap-cam-closed as ≡ *spec.cam.closed* ((−*as*) × *UNIV*)

lemma *composition-principle-1*:
fixes *P* :: (*'a*, *'s*, *'v*) *spec*
assumes *P* ∈ *spec.ap-cam-closed as*
assumes *P* ∈ *spec.term.closed* -
assumes *spec.idle* ≤ *P*
shows *spec.ap-cam-cl* (−*as*) *P* ⟶$_H$ *P* ≤ *P* (**is** *?lhs* ≤ *?rhs*)
proof(*rule spec.term.closed.singleton-le-extI*)
show ⟪*s*, *xs*, *None*⟫ ≤ *?rhs* **if** ⟪*s*, *xs*, *None*⟫ ≤ *?lhs* **for** *s xs*
using *that*
proof(*induct xs rule*: *rev-induct*)
case *Nil*
from ‹*spec.idle* ≤ *P*› **show** *?case*
by (*simp add*: *order.trans*[*OF spec.idle.minimal-le*])
next
case (*snoc x xs*)
from *snoc.prems* **have** ⟪*s*, *xs*, *None*⟫ ≤ *spec.ap-cam-cl* (− *as*) *P* ⟶$_H$ *P*
by (*simp add*: *order.trans*[*OF spec.singleton.mono*, *rotated*] *trace.less-eq-None*)
with *snoc.hyps* **have** ⟪*s*, *xs*, *None*⟫ ≤ *P* **by** *blast*
show *?case*
proof(*cases fst x* ∈ *as*)
case *True*
with ‹⟪*s*, *xs*, *None*⟫ ≤ *P*› **have** ⟪*s*, *xs* @ [*x*], *None*⟫ ≤ *spec.ap-cam-cl* (−*as*) *P*
by (*subst spec.cam.closed.snoc-conv*) (*auto simp*: *order.trans*[*OF* - *spec.cam.expansive*])
with *snoc.prems* **show** *?thesis* **by** (*blast intro*: *heyting.mp*)
next
case *False* **with** ‹*P* ∈ *spec.ap-cam-closed as*› ‹⟪*s*, *xs*, *None*⟫ ≤ *P*› **show** *?thesis*
by (*simp add*: *spec.cam.closed.snoc-conv*)
qed
qed
qed *fact*

lemma *composition-principle-half*: — Abadi and Plotkin (1993, §3.1(4)) – cleaner than in Abadi and Plotkin (1991, §3.1)
assumes M_1 ∈ *spec.ap-cam-closed* a_1
assumes M_2 ∈ *spec.ap-cam-closed* a_2
assumes M_1 ∈ *spec.term.closed* -
assumes *spec.idle* ≤ M_1
assumes $a_1 \cap a_2 = \{\}$
shows (M_1 ⟶$_H$ M_2) ⊓ (M_2 ⟶$_H$ M_1) ≤ M_1
proof −
have (M_1 ⟶$_H$ M_2) ⊓ (M_2 ⟶$_H$ M_1) ≤ (*spec.ap-cam-cl* (−a_1) M_1 ⟶$_H$ *spec.ap-cam-cl* (−a_1) M_2) ⊓ (M_2 ⟶$_H$ M_1)
by (*rule inf-mono*[*OF heyting.closure-imp-distrib-le*[*OF closure.axioms*(*2*)[*OF spec.cam.closure-axioms*]] *order.refl*])
(*simp add*: *spec.cam.cl.inf*)
also have ... ≤ *spec.ap-cam-cl* (−a_1) M_1 ⟶$_H$ M_1

proof −
from ‹$M_2 \in$ *spec.ap-cam-closed* a_2› ‹$a_1 \cap a_2 = \{\}$› **have** *spec.ap-cam-cl* $(-a_1)$ $M_2 \leq M_2$
by (*fastforce intro*: *spec.cam.least elim*: *subsetD*[*OF spec.cam.closed.antimono*, *rotated*])
then show *?thesis*
by (*simp add*: *heyting.trans order-antisym-conv spec.cam.expansive*)
qed
also have $\ldots \leq M_1$
by (*rule spec.composition-principle-1*[*OF* ‹$M_1 \in$ *spec.ap-cam-closed* a_1› ‹$M_1 \in$ *spec.term.closed* -› ‹*spec.idle* $\leq M_1$›])
finally show *?thesis* **.**
qed

theorem *composition-principle*: — Abadi and Plotkin (1993, §3.1(3))
assumes $M_1 \in$ *spec.ap-cam-closed* a_1
assumes $M_2 \in$ *spec.ap-cam-closed* a_2
assumes $M_1 \in$ *spec.term.closed* -
assumes $M_2 \in$ *spec.term.closed* -
assumes *spec.idle* $\leq M_1$
assumes *spec.idle* $\leq M_2$
assumes $a_1 \cap a_2 = \{\}$
shows $(M_1 \longrightarrow_H M_2) \sqcap (M_2 \longrightarrow_H M_1) \leq M_1 \sqcap M_2$
using *assms* **by** (*metis spec.composition-principle-half inf.bounded-iff inf.commute*)

An infinitary variant can be established in essentially the same way as *spec.composition-principle-1*.

lemma *ag-circular*:
fixes *Ps* :: $'a \Rightarrow ('a, 's, 'v)$ *spec*
assumes *cam-closed*: $\bigwedge a.\ a \in as \Longrightarrow Ps\ a \in$ *spec.ap-cam-closed* $\{a\}$
assumes *term-closed*: $\bigwedge a.\ a \in as \Longrightarrow Ps\ a \in$ *spec.term.closed* -
assumes *idle*: $\bigwedge a.\ a \in as \Longrightarrow$ *spec.idle* $\leq Ps\ a$
shows $(\bigsqcap a{\in}as.\ (\bigsqcap a'{\in}as{-}\{a\}.\ Ps\ a') \longrightarrow_H Ps\ a) \leq (\bigsqcap a{\in}as.\ Ps\ a)$ (**is** *?lhs* $\leq$ *?rhs*)
proof(*rule spec.term.closed.singleton-le-extI*)
show $\langle s, xs, None\rangle \leq$ *?rhs* **if** $\langle s, xs, None\rangle \leq$ *?lhs* **for** *s xs*
using *that*
proof(*induct xs rule*: *rev-induct*)
case *Nil* **from** *idle* **show** *?case*
by (*simp add*: *le-INF-iff order.trans*[*OF spec.idle.minimal-le*])
next
case (*snoc x xs*)
have ∗: $\langle s, xs, None\rangle \leq$ *?rhs*
by (*simp add*: *snoc*(*1*) *order.trans*[*OF spec.singleton.mono snoc*(*2*)] *trace.less-eq-same-append-conv*)
have $\langle s, xs\ @\ [x], None\rangle \leq Ps\ a$ **if** $a \in as$ **for** *a*
proof(*cases fst* $x = a$)
case *True*
with *cam-closed* ∗ **have** $\langle s, xs\ @\ [x], None\rangle \leq \bigsqcap (Ps\ `\ (as - \{a\}))$
by (*subst spec.cam.closed.snoc-conv*[**where** $r{=}\bigsqcap a'{\in}as{-}\{a\}.\ (-\ \{a'\}) \times UNIV$])
(*auto simp*: *le-INF-iff intro*: *subsetD*[*OF spec.cam.closed.antimono*, *rotated*])
with *snoc.prems*(*1*) ‹$a \in as$› **show** *?thesis*
by (*meson heyting.mp le-INF-iff*)
next
case *False* **with** *cam-closed* ∗ ‹$a \in as$› **show** *?thesis*
by (*fastforce simp*: *spec.cam.closed.snoc-conv le-INF-iff*)
qed
then show *?case* **by** (*blast intro*: *INFI*)
qed
from *term-closed* **show** *?rhs* ∈ *spec.term.closed* -
by (*fastforce simp*: *spec.term.all.monomorphic*)
qed

setup ‹*Sign.parent-path*›

9.3 Interference closure

We add environment interference to the beginnings and ends of behaviors for two reasons:

- it ensures the wellformedness of parallel composition as conjunction (see §9.5)
- it guarantees the monad laws hold (see §13.3.1)
  - *spec.cam.cl* by itself is too weak to justify these

We use this closure to build the program sublattice of the (*'a*, *'s*, *'v*) *spec* lattice (see §13).
Observations:

- if processes are made out of actions then it is not necessary to apply *spec.cam.cl*

setup ‹*Sign.mandatory-path spec*›

setup ‹*Sign.mandatory-path interference*›

definition *cl* :: (*'a*, *'s*) *steps* ⇒ (*'a*, *'s*, *'v*) *spec* ⇒ (*'a*, *'s*, *'v*) *spec* **where**
cl r P = *spec.rel r* ≫= (λ-::*unit*. *spec.cam.cl r P*) ≫= (λ*v*. *spec.rel r* ≫= (λ-::*unit*. *spec.return v*))

setup ‹*Sign.parent-path*›

interpretation *interference*: *closure-complete-distrib-lattice-distributive-class spec.interference.cl r*
for *r* :: (*'a*, *'s*) *steps*
proof
show *P* ≤ *spec.interference.cl r Q* ⟷ *spec.interference.cl r P* ≤ *spec.interference.cl r Q* (**is** *?lhs* ⟷ *?rhs*)
for *P Q* :: (*'a*, *'s*, *'v*) *spec*
proof(*rule iffI*)
assume *?lhs* **show** *?rhs*
apply (*subst spec.interference.cl-def*)
apply (*strengthen ord-to-strengthen*(*1*)[*OF* ‹*?lhs*›])
apply (*simp add*: *spec.interference.cl-def spec.cam.cl.bind spec.cam.cl.return spec.cam.cl.rel*
spec.bind.bind spec.bind.supL spec.bind.supR
spec.bind.returnL spec.idle-le
flip: *bot-fun-def spec.bind.botR*)
apply (*simp add*: *spec.rel.wind-bind flip*: *spec.bind.bind*)
apply (*simp add*: *spec.bind.bind spec.bind.mono*)
done
next
assume *?rhs* **show** *?lhs*
apply (*strengthen ord-to-strengthen*(*2*)[*OF* ‹*?rhs*›])
apply (*simp add*: *spec.interference.cl-def spec.bind.bind*)
apply (*strengthen ord-to-strengthen*(*2*)[*OF spec.cam.expansive*])
apply (*strengthen ord-to-strengthen*(*2*)[*OF spec.return.rel-le*])
apply (*auto simp*: *spec.bind.return intro*: *spec.bind.returnL-le*)
done
qed
show *spec.interference.cl r* (⨆ *P*) ≤ ⨆ (*spec.interference.cl r* ‘ *P*) ⊔ *spec.interference.cl r* ⊥
for *P* :: (*'a*, *'s*, *'v*) *spec set*
by (*simp add*: *spec.interference.cl-def spec.cam.cl.Sup image-image*
spec.bind.SupL spec.bind.supL spec.bind.SUPR
flip: *bot-fun-def*)
qed

setup ‹*Sign.mandatory-path term*›

setup ‹*Sign.mandatory-path none*›

setup ‹*Sign.mandatory-path interference*›

lemma *cl*:
  shows *spec.term.none* (*spec.interference.cl r P*) = *spec.interference.cl r* (*spec.term.none P*)
by (*simp add*: *spec.interference.cl-def spec.term.none.bind spec.term.none.return*
        spec.bind.bind spec.bind.idleR spec.bind.botR spec.term.none.cam.cl-rel-wind
      flip: *spec.term.none.cam.cl*)

setup ‹*Sign.mandatory-path closed*›

lemma *rel-le*:
  assumes *P* ∈ *spec.interference.closed r*
  shows *spec.term.none* (*spec.rel r*) ≤ *P*
by (*subst spec.interference.closed-conv*[*OF assms*])
   (*simp add*: *spec.interference.cl-def spec.term.galois spec.term.all.bind spec.term.all.rel ac-simps*)

setup ‹*Sign.parent-path*›

setup ‹*Sign.mandatory-path all*›

lemma *cl-le*: — Converse does not hold
  shows *spec.interference.cl r* (*spec.term.all P*) ≤ *spec.term.all* (*spec.interference.cl r P*)
by (*simp add*: *spec.interference.cl-def spec.bind.bind spec.bind.idleR spec.bind.botR*
        spec.term.none.bind spec.term.none.return
        spec.term.none.cam.cl-rel-wind spec.term.none.cam.cl
      flip: *spec.term.galois*)
   (*simp add*: *spec.bind.mono flip*: *spec.term.none.cam.cl*)

setup ‹*Sign.parent-path*›

setup ‹*Sign.parent-path*›

setup ‹*Sign.parent-path*›

setup ‹*Sign.parent-path*›

setup ‹*Sign.mandatory-path cam.closed.interference*›

lemma *cl*:
  shows *spec.interference.cl r P* ∈ *spec.cam.closed r*
by (*metis spec.cam.closed-clI spec.interference.cl-def spec.interference.expansive*
      spec.interference.idempotent(*1*) *spec.cam.idempotent*(*1*))

lemma *closed-subseteq*:
  shows *spec.interference.closed r* ⊆ *spec.cam.closed r*
by (*metis spec.cam.closed.interference.cl spec.interference.closed-conv subsetI*)

setup ‹*Sign.parent-path*›

setup ‹*Sign.mandatory-path interference*›

setup ‹*Sign.mandatory-path cl*›

lemma *mono*:

assumes $r \subseteq r'$
assumes $P \leq P'$
shows *spec.interference.cl r P* $\leq$ *spec.interference.cl r' P'*
unfolding *spec.interference.cl-def*
apply (*strengthen ord-to-strengthen*(*1*)[*OF* ‹$r \subseteq r'$›])
apply (*strengthen ord-to-strengthen*(*1*)[*OF* ‹$P \leq P'$›])
apply *simp*
done

declare *spec.interference.strengthen-cl*[*strg del*]

lemma *strengthen*[*strg*]:
assumes *st-ord F r r'*
assumes *st-ord F P P'*
shows *st-ord F* (*spec.interference.cl r P*) (*spec.interference.cl r' P'*)
using *assms* **by** (*cases F*; *simp add*: *spec.interference.cl.mono*)

lemma *bot*:
shows *spec.interference.cl r* $\bot$ = *spec.term.none* (*spec.rel r* :: (-, -, *unit*) *spec*)
by (*simp add*: *spec.interference.cl-def spec.bind.bind flip*: *bot-fun-def spec.bind.botR*)

lemmas *Sup* = *spec.interference.cl-Sup*
lemmas *sup* = *spec.interference.cl-sup*

lemma *idle*:
shows *spec.interference.cl r spec.idle* = *spec.term.none* (*spec.rel r* :: (-, -, *unit*) *spec*)
by (*simp add*: *spec.interference.cl-def spec.cam.cl.idle spec.bind.bind spec.rel.wind-bind*
flip: *spec.term.none.bind*)

lemma *rel-empty*:
assumes *spec.idle* $\leq$ *P*
shows *spec.interference.cl* {} *P* = *P*
by (*simp add*: *spec.interference.cl-def spec.rel.empty spec.cam.cl.rel-empty spec.bind.return*
spec.bind.returnL assms UNIV-unit)

lemma *rel-reflcl*:
shows *spec.interference.cl* (*r* $\cup$ *A* $\times$ *Id*) *P* = *spec.interference.cl r P*
and *spec.interference.cl* (*A* $\times$ *Id* $\cup$ *r*) *P* = *spec.interference.cl r P*
by (*simp-all add*: *spec.interference.cl-def spec.cam.cl.rel-reflcl spec.rel.reflcl*)

lemma *rel-minus-Id*:
shows *spec.interference.cl* (*r* $-$ *UNIV* $\times$ *Id*) *P* = *spec.interference.cl r P*
by (*metis Un-Diff-cancel2 spec.interference.cl.rel-reflcl*(*1*))

lemma *inf-rel*:
shows *spec.interference.cl s P* $\sqcap$ *spec.rel r* = *spec.interference.cl* (*r* $\cap$ *s*) (*spec.rel r* $\sqcap$ *P*)
and *spec.rel r* $\sqcap$ *spec.interference.cl s P* = *spec.interference.cl* (*r* $\cap$ *s*) (*spec.rel r* $\sqcap$ *P*)
by (*simp-all add*: *spec.interference.cl-def spec.bind.inf-rel spec.return.inf-rel spec.cam.cl.inf-rel*
flip: *spec.rel.inf*)

lemma *bindL*:
assumes *f* $\in$ *spec.interference.closed r*
shows *spec.interference.cl r* (*f* $\gg\!=$ *g*) = *f* $\gg\!=$ (λ*v*. *spec.interference.cl r* (*g v*))
apply (*subst* (*1 2*) *spec.interference.closed-conv*[*OF assms*])
apply (*simp add*: *spec.interference.cl-def spec.bind.bind spec.cam.cl.bind spec.cam.cl.rel*
spec.cam.cl.return spec.bind.supL spec.bind.return)

apply (*simp add*: *spec.rel.wind-bind flip*: *spec.bind.bind*)
done

lemma *bindR*:
  assumes $\bigwedge v.\ g\ v \in$ *spec.interference.closed r*
  shows *spec.interference.cl r* (*f* $\gg\!=$ *g*) = *spec.interference.cl r f* $\gg\!=$ *g* (**is** *?lhs* = *?rhs*)
proof −
  from *assms* **have** *?lhs* = *spec.interference.cl r* (*f* $\gg\!=$ ($\lambda v.$ *spec.interference.cl r* (*g v*)))
    by (*meson spec.interference.closed-conv*)
  also have ... = *spec.interference.cl r f* $\gg\!=$ ($\lambda v.$ *spec.interference.cl r* (*g v*))
    apply (*simp add*: *spec.interference.cl-def spec.bind.bind spec.cam.cl.bind spec.cam.cl.rel*
        spec.cam.cl.return spec.bind.supL spec.bind.supR spec.bind.return
        sup.absorb1 spec.bind.mono
      flip: *spec.bind.botR*)
    apply (*simp add*: *spec.rel.wind-bind flip*: *spec.bind.bind*)
    done
  also from *assms* **have** ... = *?rhs*
    by (*simp flip*: *spec.interference.closed-conv*)
  finally show *?thesis* **.**
qed

lemma *bind-conv*:
  assumes *f* ∈ *spec.interference.closed r*
  assumes $\forall x.\ g\ x \in$ *spec.interference.closed r*
  shows *spec.interference.cl r* (*f* $\gg\!=$ *g*) = *f* $\gg\!=$ *g*
using *assms* **by** (*simp add*: *spec.interference.cl.bindR flip*: *spec.interference.closed-conv*)

lemma *action*:
  shows *spec.interference.cl r* (*spec.action F*)
      = *spec.rel r* $\gg\!=$ (λ-::*unit*. *spec.action F* $\gg\!=$ ($\lambda v.$ *spec.rel r* $\gg\!=$ (λ-::*unit*. *spec.return v*)))
by (*simp add*: *spec.interference.cl-def spec.cam.cl.action spec.bind.supL spec.bind.supR*
      flip: *spec.bind.botR spec.bind.bind spec.rel.unwind-bind*)
  (*simp add*: *spec.bind.bind sup.absorb1 spec.bind.mono*)

lemma *return*:
  shows *spec.interference.cl r* (*spec.return v*) = *spec.rel r* $\gg\!=$ (λ-::*unit*. *spec.return v*)
by (*simp add*: *spec.return-def spec.interference.cl.action spec.bind.bind*)
  (*simp add*: *spec.bind.return spec.rel.wind-bind flip*: *spec.return-def spec.bind.bind*)

lemma *bind-return*:
  shows *spec.interference.cl r* (*f* $\gg$ *spec.return v*) = *spec.interference.cl r f* $\gg$ *spec.return v*
by (*simp add*: *spec.interference.cl-def spec.bind.bind spec.bind.return spec.cam.cl.bind-return*)

lemma *rel*: — complicated by polymorphic *spec.rel*
  assumes *r* ⊆ *r′* ∨ *r′* ⊆ *r*
  shows *spec.interference.cl r* (*spec.rel r′*) = *spec.rel* (*r* ∪ *r′*) (**is** *?lhs* = *?rhs*)
using *assms*
proof
  show *?thesis* **if** *r* ⊆ *r′*
    apply (*simp add*: ‹*r* ⊆ *r′*› *sup.absorb2 spec.eq-iff spec.interference.expansive*)
    apply (*strengthen ord-to-strengthen*(*1*)[*OF* ‹*r* ⊆ *r′*›])
    apply (*metis spec.interference.cl.bot spec.interference.idempotent*(*1*) *spec.term.all.rel*
        spec.term.all-none spec.term.none.interference.all.cl-le)
    done
  show *?thesis* **if** *r′* ⊆ *r*
  proof(*rule antisym*)
    from ‹*r′* ⊆ *r*› **show** *?lhs* ≤ *?rhs*
      by (*simp add*: *inf.absorb-iff1 spec.interference.cl.inf-rel flip*: *spec.rel.inf*)

from ‹$r' \subseteq r$› **show** *?rhs* $\leq$ *?lhs*
by (*simp add*: *sup.absorb1 spec.interference.cl-def spec.cam.cl-def*
spec.rel.wind-bind-trailing le-supI1 spec.bind.supR spec.bind.return
order.trans[*OF - spec.bind.mono*[*OF order.refl spec.bind.mono*[*OF spec.return.rel-le order.refl*]]])
qed
qed

setup ‹*Sign.parent-path*›

setup ‹*Sign.parent-path*›

setup ‹*Sign.mandatory-path idle.interference*›

lemma *cl-le*[*spec.idle-le*]:
shows *spec.idle* $\leq$ *spec.interference.cl r P*
by (*simp add*: *spec.interference.cl-def spec.idle-le*)

lemma *closed-le*[*spec.idle-le*]:
assumes $P \in$ *spec.interference.closed r*
shows *spec.idle* $\leq P$
by (*subst spec.interference.closed-conv*[*OF assms*]) (*simp add*: *spec.idle.interference.cl-le*)

setup ‹*Sign.parent-path*›

setup ‹*Sign.mandatory-path map.interference*›

lemma *cl-sf-id*:
shows *spec.map af id vf* (*spec.interference.cl r P*)
$=$ *spec.interference.cl* (*map-prod af id* ‘ *r*) (*spec.map af id vf P*)
apply (*simp add*: *spec.interference.cl-def spec.map.return*
spec.map.bind-inj-sf[*OF inj-on-id*] *spec.map.cam.cl-inj-sf*[*OF inj-on-id*])
apply (*subst* (*1 2*) *spec.map.rel*, *force*, *force*)
apply (*simp add*: *spec.vmap.eq-return*(*2*) *spec.bind.bind*
spec.bind.returnL spec.idle-le
flip: *spec.map.cam.cl-inj-sf*[**where** *af*=*id* **and** *sf*=*id* **and** *vf*=*vf* **and** *P*=*spec.amap af P*,
simplified spec.map.comp, *simplified*, *folded id-def*])
done

setup ‹*Sign.parent-path*›

setup ‹*Sign.mandatory-path invmap.interference*›

lemma *cl*:
fixes *as* :: $'b$ *set*
fixes *af* :: $'a \Rightarrow 'b$
fixes *sf* :: $'s \Rightarrow 't$
fixes *vf* :: $'v \Rightarrow 'w$
fixes *r* :: ($'b$, $'t$) *steps*
fixes *P* :: ($'b$, $'t$, $'w$) *spec*
shows *spec.invmap af sf vf* (*spec.interference.cl r P*)
$=$ *spec.interference.cl* (*map-prod af* (*map-prod sf sf*) $-$‘ ($r \cup$ *UNIV* $\times$ *Id*)) (*spec.invmap af sf vf P*)
apply (*simp add*: *spec.interference.cl-def map-prod-vimage-Times spec.rel.wind-bind-trailing*
spec.invmap.bind spec.invmap.cam.cl spec.invmap.rel spec.invmap.return
flip: *spec.bind.bind*)
apply (*subst* (*2*) *spec.invmap.split-vinvmap*)
apply (*simp add*: *spec.cam.cl.bind spec.cam.cl.return spec.cam.cl.Sup spec.term.none.cam.cl-rel-wind*
spec.bind.mono spec.bind.bind spec.bind.SupL spec.bind.supL
spec.bind.SUPR spec.bind.supR spec.bind.returnL spec.idle-le spec.bind.botR

image-image sup.absorb1)
done

setup ‹*Sign.parent-path*›

setup ‹*Sign.mandatory-path interference.closed*›

lemma *antimonotone*:
shows *antimono spec.interference.closed*
proof(*rule antimonoI*)
show *spec.interference.closed r' ⊆ spec.interference.closed r* **if** *r ⊆ r'* **for** *r r'* :: (*'a*, *'s*) *steps*
unfolding *spec.interference.closed-def* **by** (*strengthen ord-to-strengthen*(*2*)[*OF* ‹*r ⊆ r'*›]) *simp*
qed

lemmas *strengthen*[*strg*] = *st-ord-antimono*[*OF spec.interference.closed.antimonotone*]
lemmas *antimono* = *antimonoD*[*OF spec.interference.closed.antimonotone*]

lemma *Sup'*:
assumes *X ⊆ spec.interference.closed r*
shows ⨆ *X ⊔ spec.term.none* (*spec.rel r* :: (-, -, *unit*) *spec*) ∈ *spec.interference.closed r*
by (*metis assms spec.interference.cl.bot spec.interference.closed-Sup*)

lemma *Sup-not-empty*:
assumes *X ⊆ spec.interference.closed r*
assumes *X ≠ {}*
shows ⨆ *X ∈ spec.interference.closed r*
using *spec.interference.closed-Sup*[*OF assms*(*1*)] *assms*
by (*simp add*: *assms spec.interference.closed-Sup*[*OF assms*(*1*)] *less-eq-Sup spec.interference.least*
subsetD sup.absorb1)

lemma *rel*:
assumes *r' ⊆ r*
shows *spec.rel r ∈ spec.interference.closed r'*
by (*metis assms spec.eq-iff inf.absorb-iff2 spec.interference.cl.inf-rel*(*2*) *spec.interference.closed-clI*)

lemma *bind-relL*:
fixes *P* :: (*'a*, *'s*, *'v*) *spec*
assumes *P ∈ spec.interference.closed r*
shows *spec.rel r* ⤜ (λ-::*unit*. *P*) = *P*
by (*subst* (*1 2*) *spec.interference.closed-conv*[*OF assms*])
(*simp add*: *spec.interference.cl-def spec.rel.wind-bind flip*: *spec.bind.bind*)

lemma *bind-relR*:
assumes *P ∈ spec.interference.closed r*
shows *P* ⤜ (λ*v*. *spec.rel r* ⤜ (λ-::*unit*. *Q v*)) = *P* ⤜ *Q*
by (*subst* (*1 2*) *spec.interference.closed-conv*[*OF assms*])
(*simp add*: *spec.interference.cl-def spec.bind.bind spec.bind.return*;
simp add: *spec.rel.wind-bind flip*: *spec.bind.bind*)

lemma *bind-rel-unitR*:
assumes *P ∈ spec.interference.closed r*
shows *P* ⟫ (*spec.rel r* :: (-, -, *unit*) *spec*) = *P*
by (*subst* (*1 2*) *spec.interference.closed-conv*[*OF assms*])
(*simp add*: *spec.interference.cl-def spec.bind.bind spec.rel.wind-bind*)

lemma *bind-rel-botR*:
assumes *P ∈ spec.interference.closed r*
shows *P* ⤜ (λ*v*. *spec.rel r* ⤜ (λ-::*unit*. ⊥)) = *P* ⤜ ⊥

by (*subst* (*1 2*) *spec.interference.closed-conv*[*OF assms*])
(*simp add*: *spec.interference.cl-def spec.bind.bind spec.bind.return*;
simp add: *spec.rel.wind-bind flip*: *spec.bind.bind*)

lemma *bind*[*intro*]:
fixes $f :: ('a, 's, 'v)$ *spec*
fixes $g :: 'v \Rightarrow ('a, 's, 'w)$ *spec*
assumes $f \in$ *spec.interference.closed r*
assumes $\bigwedge x.\ g\ x \in$ *spec.interference.closed r*
shows $(f \gg\!= g) \in$ *spec.interference.closed r*
using *assms* **by** (*simp add*: *spec.interference.closed-clI spec.interference.cl.bindL*
flip: *spec.interference.closed-conv*)

lemma *kleene-star*:
assumes $P \in$ *spec.interference.closed r*
assumes *spec.return* $() \leq P$
shows *spec.kleene.star P* $\in$ *spec.interference.closed r*
proof(*rule spec.interference.closed-clI*,
induct rule: *spec.kleene.star.fixp-induct*[**where** $P{=}\lambda R.$ *spec.interference.cl r* $(R\ P) \leq$ *spec.kleene.star P* , *case-names adm bot step*])
case *bot* **from** ‹$P \in$ *spec.interference.closed r*› **show** *?case*
by (*simp add*: *order.trans*[*OF - spec.kleene.expansive-star*] *spec.interference.cl.bot*
spec.term.none.interference.closed.rel-le)
next
case (*step R*) **show** *?case*
apply (*simp add*: *spec.interference.cl-sup spec.interference.cl.bindL*[*OF assms*(*1*)])
apply (*strengthen ord-to-strengthen*(*1*)[*OF step*])
apply (*strengthen ord-to-strengthen*(*1*)[*OF* ‹*spec.return* $() \leq P$›])
apply (*simp add*: *spec.kleene.fold-starL spec.kleene.expansive-star*
flip: *spec.interference.closed-conv*[*OF assms*(*1*)])
done
qed *simp-all*

lemma *map-sf-id*:
fixes $af :: 'a \Rightarrow 'b$
fixes $vf :: 'v \Rightarrow 'w$
assumes $P \in$ *spec.interference.closed r*
shows *spec.map af id vf P* $\in$ *spec.interference.closed* (*map-prod af id* ‘ *r*)
by (*rule spec.interference.closed-clI*)
(*subst* (*2*) *spec.interference.closed-conv*[*OF assms*];
simp add: *spec.map.interference.cl-sf-id map-prod-image-Times*)

lemma *invmap*:
fixes $af :: 'a \Rightarrow 'b$
fixes $sf :: 's \Rightarrow 't$
fixes $vf :: 'v \Rightarrow 'w$
assumes $P \in$ *spec.interference.closed r*
shows *spec.invmap af sf vf P* $\in$ *spec.interference.closed* (*map-prod af* (*map-prod sf sf*) $-$‘ *r*)
by (*rule spec.interference.closed-clI*)
(*subst* (*2*) *spec.interference.closed-conv*[*OF assms*];
fastforce simp: *spec.invmap.interference.cl intro*: *spec.interference.cl.mono*)

setup ‹*Sign.mandatory-path term*›

lemma *none*:
assumes $P \in$ *spec.interference.closed r*
shows *spec.term.none P* $\in$ *spec.interference.closed r*
by (*rule spec.interference.closed-clI*)

(*subst* (*2*) *spec.interference.closed-conv*[*OF assms*];
simp add: *spec.term.none.interference.cl*)

setup ‹*Sign.parent-path*›

setup ‹*Sign.parent-path*›

setup ‹*Sign.parent-path*›

9.4 The *'a agent* datatype

For compositionality we often wish to designate a specific agent as the environment.

datatype *'a agent* = *proc* (*the-agent*: *'a*) | *env*
type-synonym *sequential* = *unit agent* — Sequential programs (§13)
abbreviation *self* :: *sequential* **where** *self* ≡ *proc* ()

declare *agent.map-id*[*simp*]
declare *agent.map-id0*[*simp*]
declare *agent.map-id0*[*unfolded id-def*, *simp*]
declare *agent.map-comp*[*unfolded comp-def*, *simp*]

lemma *env-not-in-range-proc*[*iff*]:
shows *env* ∉ *range proc*
by *fastforce*

lemma *range-proc-conv*[*simp*]:
shows *x* ∈ *range proc* ⟷ *x* ≠ *env*
by (*cases x*) *simp-all*

lemma *inj-proc*[*iff*]:
shows *inj proc*
by (*simp add*: *inj-def*)

lemma *surj-the-inv-proc*[*iff*]:
shows *surj* (*the-inv proc*)
by (*meson inj-proc surjI the-inv-f-f*)

lemma *the-inv-proc*[*simp*]:
shows *the-inv proc* (*proc a*) = *a*
by (*simp add*: *the-inv-f-f*)

lemma *uminus-env-range-proc*[*simp*]:
shows −{*env*} = *range proc*
by (*auto intro*: *agent.exhaust*)

lemma *env-range-proc-UNIV*[*simp*]:
shows *insert env* (*range proc*) = *UNIV*
by (*auto intro*: *agent.exhaust*)

setup ‹*Sign.mandatory-path sequential*›

lemma *not-conv*[*simp*]:
shows *a* ≠ *env* ⟷ *a* = *self*
and *a* ≠ *self* ⟷ *a* = *env*
by (*cases a*; *simp*)+

lemma *range-proc-self*[*simp*]:
shows *range proc* = {*self*}

by *fastforce*

lemma *UNIV*:
  shows *UNIV* = {*env*, *self*}
by *fastforce*

lemma *rev-UNIV*[*simp*]:
  shows {*env*, *self*} = *UNIV*
    and {*self*, *env*} = *UNIV*
by *fastforce*+

lemma *uminus-self-env*[*simp*]:
  shows −{*self*} = {*env*}
by *fastforce*

setup ‹*Sign.parent-path*›

setup ‹*Sign.mandatory-path map-agent*›

lemma *eq-conv*:
  shows *map-agent f x* = *env* ⟷ *x* = *env*
    and *env* = *map-agent f x* ⟷ *x* = *env*
    and *map-agent f x* = *proc a* ⟷ (∃ *a'*. *x* = *proc a'* ∧ *a* = *f a'*)
    and *proc a* = *map-agent f x* ⟷ (∃ *a'*. *x* = *proc a'* ∧ *a* = *f a'*)
by (*cases x*; *auto*)+

lemma *surj*:
  fixes π :: *'a* ⇒ *'b*
  assumes *surj* π
  shows *surj* (*map-agent* π)
  by (*metis assms surj-def agent.exhaust agent.map*(*1*,*2*))

lemma *bij*:
  fixes π :: *'a* ⇒ *'b*
  assumes *bij* π
  shows *bij* (*map-agent* π)
by (*rule bijI*[*OF agent.inj-map*[*OF bij-is-inj*[*OF assms*]] *map-agent.surj*[*OF bij-is-surj*[*OF assms*]]])

setup ‹*Sign.parent-path*›

definition *swap-env-self-fn* :: *sequential* ⇒ *sequential* **where**
  swap-env-self-fn a = (*case a of proc* () ⇒ *env* | *env* ⇒ *self*)

lemma *swap-env-self-fn-simps*:
  shows *swap-env-self-fn self* = *env*
      swap-env-self-fn env = *self*
unfolding *swap-env-self-fn-def* **by** *simp-all*

lemma *bij-swap-env-self-fn*:
  shows *bij swap-env-self-fn*
unfolding *swap-env-self-fn-def bij-def inj-def surj-def* **by** (*auto split*: *agent.split*)

lemma *swap-env-self-fn-vimage-singleton*:
  shows *swap-env-self-fn* −‘ {*env*} = {*self*}
    and *swap-env-self-fn* −‘ {*self*} = {*env*}
unfolding *swap-env-self-fn-def* **by** (*auto split*: *agent.splits*)

setup ‹*Sign.mandatory-path spec*›

abbreviation *swap-env-self* :: (*sequential*, *'s*, *'v*) *spec* ⇒ (*sequential*, *'s*, *'v*) *spec* **where**
swap-env-self ≡ *spec.amap swap-env-self-fn*

setup ‹*Sign.parent-path*›

9.5 Parallel composition

We compose a collection of programs (*sequential*, *'s*, *'v*) *spec* in parallel by mapping these into the (*'a agent*, *'s*, *'v*) *spec* lattice, taking the infimum, and mapping back.

definition *toConcurrent-fn* :: *'a* ⇒ *'a* ⇒ *sequential* **where**
toConcurrent-fn = (λ*a a'*. *if a'* = *a then self else env*)

definition *toSequential-fn* :: *'a agent* ⇒ *sequential* **where**
toSequential-fn = *map-agent* ⟨()⟩

lemma *toSequential-fn-alt-def*:
shows *toSequential-fn* = (λ*x*. *case x of proc x* ⇒ *self* | *env* ⇒ *env*)
by (*simp add*: *toSequential-fn-def fun-eq-iff split*: *agent.split*)

setup ‹*Sign.mandatory-path spec*›

abbreviation *toConcurrent* :: *'a* ⇒ (*sequential*, *'s*, *'v*) *spec* ⇒ (*'a agent*, *'s*, *'v*) *spec* **where**
toConcurrent a ≡ *spec.ainvmap* (*toConcurrent-fn* (*proc a*))

abbreviation *toSequential* :: (*'a agent*, *'s*, *'v*) *spec* ⇒ (*sequential*, *'s*, *'v*) *spec* **where**
toSequential ≡ *spec.amap toSequential-fn*

definition *Parallel* :: *'a set* ⇒ (*'a* ⇒ (*sequential*, *'s*, *unit*) *spec*) ⇒ (*sequential*, *'s*, *unit*) *spec* **where**
Parallel as Ps = *spec.toSequential* (*spec.rel* (*insert env* (*proc* ‘ *as*) × *UNIV*) ⊓ (⨅ *a*∈*as*. *spec.toConcurrent a* (*Ps a*)))

definition *parallel* :: (*sequential*, *'s*, *unit*) *spec* ⇒ (*sequential*, *'s*, *unit*) *spec* ⇒ (*sequential*, *'s*, *unit*) *spec* **where**
parallel P Q = *spec.Parallel UNIV* (λ*a*::*bool*. *if a then P else Q*)

adhoc-overloading
Parallel ⇌ *spec.Parallel*
adhoc-overloading
parallel ⇌ *spec.parallel*

lemma *parallel-alt-def*:
shows *spec.parallel P Q* = *spec.toSequential* (*spec.toConcurrent True P* ⊓ *spec.toConcurrent False Q*)
by (*simp add*: *spec.parallel-def spec.Parallel-def INF-UNIV-bool-expand spec.rel.UNIV*)

setup ‹*Sign.parent-path*›

setup ‹*Sign.mandatory-path toConcurrent-fn*›

lemma *simps*[*simp*]:
shows *toConcurrent-fn* (*proc a*) (*proc a*) = *self*
and *toConcurrent-fn* (*proc a*) *env* = *env*
and *toConcurrent-fn a' a''* = *self* ⟷ *a''* = *a'*
and *self* = *toConcurrent-fn a' a''* ⟷ *a''* = *a'*
and *toConcurrent-fn a' a''* = *env* ⟷ *a''* ≠ *a'*
and *env* = *toConcurrent-fn a' a''* ⟷ *a''* ≠ *a'*
and *toConcurrent-fn* (*proc a*) (*map-agent* ⟨*a*⟩ *x*) = *map-agent* ⟨()⟩ *x*
by (*auto simp*: *toConcurrent-fn-def map-agent.eq-conv intro*: *agent.exhaust*)

lemma *inj-map-agent*:
  assumes *inj-on f* (*insert x* (*set-agent a*))
  shows *toConcurrent-fn* (*proc* (*f x*)) (*map-agent f a*) = *toConcurrent-fn* (*proc x*) *a*
by (*cases a*) (*auto simp*: *toConcurrent-fn-def intro*: *inj-onD*[*OF assms*])

lemma *inv-into-map-agent*:
  fixes *f* :: *'a* ⇒ *'b*
  fixes *a* :: *'b agent*
  fixes *x* :: *'a*
  assumes *inj-on f as*
  assumes *x* ∈ *as*
  assumes *a* ∈ *insert env* ((λ*x*. *proc* (*f x*)) ‘ *as*)
  shows *toConcurrent-fn* (*proc x*) (*map-agent* (*inv-into as f*) *a*) = *toConcurrent-fn* (*proc* (*f x*)) *a*
using *assms* **by** (*auto simp*: *toConcurrent-fn-def*)

lemma *vimage-sequential*[*simp*]:
  shows *toConcurrent-fn* (*proc a*) −‘ {*self*} = {*proc a*}
    and *toConcurrent-fn* (*proc a*) −‘ {*env*} = −{*proc a*}
by (*auto simp*: *toConcurrent-fn-def split*: *if-splits*)

setup ‹*Sign.parent-path*›

setup ‹*Sign.mandatory-path toSequential-fn*›

lemma *simps*[*simp*]:
  shows *toSequential-fn env* = *env*
    and *toSequential-fn* (*proc x*) = *self*
    and *toSequential-fn* (*map-agent f a*) = *toSequential-fn a*
    and *trace.map toSequential-fn id id σ* = *σ*
    and *trace.map toSequential-fn* (λ*x*. *x*) (λ*x*. *x*) *σ* = *σ*
    and (λ*x*. *if x* = *self then self else env*) = *id*
by (*simp-all add*: *toSequential-fn-def fun-unit-id*[**where** *f*=λ*x*. ()] *fun-eq-iff flip*: *id-def*)

lemma *eq-conv*:
  shows *toSequential-fn x* = *env* ⟷ *x* = *env*
    and *toSequential-fn x* = *self* ⟷ (∃ *a*. *x* = *proc a*)
by (*simp-all add*: *toSequential-fn-def map-agent.eq-conv*)

lemma *surj*:
  shows *surj toSequential-fn*
proof −
  have *x* ∈ *range toSequential-fn* **for** *x*
    by (*cases x*)
      (*simp-all add*: *toSequential-fn-def range-eqI*[**where** *x*=*proc undefined*] *range-eqI*[**where** *x*=*env*])
  then show *?thesis* **by** *blast*
qed

lemma *image*[*simp*]:
  assumes *as* ≠ {}
  shows *toSequential-fn* ‘ *proc* ‘ *as* = {*self*}
using *assms* **by** (*auto simp*: *toSequential-fn-def image-image*)

lemma *vimage-sequential*[*simp*]:
  shows *toSequential-fn* −‘ {*env*} = {*env*}
    and *toSequential-fn* −‘ {*self*} = *range proc*
by (*auto simp*: *toSequential-fn-def map-agent.eq-conv*)

setup ‹*Sign.parent-path*›

lemma *toSequential-fn-eq-toConcurrent-fn-conv*:
shows *toSequential-fn a = toConcurrent-fn a′ a″* ⟷ (*case a of env* ⇒ *a″* ≠ *a′* | *proc -* ⇒ *a″* = *a′*)
and *toConcurrent-fn a′ a″ = toSequential-fn a* ⟷ (*case a of env* ⇒ *a″* ≠ *a′* | *proc -* ⇒ *a″* = *a′*)
by (*simp-all split*: *agent.split*)

setup ‹*Sign.mandatory-path spec*›

setup ‹*Sign.mandatory-path toSequential*›

lemma *interference*:
shows *spec.toSequential* (*spec.rel* ({*env*} × *r*)) = *spec.rel* ({*env*} × *r*)
by (*simp add*: *spec.map.rel map-prod-image-Times*)

lemma *interference-inf-toConcurrent*:
fixes *a* :: *′a*
fixes *P* :: (*sequential*, *′s*, *′v*) *spec*
shows *spec.toSequential* (*spec.rel* ({*env*, *proc a*} × *UNIV*) ⊓ *spec.toConcurrent a P*) = *P* (**is** *?lhs* = *?rhs*)
and *spec.toSequential* (*spec.toConcurrent a P* ⊓ *spec.rel* ({*env*, *proc a*} × *UNIV*)) = *P* (**is** *?thesis1*)
proof −
show *?lhs* = *?rhs*
proof(*rule spec.singleton.antisym*)
have ∗: *trace.natural′ s* (*map* (*map-prod toSequential-fn id*) *xs*)
= *trace.natural′ s* (*map* (*map-prod* (*toConcurrent-fn* (*proc a*)) *id*) *xs*)
if *trace.steps′ s xs* ⊆ {*env*, *proc a*} × *UNIV*
for *s* **and** *xs* :: (*′a agent* × *′s*) *list*
using *that* **by** (*induct xs arbitrary*: *s*) *auto*
show ⟪*σ*⟫ ≤ *?rhs* **if** ⟪*σ*⟫ ≤ *?lhs* **for** *σ*
using *that*
by (*force simp*: *spec.singleton.le-conv spec.singleton-le-conv trace.natural-def* ∗
elim: *order.trans*[*rotated*])
show ⟪*σ*⟫ ≤ *?lhs* **if** ⟪*σ*⟫ ≤ *?rhs* **for** *σ*
using *that*
by (*clarsimp intro*!: *exI*[**where** *x*=*trace.map* (*map-agent* ⟨*a*⟩) *id id σ*]
simp: *spec.singleton.le-conv trace.steps′.map map-agent.eq-conv*
fun-unit-id[**where** *f*=*λ*-::*unit*. ()]
simp flip: *id-def*)
qed
then show *?thesis1*
by (*simp add*: *ac-simps*)
qed

setup ‹*Sign.parent-path*›

setup ‹*Sign.mandatory-path toConcurrent*›

lemma *interference*:
shows *spec.toConcurrent a* (*spec.rel* ({*env*} × *UNIV*)) = *spec.rel* ((− {*proc a*}) × *UNIV*)
by (*simp add*: *spec.invmap.rel map-prod-vimage-Times spec.rel.reflcl*)

setup ‹*Sign.parent-path*›

setup ‹*Sign.mandatory-path idle*›

lemma *Parallel-le*[*spec.idle-le*]:
assumes ⋀*a*. *a* ∈ *as* ⟹ *spec.idle* ≤ *Ps a*
shows *spec.idle* ≤ *spec.Parallel as Ps*
apply (*simp add*: *spec.Parallel-def*)

apply (*strengthen ord-to-strengthen*(*2*)[*OF assms*], *assumption*)
apply (*strengthen ord-to-strengthen*(*2*)[*OF spec.idle.invmap-le*[*OF order.refl*]])
apply (*simp add*: *le-INF-iff spec.idle-le*)
done

setup ‹*Sign.parent-path*›

setup ‹*Sign.mandatory-path Parallel*›

lemma *cong*:
  assumes $as = as'$
  assumes $\bigwedge a.\ a \in as' \Longrightarrow Ps\ a = Ps'\ a$
  shows *spec.Parallel as Ps* = *spec.Parallel as' Ps'*
unfolding *spec.Parallel-def* **using** *assms* **by** *simp*

lemma *no-agents*:
  shows *spec.Parallel* {} *Ps* = *spec.rel* ({*env*} × *UNIV*)
by (*simp add*: *spec.Parallel-def spec.toSequential.interference*)

lemma *singleton-agents*:
  shows *spec.Parallel* {*a*} *Ps* = *Ps a*
by (*simp add*: *spec.Parallel-def spec.toSequential.interference-inf-toConcurrent*)

lemma *bot*:
  assumes $Ps\ a = \bot$
  assumes $a \in as$
  shows *spec.Parallel as Ps* = ⊥
by (*simp add*: *spec.Parallel-def assms INF-unwind-index*[*of a*] *spec.invmap.bot spec.map.bot*)

lemma *top*:
  shows *spec.Parallel as* ⊤ = (*if as* = {} *then spec.rel* ({*env*} × *UNIV*) *else* ⊤)
proof −
  have *spec.toSequential* (*spec.rel* (*insert env* (*proc* ‘ *as*) × *UNIV*)) = ⊤ **if** *as* ≠ {}
    using *that* **by** (*subst spec.map.rel*, *force*, *simp add*: *map-prod-image-Times flip*: *spec.rel.UNIV*)
  then show *?thesis*
    by (*simp add*: *spec.Parallel.no-agents*) (*auto simp*: *spec.Parallel-def spec.invmap.top*)
qed

lemma *mono*:
  assumes $\bigwedge a.\ a \in as \Longrightarrow Ps\ a \leq Ps'\ a$
  shows *spec.Parallel as Ps* ≤ *spec.Parallel as Ps'*
unfolding *spec.Parallel-def* **by** (*strengthen ord-to-strengthen*(*1*)[*OF assms*(*1*)]; *simp*)

lemma *strengthen*[*strg*]:
  assumes $\bigwedge a.\ a \in as \Longrightarrow$ *st-ord F* (*Ps a*) (*Ps' a*)
  shows *st-ord F* (*spec.Parallel as Ps*) (*spec.Parallel as Ps'*)
using *assms* **by** (*cases F*) (*auto simp*: *spec.Parallel.mono*)

lemma *mono2mono*[*cont-intro*, *partial-function-mono*]:
  fixes *Ps* :: $'a \Rightarrow 'b \Rightarrow$ (*sequential*, $'s$, *unit*) *spec*
  assumes $\bigwedge a.\ a \in as \Longrightarrow$ *monotone orda* (≤) (*Ps a*)
  shows *monotone orda* (≤) (λ*x*::$'b$. *spec.Parallel as* (λ*a*. *Ps a x*))
using *spec.Parallel.mono assms* **unfolding** *monotone-def* **by** *meson*

lemma *invmap*: — *af* = *id* in *spec.invmap*
  shows *spec.invmap id sf vf* (*spec.Parallel UNIV Ps*) = *spec.Parallel UNIV* (*spec.invmap id sf vf* ∘ *Ps*)

by (*simp add*: *spec.Parallel-def image-image spec.invmap.inf spec.invmap.Inf spec.invmap.comp spec.rel.UNIV*
flip: *spec.amap.surj-invmap*[*OF toSequential-fn.surj*])

lemma *discard-interference*:
assumes $\bigwedge a.\ a \in bs \Longrightarrow Ps\ a = spec.rel\ (\{env\} \times UNIV)$
shows *spec.Parallel as Ps* = *spec.Parallel* (*as* − *bs*) *Ps*
proof −
have ∗: *as* = (*as* − *bs*) ∪ (*as* ∩ *bs*) **by** *blast*
have ∗∗: (*insert env* (*proc* ‘ *as*) ∩ − *proc* ‘ (*as* ∩ *bs*)) = *insert env* (*proc* ‘ (*as* − *bs*)) **by** *blast*
from *assms* **have** ∗∗∗: ($\bigsqcap a{\in}as \cap bs.$ *spec.toConcurrent a* (*Ps a*))
= *spec.rel* ((− *proc* ‘ (*as* ∩ *bs*)) × *UNIV*)
by (*force simp*: *assms spec.toConcurrent.interference le-Inf-iff*
simp flip: *spec.rel.INF*
intro: *spec.rel.mono antisym*)
show *?thesis*
apply (*simp add*: *spec.Parallel-def*)
apply (*subst* (*2*) ∗)
apply (*simp add*: *image-Un Inf-union-distrib ac-simps* ∗∗ ∗∗∗ *Times-Int-Times*
flip: *spec.rel.inf inf.assoc*)
done
qed

lemma *rename-UNIV-aux*:
fixes *f* :: *′a* ⇒ *′b*
assumes *inj-on f as*
shows *spec.toSequential* (*spec.rel* (*insert env* (*proc* ‘ *as*) × *UNIV*)
⊓ ($\bigsqcap a{\in}as.$ *spec.toConcurrent a* (*Ps a*)))
= *spec.toSequential* (*spec.rel* (*insert env* (*proc* ‘ *f* ‘ *as*) × *UNIV*)
⊓ ($\bigsqcap a{\in}as.$ *spec.toConcurrent* (*f a*) (*Ps a*))) (**is** *?lhs* = *?rhs*)
proof(*rule spec.singleton.antisym*)
show ⟪*σ*⟫ ≤ *?rhs* **if** ⟪*σ*⟫ ≤ *?lhs* **for** *σ*
using *that assms*
apply (*clarsimp simp*: *spec.singleton.le-conv le-Inf-iff*)
apply (*rule exI*[**where** *x*=*trace.map* (*map-agent f*) *id id σ* **for** *σ*])
apply (*intro conjI*)
apply (*fastforce simp*: *trace.steps′.map*)
apply (*fastforce intro*: *ord-eq-le-trans*[*OF spec.singleton.map-cong*[*OF toConcurrent-fn.inj-map-agent refl refl refl*]]
dest: *inj-onD trace.steps′.asetD*
simp flip: *id-def*)
apply (*fastforce simp flip*: *id-def*)
done
show ⟪*σ*⟫ ≤ *?lhs* **if** ⟪*σ*⟫ ≤ *?rhs* **for** *σ*
using *that assms*
apply (*clarsimp simp*: *spec.singleton.le-conv le-Inf-iff image-image*)
apply (*rule exI*[**where** *x*=*trace.map* (*map-agent* (*inv-into as f*)) *id id σ* **for** *σ*])
apply (*auto 4 2 dest*: *trace.steps′.asetD*
simp: *spec.singleton.map-cong*[*OF toConcurrent-fn.inv-into-map-agent refl refl refl*]
comp-def trace.steps′.map
simp flip: *id-def*)
done
qed

lemma *rename-UNIV*: — expand the set of agents to *UNIV*
fixes *f* :: *′a* ⇒ *′b*
assumes *inj-on f as*
shows *spec.Parallel as Ps*
= *spec.Parallel* (*UNIV* :: *′b set*)

```
                    (λb. if b ∈ f ‘ as then Ps (inv-into as f b) else spec.rel ({env} × UNIV))
(is ?lhs = spec.Parallel - ?f)
proof −
  have ∗: (⨅x. spec.toConcurrent x (?f x))
        = spec.rel (insert env (proc ‘ f ‘ as) × UNIV)
          ⊓ (⨅x∈f ‘ as. spec.toConcurrent x (Ps (inv-into as f x)))
  proof −
    have ∗: (⋂x∈− f ‘ as. (− {proc x}) × UNIV) = insert env (proc ‘ f ‘ as) × UNIV
      by (auto intro: agent.exhaust)
    have (⨅x. spec.toConcurrent x (?f x))
       = (⨅x∈f ‘ as. spec.toConcurrent x (?f x)) ⊓ (⨅x∈−f ‘ as. spec.toConcurrent x (?f x))
      by (subst INF-union[symmetric]) simp
    also have ... = spec.rel (insert env (proc ‘ f ‘ as) × UNIV)
                    ⊓ (⨅x∈f ‘ as. spec.toConcurrent x (Ps (inv-into as f x)))
      by (simp add: ac-simps spec.invmap.rel map-prod-vimage-Times spec.rel.reflcl ∗
            flip: spec.rel.upper-INF)
    finally show ?thesis .
  qed
  show ?thesis
    by (simp add: spec.Parallel-def ∗ inv-into-f-f[OF assms] spec.rel.UNIV
                  INF-rename-bij[OF inj-on-imp-bij-betw[OF assms],
                                 where F=λ- x. spec.toConcurrent x (Ps (inv-into as f x))]
                  spec.Parallel.rename-UNIV-aux[OF assms])
qed

lemma rename:
  fixes π :: 'a ⇒ 'b
  fixes Ps :: 'b ⇒ (sequential, 's, unit) spec
  assumes bij-betw π as bs
  shows spec.Parallel as (Ps ∘ π) = spec.Parallel bs Ps
proof −
  define π' where π' = (λx::'a+'b. case x of
                  Inl a ⇒ if a ∈ as then Inr (π a) else Inl a
                | Inr b ⇒ if b ∈ bs then Inl (inv-into as π b) else Inr b)
  from assms have inj π'
    by (force intro: injI
              simp: π'-def bij-betw-apply bij-betw-imp-surj-on inv-into-into
              split: sum.split-asm if-split-asm
              dest: bij-betw-inv-into-left[rotated] bij-betw-inv-into-right[rotated])
  have simps: ⋀a. π' (Inl a) = (if a ∈ as then Inr (π a) else Inl a)
              ⋀b. π' (Inr b) = (if b ∈ bs then Inl (inv-into as π b) else Inr b)
    by (simp-all add: π'-def)
  have inv-simps: ⋀a. a ∈ as ⟹ inv π' (Inl a) = Inr (π a)
    by (simp add: inv-f-eq[OF ‹inj π'›] bij-betw-inv-into-left[OF assms] bij-betw-apply[OF assms] simps(2))
  show ?thesis
    apply (simp add: spec.Parallel.rename-UNIV[where as=as and f=Inl :: 'a ⇒ 'a + 'b]
                     spec.Parallel.rename-UNIV[where as=bs and f=Inr :: 'b ⇒ 'a + 'b] comp-def)
    apply (subst (2) spec.Parallel.rename-UNIV[where as=UNIV, OF ‹inj π'›])
    apply (fastforce intro: arg-cong[where f=spec.Parallel UNIV]
                     simp: fun-eq-iff split-sum-all image-iff simps inv-simps
                     inv-f-f[OF ‹inj π'›]  bij-betw-apply[OF bij-betw-inv-into[OF assms]]
                     bij-betw-apply[OF assms] bij-betw-inv-into-left[OF assms])
    done
qed

lemma rename-cong:
  fixes π :: 'a ⇒ 'b
  fixes Ps :: 'a ⇒ (-, -, -) spec
```

fixes $Ps' :: 'b \Rightarrow (-, -, -)\ spec$
assumes *bij-betw* π *as bs*
assumes $\bigwedge a.\ a \in as \Longrightarrow Ps\ a = Ps'\ (\pi\ a)$
shows *spec.Parallel as Ps = spec.Parallel bs Ps'*
by (*simp add*: *assms*(*2*) *flip*: *spec.Parallel.rename*[*OF assms*(*1*)] *cong*: *spec.Parallel.cong*)

lemma *inf-pre*:
assumes $as \neq \{\}$
shows *spec.Parallel as Ps* $\sqcap$ *spec.pre P* = ($\|i{\in}as.\ Ps\ i \sqcap spec.pre\ P$) (**is** *?thesis1*)
and *spec.pre P* $\sqcap$ *spec.Parallel as Ps* = ($\|i{\in}as.\ spec.pre\ P \sqcap Ps\ i$) (**is** *?thesis2*)
proof −
show *?thesis1*
by (*simp add*: *spec.Parallel-def assms spec.invmap.inf spec.invmap.pre spec.map.inf-distr*
inf.assoc INF-inf-const2)
then show *?thesis2*
by (*simp add*: *ac-simps*)
qed

lemma *inf-post*:
assumes $as \neq \{\}$
shows *spec.Parallel as Ps* $\sqcap$ *spec.post Q* = *spec.Parallel as* ($\lambda i.\ Ps\ i \sqcap spec.post\ Q$) (**is** *?thesis1*)
and *spec.post Q* $\sqcap$ *spec.Parallel as Ps* = *spec.Parallel as* ($\lambda i.\ spec.post\ Q \sqcap Ps\ i$) (**is** *?thesis2*)
proof −
show *?thesis1*
by (*simp add*: *spec.Parallel-def assms spec.invmap.inf spec.invmap.post spec.map.inf-distr*
inf.assoc INF-inf-const2)
then show *?thesis2*
by (*simp add*: *ac-simps*)
qed

lemma *unwind*:
— All other processes begin with interference
assumes *b*: $\bigwedge b.\ b \in as - \{a\} \Longrightarrow spec.rel\ (\{env\} \times UNIV) \gg\!= (\lambda\text{-}::unit.\ Ps\ b) \leq Ps\ b$
assumes *a*: $f \gg\!= g \leq Ps\ a$ — The selected process starts with f
assumes $a \in as$
shows $f \gg\!= (\lambda v.\ spec.Parallel\ as\ (Ps(a := g\ v))) \leq spec.Parallel\ as\ Ps$
proof −
have ∗: *spec.toConcurrent a f* $\sqcap$ *spec.rel* ($\bigcap x{\in}as - \{a\}.\ (-\ \{proc\ x\}) \times UNIV$)
$\gg\!= (\lambda v.\ \prod b{\in}as.\ spec.toConcurrent\ b\ ((Ps(a{:=}g\ v))\ b))$
$\leq$ ($\prod a{\in}as.\ spec.toConcurrent\ a\ (Ps\ a)$) (**is** *?lhs* $\leq$ *?rhs*)
proof −
from ‹$a \in as$›
have *?lhs* = *spec.toConcurrent a f* $\sqcap$ *spec.rel* ($\bigcap x{\in}as - \{a\}.\ (-\ \{proc\ x\}) \times UNIV$)
$\gg\!= (\lambda v.\ spec.toConcurrent\ a\ (g\ v) \sqcap (\prod b{\in}as - \{a\}.\ spec.toConcurrent\ b\ (Ps\ b)))$
by (*simp add*: *INF-unwind-index*)
also have ... $\leq$ (*spec.toConcurrent a f* $\gg\!=$ ($\lambda x.\ spec.toConcurrent\ a\ (g\ x)$))
$\sqcap$ (*spec.rel* ($\bigcap x{\in}as - \{a\}.\ (-\ \{proc\ x\}) \times UNIV$)
$\gg\!= (\lambda\text{-}::unit.\ \prod b{\in}as - \{a\}.\ spec.toConcurrent\ b\ (Ps\ b))$)
by (*strengthen ord-to-strengthen*(*2*)[*OF spec.bind.inf-rel-distr-le*]) *simp*
also have ... = (*spec.toConcurrent a f* $\gg\!=$ ($\lambda x.\ spec.toConcurrent\ a\ (g\ x)$))
$\sqcap$ (($\prod b{\in}as - \{a\}.\ spec.toConcurrent\ b\ (spec.rel\ (\{env\} \times UNIV))$)
$\gg\!= (\lambda\text{-}::unit.\ \prod b{\in}as - \{a\}.\ spec.toConcurrent\ b\ (Ps\ b))$)
by (*simp add*: *spec.invmap.rel map-prod-vimage-Times spec.rel.reflcl flip*: *spec.rel.INF*)
also have ... $\leq$ (*spec.toConcurrent a f* $\gg\!=$ ($\lambda x.\ spec.toConcurrent\ a\ (g\ x)$))
$\sqcap$ ($\prod b{\in}as - \{a\}.\ spec.toConcurrent\ b\ (spec.rel\ (\{env\} \times UNIV))$
$\gg\!= (\lambda\text{-}::unit.\ spec.toConcurrent\ b\ (Ps\ b))$)
by (*strengthen ord-to-strengthen*(*2*)[*OF spec.bind.Inf-le*]) *simp*
also have ... = *spec.toConcurrent a* ($f \gg\!= g$)

```
                ⊓ (⨅ b∈as − {a}. spec.toConcurrent b (spec.rel ({env} × UNIV) ≫= (λ-::unit. Ps b)))
    by (simp add: spec.invmap.bind)
   also have ... ≤ spec.toConcurrent a (Ps a) ⊓ (⨅ b∈as − {a}. spec.toConcurrent b (Ps b))
    by (strengthen ord-to-strengthen(2)[OF a], strengthen ord-to-strengthen(2)[OF b], assumption, rule order.refl)
   also from ‹a ∈ as› have ... = ?rhs by (simp add: INF-unwind-index)
   finally show ?thesis .
  qed
  from ‹a ∈ as›
  have **: (insert env (proc ‘ as) × UNIV ∩ (⋂ x∈as − {a}. (− {proc x}) × UNIV)) = {env, proc a} × UNIV
   by blast
  show ?thesis
   unfolding spec.Parallel-def
   by (strengthen ord-to-strengthen(2)[OF *])
     (simp add: ac-simps spec.bind.inf-rel spec.map.bind-inj-sf **
           spec.toSequential.interference-inf-toConcurrent
       flip: spec.rel.inf)
qed

lemma inf-rel:
  fixes as :: 'a set
  fixes r :: 's rel
  shows spec.rel ({env} × UNIV ∪ {self} × r) ⊓ spec.Parallel as Ps
      = spec.Parallel as (λa. spec.rel ({env} × UNIV ∪ {self} × r) ⊓ Ps a) (is ?lhs = ?rhs)
   and spec.Parallel as Ps ⊓ spec.rel ({env} × UNIV ∪ {self} × r)
      = spec.Parallel as (λa. Ps a ⊓ spec.rel ({env} × UNIV ∪ {self} × r)) (is ?thesis1)
proof −
  show ?lhs = ?rhs
  proof(cases as = {})
   case True then show ?thesis
    by (simp add: spec.Parallel.no-agents flip: spec.rel.inf)
  next
   case False show ?thesis
  proof(rule antisym)
   have *: insert env (proc ‘ as) × UNIV ∩ map-prod toSequential-fn id −‘ (UNIV × Id ∪ ({env} × UNIV ∪
{self} × r))
        ⊆ insert env (proc ‘ as) × UNIV ∩ map-prod (toConcurrent-fn (proc a)) id −‘ (UNIV × Id ∪ (({env}
× UNIV) ∪ {self} × r)) for a
    by auto
   from False
   show ?lhs ≤ ?rhs
    apply (simp add: spec.Parallel-def ac-simps spec.map.inf-rel
          flip: spec.rel.inf spec.invmap.inf-rel INF-inf-const1 INF-inf-const2
           del: vimage-Un)
    apply (strengthen ord-to-strengthen(1)[OF *])
    apply (rule order.refl)
    done
   have spec.toSequential (⨅ x∈as. spec.toConcurrent x (Ps x) ⊓ spec.rel (insert env (proc ‘ as) × UNIV ∩
map-prod (toConcurrent-fn (proc x)) id −‘ (UNIV × Id ∪ ({env} × UNIV ∪ {self} × r))))
      ≤ spec.toSequential (⨅ x∈as. spec.toConcurrent x (Ps x) ⊓ spec.rel (insert env (proc ‘ as) × UNIV ∩
map-prod toSequential-fn id −‘ (UNIV × Id ∪ ({env} × UNIV ∪ {self} × r))))
    apply (rule spec.singleton-le-extI)
    apply (clarsimp simp: spec.singleton.le-conv le-INF-iff)
    apply (rename-tac σ σ')
    apply (rule-tac x=σ' in exI)
    apply (clarsimp simp: toConcurrent-fn-def toSequential-fn-def trace.split-all)
    apply (rename-tac σ σ' a s s' a')
    apply (case-tac a)
     apply (case-tac the-agent a ∈ as; force)
```

```
    apply simp
    done
  with False
  show ?rhs ≤ ?lhs
    by (simp add: spec.Parallel-def ac-simps spec.map.inf-rel
        flip: INF-inf-const1 INF-inf-const2 spec.invmap.inf-rel spec.rel.inf)
  qed
 qed
 then show ?thesis1
  by (simp add: ac-simps)
qed

lemma flatten:
 fixes as :: 'a set
 fixes a :: 'a
 fixes bs :: 'b set
 fixes Ps :: 'a ⇒ ('sequential, 's, unit) spec
 fixes Ps' :: 'b ⇒ ('sequential, 's, unit) spec
 assumes Ps a = spec.Parallel bs Ps'
 assumes a ∈ as
 shows spec.Parallel as Ps = spec.Parallel ((as − {a}) <+> bs) (case-sum Ps Ps') (is ?lhs = ?rhs)
proof(rule spec.singleton.antisym)
 have simps:
  ⋀a'. a' ≠ a ⟹ (λx::('a + 'b) agent. toConcurrent-fn (proc a') (case x of proc (Inl a) ⇒ proc a | proc (Inr -)
⇒ proc a | env ⇒ env)) = toConcurrent-fn (proc (Inl a'))
  ⋀a'.          (λx::('a + 'b) agent. toConcurrent-fn (proc a') (case x of proc (Inl -) ⇒ env | proc (Inr a) ⇒
proc a | env ⇒ env)) = toConcurrent-fn (proc (Inr a'))
  by (auto simp: fun-eq-iff toConcurrent-fn-def split: agent.split sum.split)
 have ∗:
  ∃σ''' :: (('a + 'b) agent, 's, unit) trace.t.
       ⟪σ'⟫ = ⟪trace.map (case-agent (case-sum proc ⟨proc a⟩) env) id id σ'''⟫
     ∧ ⟪trace.map (case-agent (case-sum ⟨env⟩ proc) env) id id σ'''⟫ ≤ ⟪σ''⟫
     ∧ proc (Inl a) ∉ trace.aset (♮σ''')
   if ⟪trace.map (toConcurrent-fn (proc a)) id id σ'⟫ ≤ ⟪trace.map toSequential-fn id id σ''⟫
   for σ'  :: ('a agent, 's, unit) trace.t
   and σ'' :: ('b agent, 's, unit) trace.t
 proof(cases trace.term σ')
  case None
  have ∃ zs :: (('a + 'b) agent × 's) list.
          xs = map (map-prod (case-agent (case-sum proc (λs. proc a)) env) id) zs
        ∧ prefix (map (map-prod (case-agent (case-sum (λs. env) proc) env) id) zs) ys
        ∧ (proc (Inl a) ∉ fst ' set zs) (is ∃ zs. ?goal xs zs)
   if prefix (map (map-prod (toConcurrent-fn (proc a)) id) xs) (map (map-prod toSequential-fn id) ys)
   for xs :: ('a agent × 's) list and ys :: ('b agent × 's) list
   using that
  proof(induct xs rule: rev-induct)
   case (snoc x xs)
   then obtain zs where ?goal xs zs by (auto dest: append-prefixD)
   with snoc.prems show ?case
    apply (clarsimp simp: map-prod.comp map-prod-conv simp flip: id-def elim!: prefixE)
    subgoal for a_y
     by (rule exI[where x=zs @ [(if fst x = proc a then map-agent Inr a_y else map-agent Inl (fst x), (snd
x))]])
        (auto simp: toSequential-fn.eq-conv map-agent.eq-conv simp flip: all-simps split: agent.split)
    done
  qed simp
  from this[of (trace.natural' (trace.init σ') (trace.rest σ'))
               trace.natural' (trace.init σ') (trace.rest σ'')] that None
```

```
  show ?thesis
    apply (simp add: spec.singleton-le-conv trace.natural.map-inj-on-sf)
    apply (clarsimp simp: trace.natural-def trace.aset.simps trace.split-Ex image-iff trace.less-eq-None)
    subgoal for zs
      by (clarsimp dest!: trace.natural'.amap-noop intro!: exI[where x=zs])
    done
next
  case (Some v)
  have *: ∃ zs :: (('a + 'b) agent × 's) list.
            xs = map (map-prod (case-agent (case-sum proc (λs. proc a)) env) id) zs
          ∧ ys = map (map-prod (case-agent (case-sum (λs. env) proc) env) id) zs
          ∧ (proc (Inl a) ∉ fst ` set zs)
    if map (map-prod (toConcurrent-fn (proc a)) id) xs = map (map-prod toSequential-fn id) ys
   for xs :: ('a agent × 's) list and ys :: ('b agent × 's) list
  proof −
    from that have length xs = length ys
      using map-eq-imp-length-eq by blast
    from this that show ?thesis
    proof(induct rule: list-induct2)
      case (Cons x xs y ys) then show ?case
        by (cases x, cases y, cases fst x)
           (auto 8 0 simp: Cons-eq-map-conv comp-def toSequential-fn-eq-toConcurrent-fn-conv
                 simp flip: id-def ex-simps
                     split: if-splits agent.splits sum.split)
    qed simp
  qed
  from that Some
      *[where xs=trace.natural' (trace.init σ') (trace.rest σ')
         and ys=trace.natural' (trace.init σ') (trace.rest σ'')]
  show ?thesis
    apply (simp add: spec.singleton-le-conv trace.natural.map-inj-on-sf)
    apply (clarsimp simp: trace.natural-def)
    subgoal for zs
      by (clarsimp simp: trace.split-Ex trace.aset.simps
                  dest!: trace.natural'.amap-noop
                 intro!: exI[where x=zs])
    done
qed
{
  fix σ
  assume ⟪σ⟫ ≤ ?lhs
  then obtain σa σb σc
    where 1: trace.steps' (trace.init σa) (trace.rest σa) ⊆ insert env (proc ` as) × UNIV
      and 2: ∀ x∈as. ⟪trace.map (toConcurrent-fn (proc x)) id id σa⟫ ≤ Ps x
      and 3: ⟪σ⟫ ≤ ⟪trace.map toSequential-fn id id σa⟫
      and 4: trace.steps' (trace.init σb) (trace.rest σb) ⊆ insert env (proc ` bs) × UNIV
      and 5: ∀ x∈bs. ⟪trace.map (toConcurrent-fn (proc x)) id id σb⟫ ≤ Ps' x
      and 6: σa ≃S trace.map (case-agent (case-sum proc ⟨proc a⟩) env) id id σc
      and 7: ⟪trace.map (case-agent (case-sum ⟨env⟩ proc) env) id id σc⟫ ≤ ⟪σb⟫
      and 8: proc (Inl a) ∉ trace.aset (♮σc)
    apply (clarsimp simp: spec.Parallel-def spec.singleton.le-conv le-Inf-iff)
    apply (frule bspec[OF - ‹a ∈ as›])
    apply (clarsimp simp: assms(1) spec.Parallel-def spec.singleton.le-conv le-Inf-iff dest!: *)
    done
  show ⟪σ⟫ ≤ ?rhs
    unfolding spec.Parallel-def spec.singleton.le-conv inf.bounded-iff le-Inf-iff ball-simps
  proof(intro exI[where x=σc] conjI ballI)
    show trace.steps' (trace.init σc) (trace.rest σc)
```

```
      ⊆ insert env (proc ‘ ((as − {a}) <+> bs)) × UNIV (is ?lhs ⊆ ?rhs)
    proof(rule subsetI, unfold split-paired-all)
     show (x, s, s′) ∈ ?rhs if (x, s, s′) ∈ ?lhs for x s s′
     proof(cases x)
      case (proc y) then show ?thesis
      proof(cases y)
       case (Inl a)
       with that proc 1 arg-cong[OF 6, where f=trace.steps] 8
       show ?thesis
        by (fastforce simp: trace.steps′.natural′ trace.steps′.map trace.aset.natural-conv)
      next
       case (Inr b)
       with that proc 4 spec.steps.mono[OF 7]
       show ?thesis
        by (fastforce simp: trace.steps′.natural′ trace.steps′.map spec.steps.singleton)
      qed
     qed simp
    qed
    show ⟪trace.map (toConcurrent-fn (proc x)) id id σc⟫ ≤ (case x of Inl x ⇒ Ps x | Inr x ⇒ Ps′ x)
     if x ∈ (as − {a}) <+> bs
     for x
    proof(cases x)
     case (Inl l) with 2 6 that show ?thesis
      by (force dest: trace.stuttering.equiv.map[where af=toConcurrent-fn (proc l) and sf=id and vf=id]
            simp: sum-In-conv simps fun-unit-id[where f=λ-::unit. ()]
          simp flip: id-def
            cong: spec.singleton-cong)
    next
     case (Inr r) with 2 5 6 7 that show ?thesis
      by (fastforce dest: spec.singleton.map-le[where af=toConcurrent-fn (proc r) and sf=id and vf=id]
            simp: simps fun-unit-id[where f=λ-::unit. ()]
          simp flip: id-def
            cong: spec.singleton-cong)
    qed
    from 3 6
    show ⟪σ⟫ ≤ ⟪trace.map toSequential-fn id id σc⟫
     by (fastforce dest: trace.stuttering.equiv.map[where af=toSequential-fn and sf=id and vf=id]
           simp: spec.singleton-le-conv id-def agent.case-distrib sum.case-distrib
         simp flip: toSequential-fn-alt-def
           cong: agent.case-cong sum.case-cong)
   qed
  }
  {
   fix σ
   assume ⟪σ⟫ ≤ ?rhs
   then obtain σ′
    where 1: trace.steps′ (trace.init σ′) (trace.rest σ′) ⊆ insert env (proc ‘ ((as − {a}) <+> bs)) × UNIV
     and 2: ∀x∈(as − {a}) <+> bs. ⟪trace.map (toConcurrent-fn (proc x)) id id σ′⟫ ≤ (case x of Inl x ⇒ Ps
x | Inr x ⇒ Ps′ x)
     and 3: ⟪σ⟫ ≤ ⟪trace.map toSequential-fn id id σ′⟫
    by (clarsimp simp: spec.Parallel-def spec.singleton.le-conv le-Inf-iff)
   show ⟪σ⟫ ≤ ?lhs
   proof −
    from ‹a ∈ as› 1
    have trace.steps′ (trace.init σ′) (map (map-prod (case-agent (case-sum proc ⟨proc a⟩) env) id) (trace.rest
σ′))
      ⊆ insert env (proc ‘ as) × UNIV
     by (auto simp: trace.steps′.map)
```

```
    moreover
    have ⦇trace.map (λy. toConcurrent-fn (proc x) (case y of proc (Inl a) ⇒ proc a | proc (Inr b) ⇒ proc a |
env ⇒ env)) id id σ'⦈ ≤ Ps x
      if x ∈ as
     for x
    proof(cases x = a)
      case True show ?thesis
      proof −
        from ‹a ∈ as› 1
        have trace.steps' (trace.init σ') (map (map-prod (case-agent (case-sum ⟨env⟩ proc) env) id) (trace.rest
σ'))
          ⊆ insert env (proc ‘ bs) × UNIV
          by (auto simp: trace.steps'.map)
        moreover
        from 2
        have ⦇trace.map (λy. toConcurrent-fn (proc b) (case y of proc (Inl a) ⇒ env | proc (Inr b) ⇒ proc b |
env ⇒ env)) id id σ'⦈ ≤ Ps' b
          if b ∈ bs
         for b
          using that by (fastforce simp: simps dest: bspec[where x=Inr b])
        moreover
        from 1
        have ⦇trace.map (λy. toConcurrent-fn (proc a) (case y of proc (Inl a) ⇒ proc a | proc (Inr b) ⇒ proc a |
env ⇒ env)) id id σ'⦈
          ≤ ⦇trace.map (λy. toSequential-fn (case y of proc (Inl a) ⇒ env | proc (Inr b) ⇒ proc b | env ⇒ env))
id id σ'⦈
          by (subst spec.singleton.map-cong[where af'=λy. toSequential-fn (case y of proc (Inl s) ⇒ env | proc
(Inr b) ⇒ proc b | env ⇒ env), OF - refl refl refl];
              fastforce simp: trace.aset.natural-conv split: agent.split sum.split)
        ultimately show ?thesis
          apply (simp add: ‹x = a› assms(1) spec.Parallel-def spec.singleton.le-conv le-Inf-iff)
          apply (rule exI[where x=trace.map (case-agent (case-sum ⟨env⟩ proc) env) id id σ'])
          apply (intro conjI ballI)
          apply (simp-all add: fun-unit-id[where f=λ-::unit. ()] flip: id-def)
          done
      qed
    next
      case False with 2 that show ?thesis
        by (fastforce simp: simps dest: bspec[where x=Inl x])
    qed
    moreover
    from 3
    have ⦇σ⦈ ≤ ⦇trace.map (λx. toSequential-fn (case x of proc (Inl a) ⇒ proc a | proc (Inr b) ⇒ proc a | env
⇒ env)) id id σ'⦈
      by (simp add: agent.case-distrib sum.case-distrib
            flip: toSequential-fn-alt-def
            cong: agent.case-cong sum.case-cong)
    ultimately show ?thesis
      apply (simp add: spec.Parallel-def spec.singleton.le-conv le-Inf-iff)
      apply (rule exI[where x=trace.map (case-agent (case-sum proc ⟨proc a⟩) env) id id σ'] conjI ballI)
      apply (simp add: fun-unit-id[where f=λ-::unit. ()] flip: id-def)
      done
  qed
 }
qed

setup ‹Sign.parent-path›
```

setup ‹*Sign.mandatory-path term*›

setup ‹*Sign.mandatory-path none*›

lemma *Parallel-some-agents*:
assumes $\bigwedge a.\ a \in bs \Longrightarrow Ps\ a = spec.term.none\ (Ps'\ a)$
assumes $as \cap bs \neq \{\}$
shows $spec.Parallel\ as\ Ps = spec.term.none\ (\|\|a{\in}as.\ if\ a \in as \cap bs\ then\ Ps'\ a\ else\ Ps\ a)$
using *assms*(*1*)[*symmetric*] *assms*(*2*)
INF-union[**where** $A{=}as - bs$ **and** $B{=}as \cap bs$
and $M{=}\lambda a.\ spec.toConcurrent\ a\ (if\ a \in bs\ then\ Ps'\ a\ else\ Ps\ a)$]
by (*simp add*: *spec.Parallel-def Un-Diff-Int inf-assoc image-image*
spec.term.none.map spec.term.none.invmap spec.term.none.inf-unit(*2*) *spec.term.none.Inf-not-empty*
flip: *INF-union*)

lemma *Parallel-not-empty*:
assumes $as \neq \{\}$
shows $spec.term.none\ (Parallel\ as\ Ps) = Parallel\ as\ (spec.term.none \circ Ps)$
using *assms spec.term.none.Parallel-some-agents*[**where** $as{=}as$ **and** $bs{=}as$ **and** $Ps{=}spec.term.none \circ Ps$ **and** $Ps'{=}Ps$]
by (*simp cong*: *spec.Parallel.cong*)

lemma *parallel*:
shows $spec.term.none\ (P \parallel Q) = spec.term.none\ P \parallel spec.term.none\ Q$
by (*simp add*: *spec.parallel-def spec.term.none.Parallel-not-empty comp-def if-distrib*)

lemma
shows *parallelL*: $spec.term.none\ P \parallel Q = spec.term.none\ (P \parallel Q)$
and *parallelR*: $P \parallel spec.term.none\ Q = spec.term.none\ (P \parallel Q)$
using
spec.term.none.Parallel-some-agents[**where**
$as{=}UNIV$ **and** $bs{=}\{True\}$
and $Ps{=}\lambda a.\ if\ a\ then\ spec.term.none\ P\ else\ Q$ **and** $Ps'{=}\lambda a.\ if\ a\ then\ P\ else\ Q$]
spec.term.none.Parallel-some-agents[**where**
$as{=}UNIV$ **and** $bs{=}\{False\}$
and $Ps{=}\lambda a.\ if\ a\ then\ P\ else\ spec.term.none\ Q$ **and** $Ps'{=}\lambda a.\ if\ a\ then\ P\ else\ Q$]
by (*simp-all add*: *spec.parallel-def cong*: *if-cong*)

setup ‹*Sign.parent-path*›

setup ‹*Sign.mandatory-path all*›

lemma *Parallel*:
shows $spec.term.all\ (spec.Parallel\ as\ Ps) = spec.Parallel\ as\ (spec.term.all \circ Ps)$
by (*simp add*: *spec.Parallel-def image-image*
spec.term.all.Inf spec.term.all.inf spec.term.all.invmap spec.term.all.map spec.term.all.rel)

setup ‹*Sign.parent-path*›

setup ‹*Sign.parent-path*›

setup ‹*Sign.mandatory-path idle*›

lemma *parallel-le*:
assumes $spec.idle \leq P$
assumes $spec.idle \leq Q$
shows $spec.idle \leq P \parallel Q$
by (*simp add*: *assms spec.parallel-def spec.idle-le*)

setup ‹*Sign.parent-path*›

setup ‹*Sign.mandatory-path invmap*›

lemma *parallel*: — *af* = *id* in *spec.invmap*
  shows *spec.invmap id sf vf* (*spec.parallel P Q*)
      = *spec.parallel* (*spec.invmap id sf vf P*) (*spec.invmap id sf vf Q*)
by (*simp add*: *spec.parallel-def spec.Parallel.invmap comp-def if-distrib*)

setup ‹*Sign.parent-path*›

setup ‹*Sign.mandatory-path parallel*›

lemma *bot*:
  shows *botL*: *spec.parallel* ⊥ *P* = ⊥
    and *botR*: *spec.parallel P* ⊥ = ⊥
by (*simp-all add*: *spec.parallel-def spec.Parallel.bot*[**where** *a*=*False*] *spec.Parallel.bot*[**where** *a*=*True*])

lemma *commute*:
  shows *spec.parallel P Q* = *spec.parallel Q P*
unfolding *spec.parallel-def* **by** (*subst spec.Parallel.rename*[*symmetric*, *OF bij-Not*]) (*simp add*: *comp-def*)

lemma *mono*:
  assumes *P* ≤ *P'*
  assumes *Q* ≤ *Q'*
  shows *spec.parallel P Q* ≤ *spec.parallel P' Q'*
by (*simp add*: *assms spec.parallel-def spec.Parallel.mono*)

lemma *strengthen*[*strg*]:
  assumes *st-ord F P P'*
  assumes *st-ord F Q Q'*
  shows *st-ord F* (*spec.parallel P Q*) (*spec.parallel P' Q'*)
using *assms* **by** (*cases F*; *simp add*: *spec.parallel.mono*)

lemma *mono2mono*[*cont-intro*, *partial-function-mono*]:
  assumes *monotone orda* (≤) *F*
  assumes *monotone orda* (≤) *G*
  shows *monotone orda* (≤) (λ*f*. *spec.parallel* (*F f*) (*G f*))
using *assms* **by** (*simp add*: *spec.parallel-def spec.Parallel.mono2mono*)

lemma *Sup*:
  fixes *Ps* :: (*sequential*, *'s*, *unit*) *spec set*
  shows *SupL*: ⨆ *Ps* ∥ *Q* = (⨆ *P*∈*Ps*. *P* ∥ *Q*)
    and *SupR*: *Q* ∥ ⨆ *Ps* = (⨆ *P*∈*Ps*. *Q* ∥ *P*)
by (*simp-all add*: *spec.parallel-alt-def spec.invmap.Sup spec.map.Sup heyting.inf-SUP-distrib image-image*)

lemma *sup*:
  fixes *P* :: (*sequential*, *'s*, *unit*) *spec*
  shows *supL*: (*P* ⊔ *Q*) ∥ *R* = (*P* ∥ *R*) ⊔ (*Q* ∥ *R*)
    and *supR*: *P* ∥ (*Q* ⊔ *R*) = (*P* ∥ *Q*) ⊔ (*P* ∥ *R*)
using *spec.parallel.Sup*[**where** *Ps*={*P*, *Q*} **for** *P Q*, *simplified*] **by** *fast*+

We can residuate (∥) but not *prog.parallel* (see §13.3) as the latter is not strict. Intuitively any realistic modelling of parallel composition will be non-strict as the divergence of one process should not block the progress of others, and incorporating such interference may preclude the implementation of a specification via this residuation.

References:

- Hayes (2016, Law 23): residuate parallel
- van Staden (2015, Lemma 6) who cites Armstrong, Gomes, and Struth (2014)

definition *res* :: (*sequential*, *'s*, *unit*) *spec* ⇒ (*sequential*, *'s*, *unit*) *spec* ⇒ (*sequential*, *'s*, *unit*) *spec* **where**
res S i = ⨆{*P*. *P* ∥ *i* ≤ *S*}

interpretation *res*: *galois.complete-lattice-class* λ*S*. *spec.parallel S i* λ*S*. *spec.parallel.res S i* **for** *i* — Hayes (2016, Law 23 (rely refinement))
proof
have ∗: *spec.parallel.res S i* ∥ *i* ≤ *S* **for** *S* — Hayes (2016, Law 22 (rely quotient))
by (*simp add*: *spec.parallel.res-def spec.parallel.SupL*)
show *x* ∥ *i* ≤ *S* ⟷ *x* ≤ *spec.parallel.res S i* **for** *x S*
by (*fastforce simp*: *spec.parallel.res-def Sup-upper spec.parallel.mono intro*: *order.trans*[*OF* - ∗])
qed

lemma *mcont2mcont*[*cont-intro*]:
assumes *mcont luba orda Sup* (≤) *P*
assumes *mcont luba orda Sup* (≤) *Q*
shows *mcont luba orda Sup* (≤) (λ*x*. *spec.parallel* (*P x*) (*Q x*))
proof(*rule ccpo.mcont2mcont'*[*OF complete-lattice-ccpo* - - *assms*(*1*)])
show *mcont Sup* (≤) *Sup* (≤) (λ*y*. *spec.parallel y* (*Q x*)) **for** *x*
by (*intro mcontI contI monotoneI*) (*simp-all add*: *spec.parallel.mono spec.parallel.SupL*)
show *mcont luba orda Sup* (≤) (λ*x*. *spec.parallel y* (*Q x*)) **for** *y*
by (*simp add*: *mcontI monotoneI contI mcont-monoD*[*OF assms*(*2*)] *spec.parallel.mono mcont-contD*[*OF assms*(*2*)] *spec.parallel.SupR image-image*)
qed

lemma *inf-rel*:
shows *spec.rel* ({*env*} × *UNIV* ∪ {*self*} × *r*) ⊓ (*P* ∥ *Q*)
= (*spec.rel* ({*env*} × *UNIV* ∪ {*self*} × *r*) ⊓ *P*) ∥ (*spec.rel* ({*env*} × *UNIV* ∪ {*self*} × *r*) ⊓ *Q*)
and (*P* ∥ *Q*) ⊓ *spec.rel* ({*env*} × *UNIV* ∪ {*self*} × *r*)
= (*spec.rel* ({*env*} × *UNIV* ∪ {*self*} × *r*) ⊓ *P*) ∥ (*spec.rel* ({*env*} × *UNIV* ∪ {*self*} × *r*) ⊓ *Q*)
by (*simp-all add*: *ac-simps spec.parallel-def spec.Parallel.inf-rel if-distrib*[**where** *f*=λ*x*. *x* ⊓ *y* **for** *y*])

lemma *assoc*:
shows *spec.parallel P* (*spec.parallel Q R*) = *spec.parallel* (*spec.parallel P Q*) *R* (**is** *?lhs* = *?rhs*)
by (*auto simp*: *spec.parallel-def bij-betw-def Plus-def UNIV-bool*
spec.Parallel.flatten[**where** *a*=*False*] *spec.Parallel.flatten*[**where** *a*=*True*]
intro!: *spec.Parallel.rename-cong*[**where** π=*case-sum* (λ*a*. *if a then Inr True else undefined*)
(λ*a*. *if a then Inr False else Inl False*)])

lemma *bind-botR*:
shows *spec.parallel* (*P* ⤜ ⊥) *Q* = *spec.parallel P Q* ⤜ ⊥
and *spec.parallel P* (*Q* ⤜ ⊥) = *spec.parallel P Q* ⤜ ⊥
by (*simp-all add*: *spec.bind.botR spec.term.none.parallelL spec.term.none.parallelR*)

lemma *interference*:
shows *interferenceL*: *spec.rel* ({*env*} × *UNIV*) ∥ *c* = *c*
and *interferenceR*: *c* ∥ *spec.rel* ({*env*} × *UNIV*) = *c*
by (*simp-all add*: *spec.parallel-def spec.Parallel.singleton-agents*
flip: *spec.Parallel.rename-UNIV*[**where** *as*={*False*} **and** *f*=*id* **and** *Ps*=⟨*c*⟩, *simplified*]
spec.Parallel.rename-UNIV[**where** *as*={*True*} **and** *f*=*id* **and** *Ps*=⟨*c*⟩, *simplified*])

lemma *unwindL*:
assumes *spec.rel* ({*env*} × *UNIV*) ⤜ (λ-::*unit*. *Q*) ≤ *Q* — All other processes begin with interference
assumes *f* ⤜ *g* ≤ *P* — The selected process starts with action *f*
shows *f* ⤜ (λ*v*. *g v* ∥ *Q*) ≤ *P* ∥ *Q*

unfolding *spec.parallel-def*
by (*strengthen ord-to-strengthen*[*OF spec.Parallel.unwind*[**where** *a*=*True*]])
(*auto simp*: *spec.Parallel.mono spec.bind.mono intro*: *assms*)

lemma *unwindR*:
assumes *spec.rel* ({*env*} × *UNIV*) ⤜ (λ-::*unit*. *P*) ≤ *P* — All other processes begin with interference
assumes *f* ⤜ *g* ≤ *Q* — The selected process starts with action *f*
shows *f* ⤜ (λ*v*. *P* ∥ *g v*) ≤ *P* ∥ *Q*
by (*subst* (*1 2*) *spec.parallel.commute*) (*rule spec.parallel.unwindL*[*OF assms*])

setup ‹*Sign.parent-path*›

setup ‹*Sign.mandatory-path interference.closed*›

lemma *toConcurrent-gen*:
fixes *P* :: (*sequential*, *'s*, *'v*) *spec*
fixes *a* :: *'a*
assumes *P*: *P* ∈ *spec.interference.closed* ({*env*} × *r*)
shows *spec.toConcurrent a P* ∈ *spec.interference.closed* ((−{*proc a*}) × *r*)
proof −
have ∗: *map-prod* (*toConcurrent-fn* (*proc a*)) *id* −‘ ({*env*} × *r*) = (−{*proc a*}) × *r*
by (*force simp*: *toConcurrent-fn-def*)
show *?thesis*
apply (*rule spec.interference.closed-clI*)
apply (*subst* (*2*) *spec.interference.closed-conv*[*OF P*])
apply (*force intro*: *spec.interference.cl.mono simp*: ∗ *spec.invmap.interference.cl*)
done
qed

lemma *toConcurrent*:
fixes *P* :: (*sequential*, *'s*, *'v*) *spec*
fixes *a* :: *'a*
assumes *P*: *P* ∈ *spec.interference.closed* ({*env*} × *r*)
shows *spec.toConcurrent a P* ∈ *spec.interference.closed* ({*env*} × *r*)
by (*blast intro*: *subsetD*[*OF spec.interference.closed.antimono*
spec.interference.closed.toConcurrent-gen[*OF assms*]])

lemma *toSequential*:
fixes *P* :: (*'a agent*, *'s*, *'v*) *spec*
assumes *P* ∈ *spec.interference.closed* ({*env*} × *r*)
shows *spec.toSequential P* ∈ *spec.interference.closed* ({*env*} × *r*)
proof −
have ∗: *map-prod toSequential-fn id* ‘ ({*env*} × *r*) = {*env*} × *r*
by (*force simp*: *toSequential-fn-def*)
show *?thesis*
apply (*rule spec.interference.closed-clI*)
apply (*subst* (*2*) *spec.interference.closed-conv*[*OF assms*])
apply (*simp add*: ∗ *spec.map.interference.cl-sf-id*)
done
qed

lemma *Parallel*:
assumes ⋀*a*. *Ps a* ∈ *spec.interference.closed* ({*env*} × *UNIV*)
shows *spec.Parallel as Ps* ∈ *spec.interference.closed* ({*env*} × *UNIV*)
unfolding *spec.Parallel-def*
by (*fastforce intro*: *spec.interference.closed.rel spec.interference.closed-Inf spec.interference.closed.toSequential*
simp: *assms image-subset-iff spec.interference.closed.toConcurrent*)

lemma *parallel*:
  assumes $P \in$ *spec.interference.closed* $(\{env\} \times UNIV)$
  assumes $Q \in$ *spec.interference.closed* $(\{env\} \times UNIV)$
  shows $P \parallel Q \in$ *spec.interference.closed* $(\{env\} \times UNIV)$
by (*simp add*: *assms spec.parallel-def spec.interference.closed.Parallel*)

setup ‹*Sign.parent-path*›

setup ‹*Sign.parent-path*›

9.6 Specification Inhabitation

Given that $\bot$ satisfies any specification S, we may wish to show that a specific trace σ is allowed by S.
The strategy is to compute the allowed transitions from a given initial state and possibly a return value. We almost always discard the closures we've added for various kinds of compositionality.
References:

- Similar to how van Staden (2014, §3.3) models a small-step operational semantics.
  - i.e., we can (semantically) define something like an LTS, which is compositional wrt parallel
  - a bit like a resumption or a residual
- Similar to Hoare, He, and Sampaio (2000)

TODO:

- often want transitive variants of these rules
- automate: only stop when there's a scheduling decision to be made

definition *inhabits* :: $('a, 's, 'w)$ *spec* $\Rightarrow$ $'s \Rightarrow ('a \times 's)$ *list* $\Rightarrow ('a, 's, 'w)$ *spec* $\Rightarrow$ *bool* $(\langle\text{-}/ -\text{-}, \text{-}\!\!\rightarrow/ \text{-}\rangle\ [50, 0, 0, 50]\ 50)$ **where**
  $S\ -s, xs\!\!\rightarrow\ T \longleftrightarrow \langle s, xs, Some\ ()\rangle \gg T \leq S$

setup ‹*Sign.mandatory-path inhabits*›

lemma *incomplete*:
  assumes $S\ -s, xs\!\!\rightarrow\ S'$
  shows $\langle s, xs, None\rangle \leq S$
by (*strengthen ord-to-strengthen*[*OF assms*[*unfolded inhabits-def*]])
  (*simp add*: *spec.bind.incompleteI*)

lemma *complete*:
  assumes $S\ -s, xs\!\!\rightarrow$ *spec.return v*
  shows $\langle s, xs, Some\ v\rangle \leq S$
by (*strengthen ord-to-strengthen*[*OF assms*[*unfolded inhabits-def*]])
  (*simp add*: *spec.bind.continueI*[**where** *ys*=[], *simplified*] *spec.singleton.le-conv*)

lemmas $I =$ *inhabits.complete inhabits.incomplete*

lemma *mono*:
  assumes $S \leq S'$
  assumes $T' \leq T$
  assumes $S\ -s, xs\!\!\rightarrow\ T$
  shows $S'\ -s, xs\!\!\rightarrow\ T'$
unfolding *inhabits-def*
apply (*strengthen ord-to-strengthen*[*OF assms*(*1*)])
apply (*strengthen ord-to-strengthen*[*OF assms*(*2*)])

apply (*rule assms*(*3*)[*unfolded inhabits-def*])
done

lemma *strengthen*[*strg*]:
  assumes *st-ord F S S′*
  assumes *st-ord* (¬*F*) *T T′*
  shows *st F* (⟶) (*S* −*s*, *xs*→ *T*) (*S′* −*s*, *xs*→ *T′*)
using *assms* **by** (*cases F*; *simp add*: *inhabits.mono*)

lemma *pre*:
  assumes *S* −*s*, *xs′*→ *T*
  assumes *T′* ≤ *T*
  assumes *xs* = *xs′*
  shows *S* −*s*, *xs*→ *T′*
using *assms* **by** (*blast intro*: *inhabits.mono*[*OF order.refl assms*(*2*)])

lemma *tau*:
  assumes *spec.idle* ≤ *S*
  shows *S* −*s*, []→ *S*
unfolding *inhabits-def*
by (*strengthen ord-to-strengthen*[*OF spec.action.stutterI*[**where** *F*={()} × *UNIV* × *Id*]])
  (*simp-all add*: *assms spec.bind.returnL flip*: *spec.return-def*)

lemma *trans*:
  assumes *R* −*s*, *xs*→ *S*
  assumes *S* −*trace.final′ s xs*, *ys*→ *T*
  shows *R* −*s*, *xs* @ *ys*→ *T*
unfolding *inhabits-def*
apply (*strengthen ord-to-strengthen*(*2*)[*OF assms*(*1*)[*unfolded inhabits-def*]])
apply (*strengthen ord-to-strengthen*(*2*)[*OF assms*(*2*)[*unfolded inhabits-def*]])
apply (*simp add*: *spec.bind.continueI spec.bind.mono flip*: *spec.bind.bind*)
done

lemma *Sup*:
  assumes *P* −*s*, *xs*→ *P′*
  assumes *P* ∈ *X*
  shows ⨆ *X* −*s*, *xs*→ *P′*
using *assms* **by** (*simp add*: *inhabits-def Sup-upper2*)

lemma *supL*:
  assumes *P* −*s*, *xs*→ *P′*
  shows *P* ⊔ *Q* −*s*, *xs*→ *P′*
using *assms* **by** (*simp add*: *inhabits-def le-supI1*)

lemma *supR*:
  assumes *Q* −*s*, *xs*→ *Q′*
  shows *P* ⊔ *Q* −*s*, *xs*→ *Q′*
using *assms* **by** (*simp add*: *inhabits-def le-supI2*)

lemma *inf*:
  assumes *P* −*s*, *xs*→ *P′*
  assumes *Q* −*s*, *xs*→ *Q′*
  shows *P* ⊓ *Q* −*s*, *xs*→ *P′* ⊓ *Q′*
using *assms* **by** (*meson inf.cobounded1 inf.cobounded2 le-inf-iff inhabits.pre inhabits-def*)

lemma *infL*:
  assumes *P* −*s*, *xs*→ *R*
  assumes *Q* −*s*, *xs*→ *R*

shows $P \sqcap Q \; -s,\, xs\rightarrow \; R$
using *assms* **by** (*meson le-inf-iff inhabits-def*)

setup ‹*Sign.mandatory-path spec*›

lemma *bind*:
assumes $f \; -s,\, xs\rightarrow \; f'$
shows $f \gg\!= g \; -s,\, xs\rightarrow \; f' \gg\!= g$
using *assms* **by** (*simp add*: *inhabits-def spec.bind.mono flip*: *spec.bind.bind*)

lemmas *bind'* = *inhabits.trans*[*OF inhabits.spec.bind*]

lemma *parallelL*:
assumes $P \; -s,\, xs\rightarrow \; P'$
assumes *spec.rel* $(\{env\} \times UNIV) \gg\!= (\lambda\text{-}::unit.\; Q) \leq Q$
shows $P \parallel Q \; -s,\, xs\rightarrow \; P' \parallel Q$
by (*rule inhabits.mono*[*OF spec.parallel.unwindL*[*OF assms*(*2*) *assms*(*1*)[*unfolded inhabits-def*]]
order.refl])
(*simp add*: *inhabits-def*)

lemma *parallelR*:
assumes $Q \; -s,\, xs\rightarrow \; Q'$
assumes *spec.rel* $(\{env\} \times UNIV) \gg\!= (\lambda\text{-}::unit.\; P) \leq P$
shows $P \parallel Q \; -s,\, xs\rightarrow \; P \parallel Q'$
by (*subst* (*1 2*) *spec.parallel.commute*) (*rule inhabits.spec.parallelL assms*)+

lemmas *parallelL'* = *inhabits.trans*[*OF inhabits.spec.parallelL*]
lemmas *parallelR'* = *inhabits.trans*[*OF inhabits.spec.parallelR*]

setup ‹*Sign.mandatory-path action*›

lemma *step*:
assumes $(v,\, a,\, s,\, s') \in F$
shows *spec.action* $F \; -s,\, [(a,\, s')]\rightarrow$ *spec.return v*
by (*clarsimp simp*: *inhabits-def trace.split-all spec.bind.singletonL spec.term.none.singleton*
spec.singleton.le-conv spec.action.stepI[*OF assms*]
intro!: *ord-eq-le-trans*[*OF spec.singleton.Cons spec.action.stepI*[*OF assms*]])

lemma *stutter*:
assumes $(v,\, a,\, s,\, s) \in F$
shows *spec.action* $F \; -s,\, []\rightarrow$ *spec.return v*
using *inhabits.spec.action.step*[*OF assms*] **by** (*simp add*: *inhabits-def*)

setup ‹*Sign.parent-path*›

lemma *map*:
fixes *af* :: $'a \Rightarrow 'b$
fixes *sf* :: $'s \Rightarrow 't$
fixes *vf* :: $'v \Rightarrow 'w$
assumes $P \; -s,\, xs \rightarrow$ *spec.return v*
shows *spec.map af sf vf P* $-sf\; s,$ *map* (*map-prod af sf*) $xs\rightarrow$ *spec.return* (*vf v*)
proof −
have $\langle sf\; s,$ *map* (*map-prod af sf*) *xs*, *Some* $()\rangle \gg$ *spec.return* (*vf v*)
$\leq$ *spec.map af sf vf* $(\langle s,$ *xs*, *Some* $()\rangle \gg$ *spec.return v*)
by (*subst* (*1*) *spec.bind.singletonL*)
(*fastforce intro*: *spec.bind.incompleteI*
spec.bind.continueI[**where** *ys*=[] **and** *w*=*Some v*, *simplified*]
simp: *spec.singleton.le-conv spec.term.none.singleton*

split: *option.split-asm*)
then show *?thesis*
using *assms* **by** (*auto simp*: *inhabits-def dest*: *spec.map.mono*[**where** *af*=*af* **and** *sf*=*sf* **and** *vf*=*vf*])
qed

lemma *invmap*:
fixes *af* :: $'a \Rightarrow 'b$
fixes *sf* :: $'s \Rightarrow 't$
fixes *vf* :: $'v \Rightarrow 'w$
assumes P $-sf\ s,$ *map* (*map-prod af sf*) *xs* $\rightarrow$ P'
shows *spec.invmap af sf vf P* $-s,$ *xs*$\rightarrow$ *spec.invmap af sf vf* P'
by (*strengthen ord-to-strengthen*(*2*)[*OF assms*(*1*)[*unfolded inhabits-def*]])
(*simp add*: *inhabits-def spec.invmap.bind spec.map.singleton spec.bind.mono*
flip: *spec.map-invmap.galois*)

setup ‹*Sign.mandatory-path term.none*›

lemma *step*:
assumes P $-s,$ *xs*$\rightarrow$ P'
shows *spec.term.none P* $-s,$ *xs*$\rightarrow$ *spec.term.none* P'
by (*simp add*: *inhabits.spec.bind*[*OF assms*] *flip*: *spec.bind.botR*)

setup ‹*Sign.parent-path*›

setup ‹*Sign.mandatory-path term.all*›

lemma *step*:
assumes P $-s,$ *xs*$\rightarrow$ P'
shows *spec.term.all P* $-s,$ *xs*$\rightarrow$ *spec.term.all* P'
by (*strengthen ord-to-strengthen*(*2*)[*OF assms*(*1*)[*unfolded inhabits-def*]])
(*simp add*: *inhabits-def spec.term.all.bind*)

lemma *term*:
assumes *spec.idle* $\leq$ P
shows *spec.term.all P* $-s,$ []$\rightarrow$ *spec.return v*
by (*strengthen ord-to-strengthen*(*2*)[*OF assms*(*1*)])
(*auto simp*: *spec.term.all.idle intro*: *spec.idle-le inhabits.tau inhabits.Sup*)

setup ‹*Sign.parent-path*›

setup ‹*Sign.mandatory-path kleene.star*›

lemma *step*:
assumes P $-s,$ *xs*$\rightarrow$ P'
shows *spec.kleene.star P* $-s,$ *xs*$\rightarrow$ $P' \gg$ *spec.kleene.star P*
by (*subst spec.kleene.star.simps*) (*simp add*: *assms inhabits.supL inhabits.spec.bind*)

lemma *term*:
shows *spec.kleene.star P* $-s,$ []$\rightarrow$ *spec.return* ()
by (*metis inhabits.tau inhabits.supR spec.kleene.star.simps spec.idle.return-le*)

setup ‹*Sign.parent-path*›

setup ‹*Sign.mandatory-path rel*›

lemma *rel*:
assumes *trace.steps'* s *xs* $\subseteq$ r
shows *spec.rel r* $-s,$ *xs*$\rightarrow$ *spec.rel r*

proof −
from *assms*
have ⟪*s*, *xs*, *Some* ()⟫ ≫ *spec.rel r* ≤ *spec.rel r* ⤜ (λ-::*unit*. *spec.rel r*)
by (*simp add*: *spec.bind.mono spec.singleton.le-conv*)
then show *?thesis*
by (*simp add*: *inhabits-def spec.rel.wind-bind*)
qed

lemma *rel-term*:
assumes *trace.steps′ s xs* ⊆ *r*
shows *spec.rel r* −*s*, *xs*→ *spec.return v*
by (*rule inhabits.pre*[*OF inhabits.spec.rel.rel*[*OF assms*] *spec.return.rel-le refl*])

lemma *step*:
assumes (*a*, *s*, *s′*) ∈ *r*
shows *spec.rel r* −*s*, [(*a*, *s′*)]→ *spec.rel r*
by (*rule inhabits.pre*)
(*auto intro*: *assms inhabits.spec.action.step*[**where** *s′*=*s′*] *inhabits.spec.kleene.star.step*
inhabits.spec.term.all.step
simp: *spec.rel-def spec.rel.act-def spec.bind.returnL spec.idle-le*)

lemma *term*:
shows *spec.rel r* −*s*, []→ *spec.return v*
by (*simp add*: *inhabits.pre*[*OF inhabits.tau*[*OF spec.idle.rel-le*] *spec.return.rel-le*])

setup ‹*Sign.parent-path*›

setup ‹*Sign.parent-path*›

setup ‹*Sign.parent-path*›

10 "Next step" implication ala Abadi and Merz (and Lamport)

As was apparently well-known in the mid-1990s (see, e.g., Xu, Cau, and Collette (1994, §4) and the references therein), Heyting implication is inadequate for a general refinement story. (We show it is strong enough for a relational assume/guarantee program logic; see §9.2, §12.2 and §13.5.2. In our setting it fails to generalize (at least) because the composition theorem for Heyting implication (§9.2) is not termination sensitive.)

We therefore follow Abadi and Lamport (1995) by developing a stronger implication $P \longrightarrow_+ Q$ with the intuitive semantics that the consequent Q holds for at least one step beyond the antecedent P. This is some kind of step indexing.

Here we sketch the relevant parts of Abadi and Merz (1995, 1996), the latter of which has a fuller story, including a formal account of the logical core of TLA and the (implicit) observation that infinitary parallel composition poses no problem for safety properties (see the comments under Theorem 5.2 and §5.5). Abadi and Lamport (1995); Cau and Collette (1996); Xu et al. (1994) provide further background; Jonsson and Tsay (1996, Appendix B) provide a more syntactic account.

Observations:

- The hypothesis P is always a safety property here
- TLA does not label transitions or have termination markers
- Abadi/Cau/Collette/Lamport/Merz/Xu/... avoid naming this operator

Further references:

- Maier (2001)

definition *next-imp* :: *'a*::*preorder set* ⇒ *'a set* ⇒ *'a set* **where** — Abadi and Merz (1995, §2)

next-imp P Q = $\{\sigma.\ \forall \sigma' \leq \sigma.\ (\forall \sigma'' < \sigma'.\ \sigma'' \in P) \longrightarrow \sigma' \in Q\}$

setup ‹*Sign.mandatory-path next-imp*›

lemma *downwards-closed*:
  assumes $P \in$ *downwards.closed*
  shows *next-imp P Q* $\in$ *downwards.closed*
unfolding *next-imp-def* **by** (*blast elim*: *downwards.clE intro*: *order-trans*)

lemma *mono*:
  assumes $x' \leq x$
  assumes $y \leq y'$
  shows *next-imp x y* $\leq$ *next-imp x' y'*
unfolding *next-imp-def* **using** *assms* **by** *fast*

lemma *strengthen*[*strg*]:
  assumes *st-ord* (¬ *F*) *X X'*
  assumes *st-ord F Y Y'*
  shows *st-ord F* (*next-imp X Y*) (*next-imp X' Y'*)
using *assms* **by** (*cases F*) (*auto simp*: *next-imp.mono*)

lemma *minimal*:
  assumes *trace.T s xs v* $\in$ *next-imp P Q*
  shows *trace.T s* [] *None* $\in$ *Q*
using *assms* **by** (*simp add*: *next-imp-def trace.less trace.less-eq-None*)

lemma *alt-def*: — This definition coincides with Cau and Collette (1996), Abadi and Lamport (1995, §3.5.3)
  assumes $P \in$ *downwards.closed*
  shows *next-imp P Q*
    = {σ. *trace.T* (*trace.init* σ) [] *None* $\in$ *Q*
      ∧ ($\forall$ *i*. *trace.take i* σ $\in$ *P* $\longrightarrow$ *trace.take* (*Suc i*) σ $\in$ *Q*)} (**is** *?lhs* = *?rhs*)
proof(*rule antisym*)
  have *trace.take* (*Suc i*) (*trace.T s xs v*) $\in$ *Q*
    if *trace.T s xs v* $\in$ *?lhs* **and** *trace.take i* (*trace.T s xs v*) $\in$ *P*
   for *s xs v i*
    using *that* ‹*P* $\in$ *downwards.closed*›
    by (*force simp*: *next-imp-def trace.less-take-less-eq*
          dest: *spec*[**where** *x*=*trace.take* (*Suc i*) (*trace.T s xs v*)]
          elim: *downwards.closed-in*)
  then show *?lhs* $\subseteq$ *?rhs*
    by (*clarsimp simp*: *trace.split-all next-imp.minimal*)
next
  have *trace.T s xs v* $\in$ *?lhs*
    if *minimal*: *trace.T s* [] *None* $\in$ *Q*
   and *imp*: $\forall$ *i*. *trace.take i* (*trace.T s xs v*) $\in$ *P* $\longrightarrow$ *trace.take* (*Suc i*) (*trace.T s xs v*) $\in$ *Q*
   for *s xs v*
  proof −
    have *trace.take i* (*trace.T s xs v*) $\in$ *Q*
      if $\forall \sigma'' <$ *trace.take i* (*trace.T s xs v*). $\sigma'' \in P$
     for *i*
      using *that*
    proof(*induct i*)
      case (*Suc i*) **with** *imp* **show** *?case*
        by (*metis le-add2 order-le-less plus-1-eq-Suc trace.take.mono*)
    qed (*simp add*: *minimal*)
    then show *trace.T s xs v* $\in$ *?lhs*
      by (*clarsimp simp*: *next-imp-def trace.less-eq-take-def*)
  qed

then show *?rhs* ⊆ *?lhs*
by (*clarsimp simp*: *trace.split-all next-imp.minimal*)
qed

Abadi and Lamport (1995, §3.5.3) assert but do not prove the following connection with Heyting implication. Abadi and Merz (1995) do. See also Abadi and Merz (1996, §5.3 and §5.5).

lemma *Abadi-Merz-Prop-1-subseteq*: — First half of Abadi and Merz (1995, Proposition 1)
fixes *P* :: *'a*::*preorder set*
assumes *P* ∈ *downwards.closed*
assumes *wf*: *wfP* ((<) :: *'a relp*)
shows *next-imp P Q* ⊆ *downwards.imp* (*downwards.imp Q P*) *Q* (**is** *?lhs* ⊆ *?rhs*)
proof(*rule subsetI*)
fix σ **assume** σ ∈ *?lhs* **with** *wf* **show** σ ∈ *?rhs*
proof(*induct rule*: *wfp-induct-rule*)
case (*less* σ)
have τ ∈ *Q* **if** τ ≤ σ **and** *YYY*: ∀σ'≤τ. σ' ∈ *Q* ⟶ σ' ∈ *P* **for** τ
proof −
have ϱ ∈ *P* **if** ϱ < τ **for** ϱ
proof −
from ‹ϱ < τ› ‹τ ≤ σ› ‹*P* ∈ *downwards.closed*› **have** ϱ ∈ *next-imp P Q*
by (*meson downwards.closed-in next-imp.downwards-closed less.prems less-imp-le*)
with ‹ϱ < τ› ‹τ ≤ σ› **have** ϱ ∈ *downwards.imp* (*downwards.imp Q P*) *Q*
using *less.hyps less-le-trans* **by** *blast*
moreover from ‹ϱ < τ› *YYY* **have** ϱ ∈ *downwards.imp Q P*
by (*simp add*: *downwards.imp-def*) (*meson order.trans order-less-imp-le*)
ultimately show *?thesis* **by** (*meson downwards.imp-mp'*)
qed
with *that less.prems* **show** *?thesis*
unfolding *next-imp-def* **by** *blast*
qed
then show *?case*
unfolding *downwards.imp-def* **by** *blast*
qed
qed

The converse holds if either *Q* is a safety property or the order is partial.

lemma *Abadi-Merz-Prop1*: — Abadi and Merz (1995, Proposition 1) and Abadi and Merz (1996, Proposition 5.2)
fixes *P* :: *'a*::*preorder set*
assumes *P* ∈ *downwards.closed*
assumes *Q* ∈ *downwards.closed*
assumes *wf*: *wfP* ((<) :: *'a relp*)
shows *next-imp P Q* = *downwards.imp* (*downwards.imp Q P*) *Q* (**is** *?lhs* = *?rhs*)
proof(*rule antisym*[*OF next-imp.Abadi-Merz-Prop-1-subseteq*[*OF assms*(*1*,*3*)]])
from ‹*Q* ∈ *downwards.closed*› **show** *?rhs* ⊆ *?lhs*
by (*auto simp*: *next-imp-def downwards.imp-def order.strict-iff-not dest*: *downwards.closed-in*)
qed

lemma *Abadi-Lamport-Lemma6*: — Abadi and Lamport (1995, Lemma 6) (no proof given there)
fixes *P* :: *'a*::*order set*
assumes *P* ∈ *downwards.closed*
assumes *wf*: *wfP* ((<) :: *'a relp*)
shows *next-imp P Q* = *downwards.imp* (*downwards.imp Q P*) *Q* (**is** *?lhs* = *?rhs*)
proof(*rule Set.equalityI*[*OF next-imp.Abadi-Merz-Prop-1-subseteq*[*OF assms*]])
show *?rhs* ⊆ *?lhs*
unfolding *next-imp-def downwards.imp-def* **by** (*fastforce simp*: *le-less elim*: *downwards.closed-in*)
qed

lemmas *downwards-imp* = *next-imp.Abadi-Lamport-Lemma6*[*OF - trace.wfP-less*]

lemma *boolean-implication-le*:
  assumes $P \in$ *downwards.closed*
  shows *next-imp* $P\ Q \subseteq P \longrightarrow_B Q$
using *downwards.closed-conv*[*OF assms*]
by (*fastforce simp*: *next-imp-def boolean-implication.member*
        intro: *order-less-imp-le*)

setup ‹*Sign.parent-path*›

setup ‹*Sign.mandatory-path spec*›

lift-definition *next-imp* :: ('*a*, '*s*, '*v*) *spec* ⇒ ('*a*, '*s*, '*v*) *spec* ⇒ ('*a*, '*s*, '*v*) *spec* (**infixr** ‹$\longrightarrow_+$› *61*) **is**
  Next-Imp.next-imp
by (*simp add*: *next-imp.downwards-imp raw.spec.closed.downwards-closed raw.spec.closed.downwards-imp*)

setup ‹*Sign.mandatory-path next-imp*›

lemma *heyting*: — fundamental
  shows $P \longrightarrow_+ Q = (Q \longrightarrow_H P) \longrightarrow_H Q$
by *transfer* (*simp add*: *next-imp.downwards-imp raw.spec.closed.downwards-closed*)

setup ‹*Sign.parent-path*›

setup ‹*Sign.mandatory-path singleton*›

lemma *next-imp-le-conv*:
  fixes P :: ('*a*, '*s*, '*v*) *spec*
  shows $\langle\!|\sigma|\!\rangle \leq P \longrightarrow_+ Q \longleftrightarrow (\forall \sigma' \leq \sigma.\ (\forall \sigma'' < \sigma'.\ \langle\!|\sigma''|\!\rangle \leq P) \longrightarrow \langle\!|\sigma'|\!\rangle \leq Q)$ (**is** *?lhs* ⟷ *?rhs*)
proof(*rule iffI*)
  show *?lhs* ⟹ *?rhs*
    by (*force simp*: *spec.next-imp.heyting spec.singleton.heyting-le-conv*)
  note *spec.singleton.transfer*[*transfer-rule*]
  show *?rhs* ⟹ *?lhs*
  proof(*transfer*, *unfold raw.singleton-def*, *rule raw.spec.least*)
    show $\{\sigma\} \subseteq$ *next-imp* $P\ Q$
      if $P \in$ *raw.spec.closed*
     and $\forall \sigma' \leq \sigma.\ (\forall \sigma'' < \sigma'.$ *raw.spec.cl* $\{\sigma''\} \subseteq P) \longrightarrow$ *raw.spec.cl* $\{\sigma'\} \subseteq Q$
     for $P\ Q$ :: ('*a*, '*s*, '*v*) *trace.t set* **and** σ
      using *that* **by** (*auto simp*: *next-imp-def raw.spec.least-conv*
                    dest: *order.trans*[*OF raw.spec.expansive*])
    show *next-imp* $P\ Q \in$ *raw.spec.closed*
      if $P \in$ *raw.spec.closed*
     and $Q \in$ *raw.spec.closed*
     for $P\ Q$ :: ('*a*, '*s*, '*v*) *trace.t set*
      using *that*
      by (*simp add*: *next-imp.downwards-imp raw.spec.closed.downwards-closed raw.spec.closed.downwards-imp*)
  qed
qed

setup ‹*Sign.parent-path*›

setup ‹*Sign.mandatory-path next-imp*›

lemma *mono*:
  assumes $x' \leq x$
  assumes $y \leq y'$
  shows $x \longrightarrow_+ y \leq x' \longrightarrow_+ y'$

by (*simp add*: *assms heyting.mono spec.next-imp.heyting*)

lemma *strengthen*[*strg*]:
  assumes *st-ord* (¬ *F*) *X X′*
  assumes *st-ord F Y Y′*
  shows *st-ord F* (*X* ⟶$_+$ *Y*) (*X′* ⟶$_+$ *Y′*)
using *assms* **by** (*cases F*) (*auto simp*: *spec.next-imp.mono*)

lemma *idempotentR*:
  shows *P* ⟶$_+$ (*P* ⟶$_+$ *Q*) = *P* ⟶$_+$ *Q*
by (*simp add*: *spec.next-imp.heyting heyting heyting.detachment*(*1*) *heyting.discharge*(*2*) *inf.absorb2*
      flip: *heyting.curry-conv*)

lemma *contains*:
  assumes *X* ≤ *Q*
  shows *X* ≤ *P* ⟶$_+$ *Q*
by (*simp add*: *assms spec.next-imp.heyting heyting.curry le-infI1*)

interpretation *closure*: *closure-complete-lattice-class* (⟶$_+$) *P* **for** *P*
by *standard*
  (*metis* (*no-types*, *lifting*) *order.refl order.trans*
                spec.next-imp.idempotentR spec.next-imp.contains spec.next-imp.mono)

lemma *InfR*:
  shows *x* ⟶$_+$ ⋂ *X* = ⋂ ((⟶$_+$) *x* ‘ *X*)
by *transfer* (*auto simp*: *next-imp-def*)

lemma *SupR-not-empty*:
  fixes *P* :: (-, -, -) *spec*
  assumes *X* ≠ {}
  shows *P* ⟶$_+$ (⨆ *x*∈*X*. *Q x*) = (⨆ *x*∈*X*. *P* ⟶$_+$ *Q x*) (**is** *?lhs* = *?rhs*)
proof(*rule antisym*[*OF spec.singleton-le-extI*
                spec.next-imp.closure.Sup-cl-le[**where** *X*=*Q* ‘ *X*, *simplified image-image*]])
  show ⟪*σ*⟫ ≤ *?rhs* **if** ⟪*σ*⟫ ≤ *?lhs* **for** *σ*
  proof(*cases* ⟪*σ*⟫ ≤ *P*)
    case *True* **with** ‹⟪*σ*⟫ ≤ *?lhs*› **show** *?thesis*
      by (*fastforce simp*: *spec.singleton.next-imp-le-conv*
                intro: *order.trans*[*OF spec.singleton.mono*]
                elim!: *order-less-imp-le*
                dest: *spec*[**where** *x*=*σ*])
  next
    case *False* **show** *?thesis*
    proof(*cases* ⟪*trace.init σ*, [], *None*⟫ ≤ *P*)
      case *True* **with** ‹¬ ⟪*σ*⟫ ≤ *P*›
      obtain *j* **where** ∗: ∀ *σ″*<*trace.take* (*Suc j*) *σ*. ⟪*σ″*⟫ ≤ *P*
              and ∗∗: ¬ ⟪*trace.take* (*Suc j*) *σ*⟫ ≤ *P*
        using *ex-least-nat-less*[**where** *P*=λ*i*. ¬⟪*trace.take i σ*⟫ ≤ *P* **and** *n*=*Suc* (*length* (*trace.rest σ*))]
        by (*force dest*: *trace.less-take-less-eq*
                simp: *less-Suc-eq-le order.trans*[*OF spec.singleton.mono*]
            simp flip: *trace.take.Ex-all*)
      from ‹⟪*σ*⟫ ≤ *?lhs*› ∗∗ **show** *?thesis*
      by (*clarsimp simp*: *spec.singleton.next-imp-le-conv*
                dest!: *spec*[**where** *x*=*trace.take* (*Suc j*) *σ*] *rev-mp*[*OF* ∗]
                elim!: *rev-bexI*)
          (*meson order.trans less-le-not-le spec.singleton.mono trace.less-eq-same-cases trace.less-eq-take*)
    next
      case *False* **with** ‹*X* ≠ {}› ‹⟪*σ*⟫ ≤ *?lhs*› **show** *?thesis*
        by (*clarsimp simp*: *spec.singleton.next-imp-le-conv simp flip*: *ex-in-conv*)

(*metis trace.take.0 trace.less-eq-take-def trace.less-t-def trace.take.sel*(*1*))
qed
qed
qed

lemma *cont*:
shows *cont Sup* (≤) *Sup* (≤) ((⟶$_+$) *P*)
by (*rule contI*) (*simp add*: *spec.next-imp.SupR-not-empty*[**where** *Q*=*id*, *simplified*])

lemma *mcont*:
shows *mcont Sup* (≤) *Sup* (≤) ((⟶$_+$) *P*)
by (*simp add*: *monotoneI mcontI*[*OF - spec.next-imp.cont*])

lemmas *mcont2mcont*[*cont-intro*] = *mcont2mcont*[*OF spec.next-imp.mcont*, *of luba orda Q P*] **for** *luba orda Q P*

lemma *botL*:
assumes *spec.idle* ≤ *P*
shows ⊥ ⟶$_+$ *P* = ⊤
by (*simp add*: *assms spec.next-imp.heyting spec.eq-iff Heyting.heyting spec.heyting.non-triv*)

lemma *topL*[*simp*]:
shows ⊤ ⟶$_+$ *P* = *P*
by (*simp add*: *spec.next-imp.heyting*)

lemmas *topR*[*simp*] = *spec.next-imp.closure.cl-top*

lemma *refl*:
shows *P* ⟶$_+$ *P* ≤ *P*
by (*simp add*: *spec.next-imp.heyting*)

lemma *heyting-le*:
shows *P* ⟶$_+$ *Q* ≤ *P* ⟶$_H$ *Q*
by (*simp add*: *spec.next-imp.heyting heyting.discard heyting.mono*)

lemma *discharge*:
shows *P* ⊓ (*P* ⊓ *Q* ⟶$_+$ *R*) = *P* ⊓ (*Q* ⟶$_+$ *R*) (**is** *?thesis1 P Q*)
and (*P* ⊓ *Q* ⟶$_+$ *R*) ⊓ *P* = *P* ⊓ (*Q* ⟶$_+$ *R*) (**is** *?thesis2*)
and *Q* ⊓ (*P* ⊓ *Q* ⟶$_+$ *R*) = *Q* ⊓ (*P* ⟶$_+$ *R*) (**is** *?thesis3*)
and (*P* ⊓ *Q* ⟶$_+$ *R*) ⊓ *Q* = *Q* ⊓ (*P* ⟶$_+$ *R*) (**is** *?thesis4*)
proof −
show *?thesis1 P Q* **for** *P Q*
by (*simp add*: *spec.next-imp.heyting heyting.infR heyting.curry-conv heyting.discard heyting.discharge*)
then show *?thesis2* **by** (*rule inf-commute-conv*)
from ‹*?thesis1 Q P*› **show** *?thesis3* **by** (*simp add*: *ac-simps*)
then show *?thesis4* **by** (*rule inf-commute-conv*)
qed

lemma *detachment*:
shows *x* ⊓ (*x* ⟶$_+$ *y*) ≤ *y*
and (*x* ⟶$_+$ *y*) ⊓ *x* ≤ *y*
by (*simp-all add*: *spec.next-imp.heyting heyting.discard heyting.discharge*)

lemma *infR*:
shows *P* ⟶$_+$ *Q* ⊓ *R* = (*P* ⟶$_+$ *Q*) ⊓ (*P* ⟶$_+$ *R*)
by (*rule antisym*[*OF spec.next-imp.closure.cl-inf-le*])
(*rule spec.singleton-le-extI*; *clarsimp simp*: *spec.singleton.next-imp-le-conv*)

lemma *supL-le*:

shows $x \sqcup y \longrightarrow_+ z \leq (x \longrightarrow_+ z) \sqcup (y \longrightarrow_+ z)$
by (*simp add*: *le-supI1 spec.next-imp.mono*)

lemma *heytingL*:
shows $(P \longrightarrow_H Q) \sqcap (Q \longrightarrow_+ R) \leq P \longrightarrow_+ R$
by (*simp add*: *spec.next-imp.heyting heyting ac-simps*)
(*simp add*: *heyting.rev-trans heyting.discharge flip*: *inf.assoc*)

lemma *heytingR*:
shows $(P \longrightarrow_+ Q) \sqcap (Q \longrightarrow_H R) \leq P \longrightarrow_+ R$
by (*simp add*: *spec.next-imp.heyting heyting ac-simps*)
(*simp add*: *heyting.discharge heyting.trans heyting.uncurry flip*: *inf.assoc*)

lemma *heytingL-distrib*:
shows $P \longrightarrow_H (Q \longrightarrow_+ R) = (P \sqcap Q) \longrightarrow_+ (P \longrightarrow_H R)$
by (*metis* (*no-types*, *opaque-lifting*) *heyting.curry-conv heyting.detachment*(*2*) *heyting.infR*
heyting.refl heyting.swap inf-top-left spec.next-imp.heyting)

lemma *trans*:
shows $(P \longrightarrow_+ Q) \sqcap (Q \longrightarrow_+ R) \leq P \longrightarrow_+ R$
by (*meson order.trans Heyting.heyting spec.next-imp.heytingL spec.next-imp.heyting-le*)

lemma *rev-trans*:
shows $(Q \longrightarrow_+ R) \sqcap (P \longrightarrow_+ Q) \leq P \longrightarrow_+ R$
by (*simp add*: *inf.commute spec.next-imp.trans*)

lemma
assumes $x' \leq x$
shows *discharge-leL*: $x' \sqcap (x \longrightarrow_+ y) = x' \sqcap y$ (**is** *?thesis1*)
and *discharge-leR*: $(x \longrightarrow_+ y) \sqcap x' = y \sqcap x'$ (**is** *?thesis2*)
proof −
from *assms* **show** *?thesis1*
by (*metis inf.absorb-iff2 inf-top.right-neutral spec.next-imp.discharge*(*4*) *spec.next-imp.topL*)
then show *?thesis2* **by** (*simp add*: *ac-simps*)
qed

lemma *invmap*:
shows *spec.invmap af sf vf* $(P \longrightarrow_+ Q) =$ *spec.invmap af sf vf* $P \longrightarrow_+$ *spec.invmap af sf vf* Q
by (*simp add*: *spec.next-imp.heyting spec.invmap.heyting*)

lemma *Abadi-Lamport-Lemma7*:
assumes $Q \sqcap R \leq P$
shows $P \longrightarrow_+ Q \leq R \longrightarrow_+ Q$
by (*simp add*: *assms spec.next-imp.heyting Heyting.heyting heyting.detachment*(*2*) *heyting.discharge*(*2*))

setup ‹*Sign.parent-path*›

setup ‹*Sign.mandatory-path term*›

setup ‹*Sign.mandatory-path none*›

lemma *next-imp*:
shows *spec.term.none* $(P \longrightarrow_+ Q) \leq$ *spec.term.all* $P \longrightarrow_+$ *spec.term.none* Q
by (*simp add*: *spec.next-imp.heyting order.trans*[*OF spec.term.none.heyting-le*] *spec.term.all.heyting*)

setup ‹*Sign.parent-path*›

setup ‹*Sign.mandatory-path all*›

lemma *next-imp*:
shows *spec.term.all* $(P \longrightarrow_+ Q) = spec.term.all\ P \longrightarrow_+ spec.term.all\ Q$
by (*simp add*: *spec.next-imp.heyting*)
(*metis spec.term.all.heyting spec.term.all-none spec.term.heyting-noneL-allR-mono spec.term.none-all*)

setup ‹*Sign.parent-path*›

setup ‹*Sign.mandatory-path closed*›

lemma *next-imp*:
assumes $Q \in$ *spec.term.closed* -
shows $P \longrightarrow_+ Q \in$ *spec.term.closed* -
using *assms*
by (*simp add*: *spec.next-imp.heyting spec.term.closed.heyting*)

setup ‹*Sign.parent-path*›

setup ‹*Sign.parent-path*›

setup ‹*Sign.mandatory-path pre*›

lemma *next-imp-eq-heyting*:
assumes $spec.idle \leq R$
shows $Q \sqcap spec.pre\ P \longrightarrow_+ R = spec.pre\ P \longrightarrow_H (Q \longrightarrow_+ R)$ (**is** *?lhs* = *?rhs*)
and $spec.pre\ P \sqcap Q \longrightarrow_+ R = spec.pre\ P \longrightarrow_H (Q \longrightarrow_+ R)$ (**is** *?thesis1*)
proof −
show *?lhs* = *?rhs*
proof(*rule antisym*[*OF* - *spec.singleton-le-extI*])
show *?lhs* ≤ *?rhs*
by (*simp add*: *heyting spec.next-imp.discharge*)
show $\langle\sigma\rangle \leq$ *?lhs* **if** $\langle\sigma\rangle \leq$ *?rhs* **for** σ
using *assms that*
by (*clarsimp simp*: *spec.singleton.next-imp-le-conv spec.singleton.heyting-le-conv*
spec.singleton.le-conv)
(*metis order.refl append-self-conv2 spec.singleton.idle-le-conv spec.singleton-le-ext-conv*
trace.less(*1*) *trace.less-eqE trace.steps'.simps*(*1*) *trace.t.sel*(*1*))
qed
then show *?thesis1*
by (*simp add*: *ac-simps*)
qed

setup ‹*Sign.parent-path*›

setup ‹*Sign.parent-path*›

10.1 Compositionality ala Abadi and Merz (and Lamport)

The main theorem for this implication (Abadi and Merz (1995, Theorem 4) and Abadi and Merz (1996, Corollary 5.1)) shows how to do assumption/commitment proofs for TLA considered as an algebraic logic. See also Cau and Collette (1996).

setup ‹*Sign.mandatory-path spec*›

lemma *Abadi-Lamport-Lemma5*:
shows $(\bigsqcap i{\in}I.\ P\ i \longrightarrow_+ Q\ i) \leq (\bigsqcap i{\in}I.\ P\ i) \longrightarrow_+ (\bigsqcap i{\in}I.\ Q\ i)$
by (*simp add*: *spec.next-imp.InfR INF-lower INF-superset-mono image-image spec.next-imp.mono*)

lemma *Abadi-Merz-Prop2-1*:

shows $(P \longrightarrow_+ Q) \sqcap (P \longrightarrow_+ (Q \longrightarrow_H R)) \leq P \longrightarrow_+ R$
by (*metis heyting.detachment*(*1*) *inf-sup-ord*(*2*) *spec.next-imp.infR*)

lemma *Abadi-Merz-Theorem3-5*:
shows $P \longrightarrow_H (Q \longrightarrow_H R) \leq (R \longrightarrow_+ Q) \longrightarrow_H (P \longrightarrow_+ Q)$
by (*simp add*: *heyting order.trans*[*OF spec.next-imp.heytingL*] *spec.next-imp.Abadi-Lamport-Lemma7*
flip: *heyting.curry-conv*)

theorem *Abadi-Merz-Theorem4*:
shows $(A \sqcap (\bigsqcap i{\in}I.\ Cs\ i) \longrightarrow_H (\bigsqcap i{\in}I.\ As\ i))$
$\sqcap (A \longrightarrow_+ ((\bigsqcap i{\in}I.\ Cs\ i) \longrightarrow_H C))$
$\sqcap (\bigsqcap i{\in}I.\ As\ i \longrightarrow_+ Cs\ i)$
$\leq A \longrightarrow_+ C$ (**is** *?lhs* $\leq$ *?rhs*)
proof −
have *?lhs* $\leq A \longrightarrow_H (\bigsqcap i{\in}I.\ Cs\ i) \longrightarrow_H (\bigsqcap i{\in}I.\ As\ i)$
by (*simp add*: *heyting.curry-conv inf.coboundedI1*)
then have *2*: *?lhs* $\leq ((\bigsqcap i{\in}I.\ As\ i) \longrightarrow_+ (\bigsqcap i{\in}I.\ Cs\ i)) \longrightarrow_H (A \longrightarrow_+ (\bigsqcap i{\in}I.\ Cs\ i))$
by (*simp add*: *heyting.curry-conv inf.coboundedI1 spec.Abadi-Merz-Theorem3-5*)
have *3*: *?lhs* $\leq (\bigsqcap i{\in}I.\ As\ i) \longrightarrow_+ (\bigsqcap i{\in}I.\ Cs\ i)$
using *spec.Abadi-Lamport-Lemma5 le-infI2* **by** *blast*
from *2 3* **have** *?lhs* $\leq A \longrightarrow_+ (\bigsqcap i{\in}I.\ Cs\ i)$
using *heyting.mp* **by** *blast*
then show *?thesis*
by − (*rule order.trans*[*OF - spec.Abadi-Merz-Prop2-1*[**where** *Q*=$\bigsqcap$ (*Cs* ‘ *I*)]]; *simp add*: *inf.coboundedI1*)
qed

setup ‹*Sign.parent-path*›

11 Stability

The essence of rely/guarantee reasoning is that sequential assertions must be *stable* with respect to interfering transitions as expressed in a *rely* relation. Formally an assertion P is stable if it becomes no less true for each transition in the rely r. This is essentially monotonicity, or that the extension of P is r-closed.
References:

- Vafeiadis (2008, §3.1.3) has a def for stability in terms of separation logic

definition *stable* :: *'a rel* ⇒ *'a pred* ⇒ *bool* **where**
stable r P = *monotone* (λ*x y*. (*x*, *y*) ∈ *r*) (≤) *P*

setup ‹*Sign.mandatory-path stable*›

named-theorems *intro stability intro rules*

lemma *singleton-conv*:
shows *stable* {(*s*, *s'*)} *P* ⟷ (*P s* ⟶ *P s'*)
by (*simp add*: *stable-def monotone-def le-bool-def*)

lemma *closed*:
shows *stable r P* ⟷ *r* ‘‘ *Collect P* ⊆ *Collect P*
unfolding *stable-def monotone-def le-bool-def* **by** *auto*

lemma *rtrancl-conv*:
shows *stable* (*r**) = *stable r*
by (*auto simp*: *stable-def monotone-def le-bool-def fun-eq-iff elim*!: *rtrancl-induct*)

lemma *reflcl-conv*:

shows *stable* $(r^=)$ = *stable r*
unfolding *stable-def monotone-def* **by** *simp*

lemma *empty*[*stable.intro*]:
shows *stable* {} *P*
unfolding *stable-def* **by** *simp*

lemma [*stable.intro*]:
shows *Id*: *stable Id P*
and *Id-fst*: $\bigwedge P$. *stable* (*Id* $\times_R$ *A*) (λs. *P* (*fst s*))
and *Id-fst-fst-snd*: $\bigwedge P$. *stable* (*Id* $\times_R$ *Id* $\times_R$ *A*) (λs. *P* (*fst s*) (*fst* (*snd s*)))
by (*simp-all add*: *stable-def monotone-def*)

lemma *UNIV*:
shows *stable UNIV P* $\longleftrightarrow$ ($\exists\, c$. *P* = $\langle c \rangle$)
unfolding *stable-def monotone-def le-bool-def* **by** *simp meson*

lemma *antimono-rel*:
shows *antimono* (λr. *stable r P*)
unfolding *stable-def monotone-def* **using** *subset-iff* **by** (*fastforce intro*: *antimonoI*)

lemmas *strengthen-rel*[*strg*] = *st-ord-antimono*[*OF stable.antimono-rel*, *unfolded le-bool-def*]

lemma *infI*:
assumes *stable r P*
shows *infI1*: *stable* (*r* $\cap$ *s*) *P*
and *infI2*: *stable* (*s* $\cap$ *r*) *P*
using *assms* **unfolding** *stable-def monotone-def* **by** *simp-all*

lemma *UNION-conv*:
shows *stable* ($\bigcup x{\in}X$. *r x*) *P* $\longleftrightarrow$ ($\forall\, x{\in}X$. *stable* (*r x*) *P*)
unfolding *stable-def monotone-def* **by** *blast*

lemmas *UNIONI*[*stable.intro*] = *iffD2*[*OF stable.UNION-conv*, *rule-format*]

lemma *Union-conv*:
shows *stable* ($\bigcup X$) *P* $\longleftrightarrow$ ($\forall\, x{\in}X$. *stable x P*)
unfolding *stable-def monotone-def* **by** *blast*

lemma *union-conv*:
shows *stable* (*r* $\cup$ *s*) *P* $\longleftrightarrow$ *stable r P* $\wedge$ *stable s P*
unfolding *stable-def monotone-def* **by** *blast*

lemmas *UnionI*[*stable.intro*] = *iffD2*[*OF stable.Union-conv*, *rule-format*]
lemmas *unionI*[*stable.intro*] = *iffD2*[*OF stable.union-conv*, *rule-format*, *unfolded conj-explode*]

Properties of *stable* **with respect to the predicate** **lemma** *const*[*stable.intro*]:
shows *stable r* $\langle c \rangle$
and *stable r* $\bot$
and *stable r* $\top$
by (*simp-all add*: *stable-def monotone-def*)

lemma *conjI*[*stable.intro*]:
assumes *stable r P*
assumes *stable r Q*
shows *stable r* (*P* $\wedge$ *Q*)
using *assms* **by** (*simp add*: *stable-def*)

lemma *disjI*[*stable.intro*]:
assumes *stable r P*
assumes *stable r Q*
shows *stable r* (*P* ∨ *Q*)
using *assms* **by** (*simp add*: *stable-def monotone-def le-bool-def*)

lemma *implies-constI*[*stable.intro*]:
assumes *P* ⟹ *stable r Q*
shows *stable r* (λ*s*. *P* ⟶ *Q s*)
using *assms* **by** (*auto simp*: *stable-def monotone-def le-bool-def*)

lemma *allI*[*stable.intro*]:
assumes ⋀*x*. *stable r* (*P x*)
shows *stable r* (∀ *x*. *P x*)
using *assms* **by** (*simp add*: *stable-def monotone-def le-bool-def*)

lemma *ballI*[*stable.intro*]:
assumes ⋀*x*. *x* ∈ *X* ⟹ *stable r* (*P x*)
shows *stable r* (λ*s*. ∀ *x*∈*X*. *P x s*)
using *assms* **by** (*simp add*: *stable-def monotone-def le-bool-def*)

lemma *stable-relprod-fstI*[*stable.intro*]:
assumes *stable r P*
shows *stable* (*r* ×$_R$ *s*) (λ*s*. *P* (*fst s*))
using *assms* **by** (*clarsimp simp*: *stable-def monotone-def*)

lemma *stable-relprod-sndI*[*stable.intro*]:
assumes *stable s P*
shows *stable* (*r* ×$_R$ *s*) (λ*s*. *P* (*snd s*))
using *assms* **by** (*clarsimp simp*: *stable-def monotone-def*)

lemma *local-only*: — for predicates that are insensitive to the shared state
assumes ⋀*ls s s'*. *P* (*ls*, *s*) ⟷ *P* (*ls*, *s'*)
shows *stable* (*Id* ×$_R$ *UNIV*) *P*
using *assms* **by** (*fastforce simp*: *stable-def monotone-def le-bool-def*)

lemma *Id-on-proj*:
assumes ⋀*v*. *stable Id*$_f$ (λ*s*. *P v s*)
shows *stable Id*$_f$ (λ*s*. *P* (*f s*) *s*)
using *assms* **unfolding** *stable-def* **by** (*rule monotone-Id-on-proj*)

lemma *if-const-conv*:
shows *stable r* (*if c then P else Q*) ⟷ *stable r* (λ*s*. *c* ⟶ *P s*) ∧ *stable r* (λ*s*. ¬*c* ⟶ *Q s*)
by (*simp add*: *stable-def*)

lemma *ifI*[*stable.intro*]:
assumes *stable r* (λ*s*. *c s* ⟶ *P s*)
assumes *stable r* (λ*s*. ¬*c s* ⟶ *Q s*)
shows *stable r* (λ*s*. *if c s then P s else Q s*)
using *assms* **by** (*simp add*: *stable.intro*)

lemma *ifI2*[*stable.intro*]:
assumes *stable r* (λ*s*. *c s* ⟶ *P s s*)
assumes *stable r* (λ*s*. ¬*c s* ⟶ *Q s s*)
shows *stable r* (λ*s*. (*if c s then P s else Q s*) *s*)
using *assms* **by** (*simp add*: *stable.intro*)

lemma *case-optionI*[*stable.intro*]:

assumes *stable r* (λ*s*. *opt s* = *None* ⟶ *none s*)
assumes ⋀*v*. *stable r* (λ*s*. *opt s* = *Some v* ⟶ *some v s*)
shows *stable r* (λ*s*. *case opt s of None* ⇒ *none s* | *Some v* ⇒ *some v s*)
using *assms* **by** (*simp add*: *stable.intro split*: *option.split*)

lemma *case-optionI2*[*stable.intro*]:
assumes *opt* = *None* ⟹ *stable r none*
assumes ⋀*v*. *opt* = *Some v* ⟹ *stable r* (*some v*)
shows *stable r* (*case opt of None* ⇒ *none* | *Some v* ⇒ *some v*)
using *assms* **by** (*simp add*: *stable.intro split*: *option.split*)

In practice the following rules are often too weak

lemma *impliesI*:
assumes *stable r* (¬*P*)
assumes *stable r Q*
shows *stable r* (*P* ⟶ *Q*)
using *assms* **by** (*auto simp*: *stable-def monotone-def le-bool-def*)

lemma *exI*:
assumes ⋀*x*. *stable r* (*P x*)
shows *stable r* (∃ *x*. *P x*)
using *assms* **by** (*auto simp*: *stable-def monotone-def le-bool-def*)

lemma *bexI*:
assumes ⋀*x*. *x* ∈ *X* ⟹ *stable r* (*P x*)
shows *stable r* (λ*s*. ∃ *x*∈*X*. *P x s*)
using *assms* **by** (*auto simp*: *stable-def monotone-def le-bool-def*)

setup ‹*Sign.parent-path*›

12 Refinement

We develop a refinement story for the (′*a*, ′*s*, ′*v*) *spec* lattice.
References:

- Vafeiadis (2008) (RGsep, program logic) and Liang, Feng, and Fu (2014) (RGsim, refinement)
- Armstrong et al. (2014)
- van Staden (2015)

definition *refinement* :: ′*s pred* ⇒ (′*a*, ′*s*, ′*v*) *spec* ⇒ (′*a*, ′*s*, ′*v*) *spec* ⇒ (′*v* ⇒ ′*s pred*) ⇒ (′*a*, ′*s*, ′*v*) *spec* (‹⦃-⦄, - ⊨ -, ⦃-⦄› [*0*,*0*,*0*,*0*] *100*) **where**
⦃*P*⦄, *A* ⊨ *G*, ⦃*Q*⦄ = *spec.pre P* ⊓ *A* ⟶₊ *G* ⊓ *spec.post Q*

An intuitive gloss on the proposition $c \leq \{\!|P|\!\}, A \Vdash G, \{\!|Q|\!\}$ is: assuming the precondition P holds and all steps conform to the process A, then c is a refinement of G and satisfies the postcondition Q.
Observations:

- We use *next-imp* here;$(\longrightarrow_H)$ is (only) enough for an assume/guarantee program logic (see §12.2)
- A is arbitrary but is intended to constrain only *env* steps
  - similarly termination can depend on A: a parallel composition can only terminate if all of its constituent processes terminate
- As $P \longrightarrow_+ Q$ implies $idle \leq Q$, in practice $idle \leq G$
- see §13.4.1 for some introduction rules

setup ‹*Sign.mandatory-path refinement*›

lemma *E*:
  assumes $c \leq \{\!|P|\!\}, A \Vdash G, \{\!|Q|\!\}$
  obtains $c \leq$ *spec.pre* $P \sqcap A \longrightarrow_+ G$
    and $c \leq$ *spec.pre* $P \sqcap A \longrightarrow_+$ *spec.post* Q
using *assms* **by** (*simp add*: *refinement-def spec.next-imp.infR*)

lemma *pre-post-cong*:
  assumes $P = P'$
  assumes $Q = Q'$
  shows $\{\!|P|\!\}, A \Vdash G, \{\!|Q|\!\} = \{\!|P'|\!\}, A \Vdash G, \{\!|Q'|\!\}$
using *assms* **by** *simp*

lemma *top*:
  shows $\{\!|P|\!\}, A \Vdash \top, \{\!|\top|\!\} = \top$
    and $\{\!|P|\!\}, A \Vdash \top, \{\!|\langle\top\rangle|\!\} = \top$
    and $\{\!|P|\!\}, A \Vdash \top, \{\!|\lambda\text{- -. } \mathit{True}|\!\} = \top$
by (*simp-all add*: *refinement-def*)

lemma *mcont2mcont*[*cont-intro*]:
  assumes *mcont luba orda Sup* $(\leq)$ G
  shows *mcont luba orda Sup* $(\leq)$ $(\lambda x.\ \{\!|P|\!\}, A \Vdash G\ x, \{\!|Q|\!\})$
by (*simp add*: *assms refinement-def*)

lemma *mono*:
  assumes $P' \leq P$
  assumes $A' \leq A$
  assumes $G \leq G'$
  assumes $Q \leq Q'$
  shows $\{\!|P|\!\}, A \Vdash G, \{\!|Q|\!\} \leq \{\!|P'|\!\}, A' \Vdash G', \{\!|Q'|\!\}$
unfolding *refinement-def*
apply (*strengthen ord-to-strengthen*(*1*)[*OF assms*(*1*)])
apply (*strengthen ord-to-strengthen*(*1*)[*OF assms*(*2*)])
apply (*strengthen ord-to-strengthen*(*1*)[*OF assms*(*3*)])
apply (*strengthen ord-to-strengthen*(*1*)[*OF assms*(*4*)])
apply *simp*
done

lemma *strengthen*[*strg*]:
  assumes *st-ord* $(\neg F)$ P P'
  assumes *st-ord* $(\neg F)$ A A'
  assumes *st-ord* F G G'
  assumes *st-ord* F Q Q'
  shows *st-ord* F $(\{\!|P|\!\}, A \Vdash G, \{\!|Q|\!\})$ $(\{\!|P'|\!\}, A' \Vdash G', \{\!|Q'|\!\})$
using *assms* **by** (*cases F*; *simp add*: *refinement.mono*)

lemma *mono-stronger*:
  assumes $P' \leq P$
  assumes *spec.pre* $P' \sqcap A' \leq A$
  assumes *spec.pre* $P' \sqcap G \leq A' \longrightarrow_+ G'$
  assumes $Q \leq Q'$
  assumes *spec.idle* $\leq G'$
  shows $\{\!|P|\!\}, A \Vdash G, \{\!|Q|\!\} \leq \{\!|P'|\!\}, A' \Vdash G', \{\!|Q'|\!\}$
unfolding *refinement-def*
apply (*strengthen ord-to-strengthen*(*2*)[*OF assms*(*1*)])
apply (*strengthen ord-to-strengthen*(*2*)[*OF assms*(*2*)])
apply (*strengthen ord-to-strengthen*(*2*)[*OF assms*(*4*)])

apply (*simp add*: *spec.next-imp.infR*)
apply (*metis assms*(*3*) *heyting.commute le-infI1*
spec.next-imp.closure.cl spec.pre.next-imp-eq-heyting(*2*)[*OF assms*(*5*)])
done

lemmas *pre-ag* = *order.trans*[*OF - refinement.mono*[*OF order.refl - - order.refl*], *of c*] **for** *c*
lemmas *pre-a* = *refinement.pre-ag*[*OF - - order.refl*]
lemmas *pre-g* = *refinement.pre-ag*[*OF - order.refl*]

lemma *pre*:
assumes $c \leq \{\!|P|\!\}, A \Vdash G, \{\!|Q|\!\}$
assumes $\bigwedge s.\ P'\ s \Longrightarrow P\ s$
assumes $A' \leq A$
assumes $G \leq G'$
assumes $\bigwedge s\ v.\ Q\ s\ v \Longrightarrow Q'\ s\ v$
shows $c \leq \{\!|P'|\!\}, A' \Vdash G', \{\!|Q'|\!\}$
using *assms*(*2*−) **by** (*blast intro*: *order.trans*[*OF assms*(*1*) *refinement.mono*])

lemmas *pre-pre-post* = *refinement.pre*[*OF - - order.refl order.refl*, *of c*] **for** *c*

lemma *pre-imp*:
assumes $\bigwedge s.\ P\ s \Longrightarrow P'\ s$
assumes $c \leq \{\!|P'|\!\}, A \Vdash G, \{\!|Q|\!\}$
shows $c \leq \{\!|P|\!\}, A \Vdash G, \{\!|Q|\!\}$
using *assms refinement.pre* **by** *blast*

lemmas *pre-pre* = *refinement.pre-imp*[*rotated*]

lemma *post-imp*:
assumes $\bigwedge v\ s.\ Q\ v\ s \Longrightarrow R\ v\ s$
assumes $c \leq \{\!|P|\!\}, A \Vdash G, \{\!|Q|\!\}$
shows $c \leq \{\!|P|\!\}, A \Vdash G, \{\!|R|\!\}$
using *assms refinement.pre* **by** *blast*

lemmas *pre-post* = *refinement.post-imp*[*rotated*]
lemmas *strengthen-post* = *refinement.pre-post*

lemma *pre-inf-assume*:
shows $\{\!|P|\!\}, A \Vdash G, \{\!|Q|\!\} = \{\!|P|\!\}, A \sqcap \mathit{spec.pre}\ P \Vdash G, \{\!|Q|\!\}$
by (*simp add*: *refinement-def ac-simps*)

lemma *pre-assume-absorb*:
assumes $A \leq \mathit{spec.pre}\ P$
shows $\{\!|P|\!\}, A \Vdash G, \{\!|Q|\!\} = \{\!|\top|\!\}, A \Vdash G, \{\!|Q|\!\}$
by (*simp add*: *assms refinement-def inf-absorb2*)

lemmas *sup* = *sup-least*[**where** x=$\{\!|P|\!\}, A \Vdash G, \{\!|Q|\!\}$] **for** *A G P Q*

lemma
shows *supRL*: $c \leq \{\!|P|\!\}, A \Vdash G_1, \{\!|Q|\!\} \Longrightarrow c \leq \{\!|P|\!\}, A \Vdash G_1 \sqcup G_2, \{\!|Q|\!\}$
and *supRR*: $c \leq \{\!|P|\!\}, A \Vdash G_2, \{\!|Q|\!\} \Longrightarrow c \leq \{\!|P|\!\}, A \Vdash G_1 \sqcup G_2, \{\!|Q|\!\}$
by (*simp-all add*: *refinement.pre-g*)

lemma *infR-conv*:
shows $\{\!|P|\!\}, A \Vdash G_1 \sqcap G_2, \{\!|Q_1 \sqcap Q_2|\!\} = \{\!|P|\!\}, A \Vdash G_1, \{\!|Q_1|\!\} \sqcap \{\!|P|\!\}, A \Vdash G_2, \{\!|Q_2|\!\}$
by (*simp add*: *refinement-def ac-simps spec.next-imp.infR spec.post.inf*)

lemma *inf-le*:

shows $X \sqcap \{\!|P|\!\}, A \Vdash G, \{\!|Q|\!\} \leq \{\!|P|\!\}, X \sqcap A \Vdash X \sqcap G, \{\!|Q|\!\}$
by (*simp add*: *refinement-def le-infI1 le-infI2*
spec.next-imp.infR spec.next-imp.mono spec.next-imp.contains)

lemma *heyting-le*:
shows $\{\!|P|\!\}, A \sqcap B \Vdash B \longrightarrow_H G, \{\!|Q|\!\} \leq B \longrightarrow_H \{\!|P|\!\}, A \Vdash G, \{\!|Q|\!\}$
by (*simp add*: *refinement-def ac-simps heyting.infR heyting.commute*
spec.next-imp.heytingL-distrib spec.next-imp.mono)

lemma *heyting-pre*:
assumes *spec.idle* $\leq G$
shows *spec.pre* $P \longrightarrow_H \{\!|P'|\!\}, A \Vdash G, \{\!|Q|\!\} = \{\!|P \wedge P'|\!\}, A \Vdash G, \{\!|Q|\!\}$
by (*simp add*: *ac-simps refinement-def spec.pre.conj assms spec.idle.post-le*
flip: *spec.pre.next-imp-eq-heyting*)

lemma *sort-of-refl*:
assumes $c \leq \{\!|P|\!\}, A \Vdash G, \{\!|Q|\!\}$
shows $c \leq \{\!|P|\!\}, A \Vdash c, \{\!|Q|\!\}$
using *assms* **by** (*simp add*: *refinement-def spec.next-imp.infR spec.next-imp.closure.expansive*)

lemma *gen-asm-base*:
assumes $P \Longrightarrow c \leq \{\!|P' \wedge P''|\!\}, A \Vdash G, \{\!|Q|\!\}$
assumes *spec.idle* $\leq G$
shows $c \leq \{\!|P' \wedge \langle P \rangle \wedge P''|\!\}, A \Vdash G, \{\!|Q|\!\}$
using *assms* **by** (*simp add*: *refinement-def spec.pre.conj spec.pre.K spec.next-imp.botL spec.idle.post-le*)

lemmas *gen-asm* =
refinement.gen-asm-base[**where** P'=$\langle True \rangle$ **and** P''=$\langle True \rangle$, *simplified*]
refinement.gen-asm-base[**where** P'=$\langle True \rangle$, *simplified*]
refinement.gen-asm-base[**where** P''=$\langle True \rangle$, *simplified*]
refinement.gen-asm-base

lemma *post-conj*:
assumes $c \leq \{\!|P|\!\}, A \Vdash G, \{\!|Q|\!\}$
assumes $c \leq \{\!|P|\!\}, A \Vdash G, \{\!|Q'|\!\}$
shows $c \leq \{\!|P|\!\}, A \Vdash G, \{\!|\lambda rv.\ Q\ rv \wedge Q'\ rv|\!\}$
using *assms* **unfolding** *refinement-def* **by** (*simp add*: *spec.post.conj spec.next-imp.infR*)

lemma *conj-lift*:
assumes $c \leq \{\!|P|\!\}, A \Vdash G, \{\!|Q|\!\}$
assumes $c \leq \{\!|P'|\!\}, A \Vdash G, \{\!|Q'|\!\}$
shows $c \leq \{\!|P \wedge P'|\!\}, A \Vdash G, \{\!|\lambda rv.\ Q\ rv \wedge Q'\ rv|\!\}$
using *assms* **by** (*blast intro*: *refinement.post-conj refinement.pre*)

lemma *drop-imp*:
assumes $c \leq \{\!|P|\!\}, A \Vdash G, \{\!|Q|\!\}$
shows $c \leq \{\!|P|\!\}, A \Vdash G, \{\!|\lambda rv.\ Q'\ rv \longrightarrow Q\ rv|\!\}$
using *assms refinement.strengthen-post* **by** *fastforce*

lemma *prop*:
shows $c \leq \{\!|\langle P \rangle|\!\}, A \Vdash c, \{\!|\lambda v.\ \langle P \rangle|\!\}$
by (*simp add*: *refinement.sort-of-refl*[**where** G=$\top$] *refinement.gen-asm refinement.top*)

lemma *name-pre-state*:
assumes $\bigwedge s.\ P\ s \Longrightarrow c \leq \{\!|(=)\ s|\!\}, A \Vdash G, \{\!|Q|\!\}$
assumes *spec.idle* $\leq G$
shows $c \leq \{\!|P|\!\}, A \Vdash G, \{\!|Q|\!\}$ (**is** *?lhs* $\leq$ *?rhs*)
proof −

```
  have ⦇σ⦈ ≤ G ∧ ⦇σ⦈ ≤ spec.post Q
    if ⦇σ⦈ ≤ c
   and ∀σ''<σ. ⦇σ''⦈ ≤ spec.pre P ∧ ⦇σ''⦈ ≤ A
   for σ
  proof(cases trace.rest σ = [] ∧ trace.term σ = None)
    case True with ‹spec.idle ≤ G› show ?thesis
      by (cases σ) (simp add: spec.singleton.le-conv order.trans[OF spec.idle.minimal-le])
  next
    case False with order.trans[OF ‹⦇σ⦈ ≤ c› assms(1)[where s=trace.init σ]] that(2)
    show ?thesis
      by (cases σ)
         (clarsimp simp: refinement-def spec.singleton.next-imp-le-conv spec.singleton.le-conv;
          fastforce simp: trace.less dest: spec[where x=trace.T (trace.init σ) [] None])
  qed
  then show ?thesis
    by − (rule spec.singleton-le-extI;
          auto simp: refinement-def spec.singleton.next-imp-le-conv
               intro: order.trans[OF spec.singleton.mono])
qed

setup ‹Sign.parent-path›
```

12.1 Geenral rules for the *('a, 's, 'v) spec* lattice

```
setup ‹Sign.mandatory-path spec›

setup ‹Sign.mandatory-path term›

setup ‹Sign.mandatory-path all›

lemma refinement:
  shows spec.term.all (⦃P⦄, A ⊨ G, ⦃Q⦄) = ⦃P⦄, spec.term.all A ⊨ spec.term.all G, ⦃⊤⦄
by (simp add: refinement-def spec.term.all.next-imp spec.term.all.inf spec.term.all.pre spec.term.all.post)

setup ‹Sign.parent-path›

setup ‹Sign.mandatory-path none›

lemma refinement-le:
  shows spec.term.none (⦃P⦄, A ⊨ G, ⦃Q⦄) ≤ ⦃P⦄, spec.term.all A ⊨ spec.term.all G, ⦃⊥⦄
by (simp add: spec.term.galois spec.term.all.refinement order.trans[OF spec.term.all.expansive])

setup ‹Sign.parent-path›

setup ‹Sign.parent-path›

setup ‹Sign.mandatory-path invmap›

lemma refinement:
  fixes af :: 'a ⇒ 'b
  fixes sf :: 's ⇒ 't
  fixes vf :: 'v ⇒ 'w
  fixes A :: ('b, 't, 'w) spec
  fixes G :: ('b, 't, 'w) spec
  fixes P :: 't pred
  fixes Q :: 'w ⇒ 't pred
  shows spec.invmap af sf vf (⦃P⦄, A ⊨ G, ⦃Q⦄)
     = (⦃λs. P (sf s)⦄, spec.invmap af sf vf A ⊨ spec.invmap af sf vf G, ⦃λv s. Q (vf v) (sf s)⦄)
```

unfolding *refinement-def*
by (*simp only*: *spec.next-imp.invmap spec.invmap.inf spec.invmap.pre spec.invmap.post*)

setup ‹*Sign.parent-path*›

setup ‹*Sign.parent-path*›

12.1.1 Actions

Actions are anchored at the start of a trace, and therefore must be an initial step of the assume A. However by the semantics of $(\longrightarrow_+)$ we may only know that that initial state of the trace is acceptable to A when showing that F-steps are F'-steps (the second assumption). This hypothesis is vacuous when $idle \leq A$.

setup ‹*Sign.mandatory-path refinement.spec*›

lemma *action*:
  fixes $F :: ('v \times 'a \times 's \times 's)\ set$
  assumes $\bigwedge v\ a\ s\ s'.\ [\![P\ s;\ (v,\ a,\ s,\ s') \in F;\ (a,\ s,\ s') \in spec.initial\text{-}steps\ A]\!] \Longrightarrow Q\ v\ s'$
  assumes $\bigwedge v\ a\ s\ s'.\ [\![P\ s;\ (v,\ a,\ s,\ s') \in F;\ (a,\ s,\ s) \in spec.initial\text{-}steps\ A]\!] \Longrightarrow (v,\ a,\ s,\ s') \in F'$
  shows $spec.action\ F \leq \{\!|P|\!\},\ A \Vdash spec.action\ F',\ \{\!|Q|\!\}$
proof −
  have $spec.action\ (F \cap UNIV \times UNIV \times Pre\ P) \leq A \longrightarrow_+ spec.action\ F' \sqcap spec.post\ Q$
  proof(*induct rule*: *spec.action-le*)
    case *idle* **show** *?case*
      by (*simp add*: *spec.next-imp.contains spec.idle.action-le spec.idle.post-le*)
  next
    case (*step v a s s'*) **then show** *?case*
      by (*fastforce simp*: *spec.singleton.next-imp-le-conv trace.split-all spec.initial-steps-def*
                trace.less Cons-eq-append-conv spec.singleton.post-le-conv
                order.trans[*OF spec.idle.minimal-le spec.idle.action-le*]
              elim: *assms trace.less-eqE prefixE*
              intro: *spec.action.stepI*)
  qed
  then show *?thesis*
    by (*simp add*: *refinement-def spec.pre.next-imp-eq-heyting spec.idle.post-le spec.idle.action-le*
          heyting order.trans[*OF spec.pre.inf-action-le*(*2*)])
qed

lemma *return*:
  shows $spec.return\ v \leq \{\!|Q\ v|\!\},\ A \Vdash spec.return\ v,\ \{\!|Q|\!\}$
by (*auto simp*: *spec.return-def intro*: *refinement.spec.action*)

lemma *action-rel*:
  fixes $F :: ('v \times 'a \times 's \times 's)\ set$
  assumes $\bigwedge v\ a\ s\ s'.\ [\![P\ s;\ (v,\ a,\ s,\ s') \in F;\ (a,\ s,\ s') \in spec.initial\text{-}steps\ A]\!] \Longrightarrow Q\ v\ s'$
  assumes $\bigwedge v\ a\ s\ s'.\ [\![P\ s;\ (v,\ a,\ s,\ s') \in F;\ (a,\ s,\ s) \in spec.initial\text{-}steps\ A;\ s \neq s']\!] \Longrightarrow (a,\ s,\ s') \in r$
  shows $spec.action\ F \leq \{\!|P|\!\},\ A \Vdash spec.rel\ r,\ \{\!|Q|\!\}$
by (*force simp*: *spec.rel-def spec.rel.act-def spec.term.all.action*
      intro: *assms refinement.supRL refinement.spec.action*
        refinement.pre-g[*OF - spec.term.all.mono*[*OF spec.kleene.expansive-star*]])

setup ‹*Sign.parent-path*›

12.1.2 Bind

Consider showing $f \gg\!= g \leq f' \gg\!= g'$ under the assume A and pre/post conditions P/Q.
The tricky part is to residuate the assume A wrt the process f for use in the refinement proof of g.

- we want to preserve as much of the structure of A as possible

- intuition: we want all the ways a trace can continue in A having started with a terminating trace in f
- intuitively a right residual for $(\gg\!=)$ should do the job
  - however unlike Hoare and He (1987) we have no chance of a right residual for $(\gg\!=)$ as we use traces (they use relations)
    - i.e., if it is not the case that $f \gg\!= \bot \leq A$ then there is no continuation h such that $f \gg\!= h \leq A$.
    - also such a residual h has arbitrary behavior starting from states that f cannot reach
      - i.e., for traces $\neg\sigma \leq f \langle\!|\sigma|\!\rangle \gg h \leq A$ need not hold
      - and the user-provided assertions may be too weak to correct for this
- in practice the termination information in the assume A is not useful

We therefore define an ad hoc residual that does the trick.
See Emerson (1983, §4) for some related concerns.

setup ‹*Sign.mandatory-path refinement.spec.bind*›

definition *res* :: ('*a*, '*s*, '*v*) *spec* ⇒ ('*a*, '*s*, '*w*) *spec* ⇒ '*v* ⇒ ('*a*, '*s*, '*w*) *spec* **where**
res f A v = ⨆{⟨|*trace.final'* *s xs*, *ys*, *w*|⟩ |*s xs ys w*. ⟨|*s*, *xs*, *Some v*|⟩ ≤ *f* ∧ ⟨|*s*, *xs* @ *ys*, *None*|⟩ ≤ *A*}

setup ‹*Sign.parent-path*›

setup ‹*Sign.mandatory-path spec.singleton.refinement.bind*›

lemma *res-le-conv*[*spec.singleton.le-conv*]:
shows ⟨|*σ*|⟩ ≤ *refinement.spec.bind.res f A v*
⟷ (∃ *s xs*. ⟨|*s*, *xs*, *Some v*|⟩ ≤ *f*
∧ *trace.init σ* = *trace.final'* *s xs*
∧ ⟨|*s*, *xs* @ *trace.rest σ*, *None*|⟩ ≤ *A*) (**is** *?lhs* ⟷ *?rhs*)
proof(*rule iffI*)
show *?lhs* ⟹ *?rhs*
by (*fastforce simp*: *refinement.spec.bind.res-def trace.split-Ex spec.singleton-le-conv*
trace.less-eq-None trace.natural'.append trace.natural-def
elim: *trace.less-eqE order.trans*[*rotated*])
show *?rhs* ⟹ *?lhs*
by (*cases σ*) (*clarsimp simp*: *refinement.spec.bind.res-def*; *blast*)
qed

setup ‹*Sign.parent-path*›

setup ‹*Sign.mandatory-path spec.term.none.refinement.bind*›

lemma *resL*:
shows *refinement.spec.bind.res* (*spec.term.none f*) *A v* = ⊥
by (*simp add*: *refinement.spec.bind.res-def spec.singleton.le-conv*)

lemma *resR*:
shows *refinement.spec.bind.res f* (*spec.term.none A*) *v* = *refinement.spec.bind.res f A v*
by (*simp add*: *refinement.spec.bind.res-def spec.singleton.le-conv*)

setup ‹*Sign.parent-path*›

setup ‹*Sign.mandatory-path spec.term.all.refinement.bind*›

lemma *resR-mono*:
shows *refinement.spec.bind.res f* (*spec.term.all A*) *v* = *refinement.spec.bind.res f A v*
by (*simp add*: *refinement.spec.bind.res-def spec.singleton.le-conv*)

(*meson dual-order.trans spec.singleton.less-eq-None*)

lemma *res*:
shows *spec.term.all* (*refinement.spec.bind.res f A v*) = *refinement.spec.bind.res f A v*
by (*rule spec.singleton.antisym*) (*auto simp*: *spec.singleton.le-conv*)

setup ‹*Sign.parent-path*›

setup ‹*Sign.mandatory-path spec.term.closed.refinement.bind*›

lemma *res*:
shows *refinement.spec.bind.res f A v* ∈ *spec.term.closed* -
by (*subst spec.term.all.refinement.bind.res*[*symmetric*]) (*rule spec.term.all.closed*)

setup ‹*Sign.parent-path*›

setup ‹*Sign.mandatory-path refinement.spec.bind.res*›

lemma *bot*:
shows *botL*: *refinement.spec.bind.res* ⊥ = ⊥
and *botR*: *refinement.spec.bind.res f* ⊥ = ⊥
by (*simp-all add*: *refinement.spec.bind.res-def fun-eq-iff spec.singleton.not-bot*)

lemma *mono*:
assumes $f \leq f'$
assumes $A \leq A'$
shows *refinement.spec.bind.res f A v* ≤ *refinement.spec.bind.res f' A' v*
using *assms* **unfolding** *refinement.spec.bind.res-def* **by** (*fastforce intro*!: *Sup-subset-mono*)

lemma *strengthen*[*strg*]:
assumes *st-ord F f f'*
assumes *st-ord F A A'*
shows *st-ord F* (*refinement.spec.bind.res f A v*) (*refinement.spec.bind.res f' A' v*)
using *assms* **by** (*cases F*; *simp add*: *refinement.spec.bind.res.mono*)

lemma *mono2mono*[*cont-intro*, *partial-function-mono*]:
assumes *monotone orda* (≤) *f*
assumes *monotone orda* (≤) *A*
shows *monotone orda* (≤) (λ*x*. *refinement.spec.bind.res* (*f x*) (*A x*) *v*)
using *assms* **by** (*simp add*: *monotone-def refinement.spec.bind.res.mono*)

lemma *SupL*:
shows *refinement.spec.bind.res* (⨆ *X*) *A v* = (⨆ *x*∈*X*. *refinement.spec.bind.res x A v*)
by (*rule antisym*; *simp add*: *refinement.spec.bind.res-def*; *blast*)

lemma *SupR*:
shows *refinement.spec.bind.res f* (⨆ *X*) *v* = (⨆ *x*∈*X*. *refinement.spec.bind.res f x v*)
by (*rule antisym*; *simp add*: *refinement.spec.bind.res-def*; *blast*)

lemma *InfL-le*:
shows *refinement.spec.bind.res* (⨅ *X*) *A v* ≤ (⨅ *x*∈*X*. *refinement.spec.bind.res x A v*)
by (*simp add*: *refinement.spec.bind.res-def le-Inf-iff*) *fast*

lemma *InfR-le*:
shows *refinement.spec.bind.res f* (⨅ *X*) *v* ≤ (⨅ *x*∈*X*. *refinement.spec.bind.res f x v*)
by (*simp add*: *refinement.spec.bind.res-def le-Inf-iff*) *fast*

lemma *mcont2mcont*[*cont-intro*]:

assumes *mcont luba orda Sup* $(\leq)$ *f*
assumes *mcont luba orda Sup* $(\leq)$ *A*
shows *mcont luba orda Sup* $(\leq)$ $(\lambda x.$ *refinement.spec.bind.res* $(f\ x)\ (A\ x)\ v)$
proof(*rule ccpo.mcont2mcont'*[*OF complete-lattice-ccpo - - assms*(*1*)])
show *mcont Sup* $(\leq)$ *Sup* $(\leq)$ $(\lambda f.$ *refinement.spec.bind.res f* $(A\ x)\ v)$ **for** *x*
by (*intro mcontI contI monotoneI*)
(*simp-all add*: *refinement.spec.bind.res.mono refinement.spec.bind.res.SupL*)
show *mcont luba orda Sup* $(\leq)$ $(\lambda x.$ *refinement.spec.bind.res f* $(A\ x)\ v)$ **for** *f*
by (*intro mcontI monotoneI contI*)
(*simp-all add*: *mcont-monoD*[*OF assms*(*2*)] *refinement.spec.bind.res.mono contD*[*OF mcont-cont*[*OF assms*(*2*)]]
refinement.spec.bind.res.SupR image-image)
qed

lemma *returnL*:
assumes *spec.idle* $\leq A$
shows *refinement.spec.bind.res* (*spec.return v*) *A v* $=$ *spec.term.all A* (**is** *?lhs* $=$ *?rhs*)
proof(*rule antisym*[*OF - spec.singleton-le-extI*])
show *?lhs* $\leq$ *?rhs*
by (*auto simp*: *refinement.spec.bind.res-def trace.split-all spec.singleton.le-conv*)
show $\langle\!|\sigma|\!\rangle \leq$ *?lhs* **if** $\langle\!|\sigma|\!\rangle \leq$ *?rhs* **for** σ
using *that*
by (*auto simp*: *spec.singleton.le-conv*
intro!: *exI*[**where** *x*=*trace.init* σ] *exI*[**where** *x*=[]]
elim: *order.trans*[*rotated*])
qed

lemma *rel-le*:
assumes $r \subseteq r'$
shows *refinement.spec.bind.res f* (*spec.rel r*) *v* $\leq$ *spec.rel r'*
using *assms* **by** (*force intro*: *spec.singleton-le-extI simp*: *spec.singleton.le-conv trace.steps'.append*)

setup ‹*Sign.parent-path*›

setup ‹*Sign.mandatory-path spec.steps.refinement.spec.bind*›

lemma *res-le*: — we can always discard the extra structure
shows *spec.steps* (*refinement.spec.bind.res f A v*) $\subseteq$ *spec.steps A*
by (*meson order-trans refinement.spec.bind.res.mono refinement.spec.bind.res.rel-le*
spec.rel.galois spec.rel.upper-lower-expansive)

setup ‹*Sign.parent-path*›

A refinement rule for $(\gg\!=)$. The function *vf* abstracts interstitial monadic return values.

setup ‹*Sign.mandatory-path refinement.spec*›

lemma *bind-abstract*:
fixes *f* :: $('a, 's, 'v)$ *spec*
fixes *f'* :: $('a, 's, 'v')$ *spec*
fixes *g* :: $'v \Rightarrow ('a, 's, 'w)$ *spec*
fixes *g'* :: $'v' \Rightarrow ('a, 's, 'w)$ *spec*
fixes *vf* :: $'v \Rightarrow 'v'$
assumes *g*: $\bigwedge v.\ g\ v \leq \{\!|Q'\ (vf\ v)|\!\}$, *refinement.spec.bind.res* (*spec.pre P* $\sqcap$ *spec.term.all A* $\sqcap$ *f'*) *A* (*vf v*) $\Vdash$ *g'* (*vf v*), $\{\!|Q|\!\}$
assumes *f*: *f* $\leq \{\!|P|\!\}$, *spec.term.all A* $\Vdash$ *spec.vinvmap vf f'*, $\{\!|\lambda v.\ Q'\ (vf\ v)|\!\}$
shows *f* $\gg\!=$ *g* $\leq \{\!|P|\!\}$, *A* $\Vdash$ *f'* $\gg\!=$ *g'*, $\{\!|Q|\!\}$
proof (*rule order.trans*[*OF spec.bind.mono*[*OF f g*]],
unfold refinement-def spec.bind.inf-post,

```
      induct rule: spec.bind-le)
  case incomplete show ?case
   apply (simp add: spec.term.galois spec.term.all.next-imp spec.term.all.bind spec.term.all.inf
                    spec.term.all.post spec.term.all.pre)
   apply (simp add: spec.next-imp.mono[OF order.refl] le-supI1 le-infI1 spec.term.none.invmap
                    spec.invmap.id
               flip: spec.term.galois)
   done
next
  case (continue σ_f σ_g v)
  have ⦇s, xs, w⦈ ≤ f' ⟜ (λv. g' v ⊓ spec.post Q)
   if le: trace.T s xs w ≤ trace.T (trace.init σ_f) (trace.rest σ_f @ trace.rest σ_g) (trace.term σ_g)
   and pre: ∀σ''<trace.T s xs w. ⦇σ''⦈ ≤ spec.pre P ⊓ A
   for s xs w
   using le
  proof(induct rule: trace.less-eq-extE)
   case prefix
   from prefix(3) show ?case
   proof(induct rule: prefix-append-not-NilE[case-names incomplete1 continue1])
    case incomplete1 with pre continue(1) prefix(1,2) show ?case
     apply (clarsimp simp: spec.singleton.next-imp-le-conv)
     apply (drule spec[where x=trace.T s xs None],
            drule mp[where P=trace.T s xs None ≤ σ_f])
      apply (force simp: spec.singleton.le-conv spec.map.singleton
                         le-inf-iff trace.less trace.split-All trace.less-eq-None
                  simp flip: spec.map-invmap.galois
                     intro!: spec.bind.incompleteI)+
     done
   next
    case (continue1 us)
    from continue(1,3) prefix(2) continue1(1,2)
         spec[OF pre, where x=trace.T (trace.init σ_f) (trace.rest σ_f) None]
    have ⦇trace.init σ_f, trace.rest σ_f, Some (vf v)⦈ ≤ spec.pre P ⊓ spec.term.all A ⊓ f' ⊓ spec.post Q'
     apply (cases σ_f)
     apply (clarsimp simp: spec.singleton.le-conv spec.singleton.next-imp-le-conv
                           trace.less le-inf-iff exI[where x=None]
                     split: option.split-asm
                     dest!: spec[where x=σ_f])
     apply (metis append-is-Nil-conv le-inf-iff pre trace.less-same-append-conv)
     done
    with pre continue(1,2,5) prefix(1,2) continue1
         spec[OF continue(4)[unfolded spec.singleton.next-imp-le-conv],
              where x=trace.T (trace.init σ_g) us None]
    show ?case
     apply clarsimp
     apply (rule spec.bind.continueI[where v=vf v], assumption)
     apply (clarsimp simp: spec.singleton.le-conv trace.split-All trace.less-eq-None trace.less)
     apply (metis append.assoc)
     done
   qed
  next
   case (maximal w')
   from maximal(1−3) continue(1,3)
        spec[OF pre, where x=trace.T (trace.init σ_f) (trace.rest σ_f) None]
   have ⦇trace.init σ_f, trace.rest σ_f, Some (vf v)⦈ ≤ spec.pre P ⊓ spec.term.all A ⊓ f' ⊓ spec.post Q'
    apply (cases σ_f)
    apply (clarsimp simp: spec.singleton.le-conv spec.singleton.next-imp-le-conv le-inf-iff
                          split: option.split-asm)
```

```
    apply (force simp: trace.less spec.singleton.mono trace.less-eq-same-append-conv
                elim: notE order.trans[rotated]
                dest!: spec[where x=σ_f] spec[where x=None])
    done
  with maximal(2−4) pre continue(2)
    spec[OF continue(4)[unfolded spec.singleton.next-imp-le-conv], where x=σ_g]
  show ?case
    by (cases σ_g)
      (auto 0 2 intro!: spec.bind.continueI[where v=vf v] exI[where x=s]
                simp: spec.singleton.le-conv trace.split-All trace.less)
  qed
  then show ?case
    by (clarsimp simp: spec.singleton.next-imp-le-conv trace.split-all)
qed

lemmas bind = refinement.spec.bind-abstract[where vf=id, simplified spec.invmap.id, simplified]
```

12.1.3 Interference

```
lemma rel-mono:
  assumes r ⊆ r'
  assumes stable (snd ‘ (spec.steps A ∩ r)) P
  shows spec.rel r ≤ ⟦P⟧, A ⊩ spec.rel r', ⟦λ-::unit. P⟧
apply (subst (1) spec.rel.monomorphic-conv)
using assms(2)
proof(induct arbitrary: A rule: spec.kleene.star.fixp-induct[case-names adm bot step])
  case (step R A)
  have ∗: spec.rel.act r ≤ ⟦P⟧, spec.term.all A ⊩ spec.rel r', ⟦⟨P⟩⟧
    unfolding spec.rel.act-def
  proof(rule refinement.spec.action-rel)
    show P s'
      if P s
     and (v, a, s, s') ∈ {()} × (r ∪ UNIV × Id)
     and (a, s, s') ∈ spec.initial-steps (spec.term.all A)
     for v a s s'
      using that monotoneD[OF stable.antimono-rel, unfolded le-bool-def, rule-format, OF - step.prems,
                           where x={(s, s')}]
      by (cases s = s';
          force simp: spec.initial-steps.term.all stable.singleton-conv
                dest: subsetD[OF spec.initial-steps.steps-le])
    show (a, s, s') ∈ r' if (v, a, s, s') ∈ {()} × (r ∪ UNIV × Id) and s ≠ s' for v a s s'
      using that assms(1) by fast
  qed
  show ?case
    apply (rule refinement.sup[OF - refinement.pre-g[OF refinement.spec.return spec.return.rel-le]])
    apply (subst spec.rel.unwind-bind)
    apply (rule refinement.spec.bind[OF step.hyps ∗])
    apply (force intro: monotoneD[OF stable.antimono-rel, unfolded le-bool-def, rule-format, OF - step.prems]
                dest: subsetD[OF spec.steps.refinement.spec.bind.res-le])
    done
qed simp-all

setup ‹Sign.parent-path›
```

12.1.4 Parallel

Our refinement rule for *Parallel* does not constrain the constituent processes in any way, unlike Abadi and Plotkin's proposed rule (see §9.2).

setup ‹*Sign.mandatory-path refinement.spec*›

definition — roughly the *Parallel* construction with roles reversed
env-hyp :: (*'a* ⇒ *'s pred*) ⇒ (*sequential*, *'s*, *unit*) *spec* ⇒ *'a set* ⇒ (*'a* ⇒ (*sequential*, *'s*, *unit*) *spec*) ⇒ *'a* ⇒ (*sequential*, *'s*, *unit*) *spec*
where
env-hyp P A as Ps a =
spec.pre (⨅ (*P* ‘ *as*))
⊓ *spec.amap* (*toConcurrent-fn* (*proc a*))
(*spec.rel* (({*env*} ∪ *proc* ‘ *as*) × *UNIV*)
⊓ (⨅ *i*∈*as*. *spec.toConcurrent i* (*Ps i*))
⊓ *spec.ainvmap toSequential-fn A*)

setup ‹*Sign.mandatory-path env-hyp*›

lemma *mono*:
assumes ⋀*a*. *a* ∈ *as* ⟹ *P a* ≤ *P' a*
assumes *A* ≤ *A'*
assumes ⋀*a*. *a* ∈ *as* ⟹ *Ps a* ≤ *Ps' a*
shows *refinement.spec.env-hyp P A as Ps a* ≤ *refinement.spec.env-hyp P' A' as Ps' a*
unfolding *refinement.spec.env-hyp-def*
apply (*strengthen ord-to-strengthen*(*2*)[*OF assms*(*1*)], *assumption*)
apply (*strengthen ord-to-strengthen*(*1*)[*OF assms*(*2*)])
apply (*strengthen ord-to-strengthen*(*1*)[*OF assms*(*3*)], *assumption*)
apply *simp*
done

lemma *strengthen*[*strg*]:
assumes ⋀*a*. *a* ∈ *as* ⟹ *st-ord F* (*P a*) (*P' a*)
assumes *st-ord F A A'*
assumes ⋀*a*. *a* ∈ *as* ⟹ *st-ord F* (*Ps a*) (*Ps' a*)
shows *st-ord F* (*refinement.spec.env-hyp P A as Ps a*) (*refinement.spec.env-hyp P' A' as Ps' a*)
using *assms* **by** (*cases F*; *simp add*: *refinement.spec.env-hyp.mono*)

setup ‹*Sign.parent-path*›

lemma *Parallel*:
fixes *A* :: (*sequential*, *'s*, *unit*) *spec*
fixes *Q* :: *'a* ⇒ *'s* ⇒ *bool*
fixes *Ps* :: *'a* ⇒ (*sequential*, *'s*, *unit*) *spec*
fixes *Ps'* :: *'a* ⇒ (*sequential*, *'s*, *unit*) *spec*
assumes ⋀*a*. *a* ∈ *as* ⟹ *Ps a* ≤ ⦃*P a*⦄, *refinement.spec.env-hyp P A as Ps' a* ⊨ *Ps' a*, ⦃λ*v*. *Q a*⦄
shows *spec.Parallel as Ps* ≤ ⦃⨅ *a*∈*as*. *P a*⦄, *A* ⊨ *spec.Parallel as Ps'*, ⦃λ*v*. ⨅ *a*∈*as*. *Q a*⦄
proof(*cases as* = {})
case *True* **then show** *?thesis*
by (*simp add*: *spec.Parallel.no-agents refinement.sort-of-refl*[**where** *G*=⊤] *refinement.top*)
next
case *False* **then show** *?thesis*
unfolding *refinement-def*
apply (*subst* (*1*) *spec.Parallel-def*)
apply (*simp only*: *spec.map-invmap.galois spec.invmap.inf*
spec.next-imp.invmap spec.invmap.post spec.invmap.pre)
apply (*subst* (*1*) *spec.Parallel-def*)
apply (*strengthen ord-to-strengthen*(*2*)[*OF spec.map-invmap.upper-lower-expansive*])
apply (*subst inf.assoc*)
apply (*subst spec.next-imp.infR*)
apply (*simp only*: *spec.next-imp.contains inf.bounded-iff inf-le1*)
apply (*strengthen ord-to-strengthen*[*OF assms*], *assumption*)

```
  apply (simp only: spec.invmap.refinement id-apply simp-thms)
  apply (rule order.trans[rotated, OF spec.Abadi-Merz-Theorem4[where
          I=as
      and As=λa. spec.pre (P a) ⊓ spec.toConcurrent a (refinement.spec.env-hyp P A as Ps' a)
      and Cs=λa. spec.toConcurrent a (Ps' a) ⊓ spec.post ⟨Q a⟩]])
  apply (simp only: inf.bounded-iff)
  apply (intro conjI)
   — the meat of refinement.spec.env-hyp
   apply (simp only: heyting)
   apply (rule INFI)
   apply (simp only: inf.bounded-iff
             flip: INF-inf-distrib)
   apply (intro conjI)
    apply (force simp: ac-simps spec.pre.INF
             intro: le-infI1 le-infI2)
   apply (simp add: refinement.spec.env-hyp-def ac-simps
            flip: spec.map-invmap.galois)
   apply (rule conjI)
    apply (simp add: spec.map-invmap.galois spec.invmap.pre le-infI2
             del: Inf-apply INF-apply; fail)
   apply (simp add: spec.map.mono le-infI2; fail)
  apply (simp add: spec.next-imp.contains heyting spec.post.INF flip: INF-inf-distrib; fail)
  apply (force simp: refinement-def)
  done
qed

setup ‹Sign.parent-path›
```

12.2 A relational assume/guarantee program logic for the *(sequential, 's, 'v) spec* lattice

Here we develop an assume/guarantee story based on abstracting processes (represented as safety properties) to binary relations.

Observations:

- this can be seen as a reconstruction of the algebraic account given by van Staden (2015) in our setting
- we show Heyting implication suffices for relations (see *ag.refinement*)
  - the processes' agent type is required to be *sequential*
- we use predicates and not relations for pre/post assertions
  - we can use the metalanguage to do some relational reasoning; see, for example, *ag.name-pre-state*
- *Id* is the smallest significant assume and guarantee relation here; processes can always stutter any state

```
setup ‹Sign.mandatory-path ag›

abbreviation (input) assm :: 's rel ⇒ (sequential, 's, 'v) spec where
  assm A ≡ spec.rel ({env} × A ∪ {self} × UNIV)

abbreviation (input) guar :: 's rel ⇒ (sequential, 's, 'v) spec where
  guar G ≡ spec.rel ({env} × UNIV ∪ {self} × G)

setup ‹Sign.parent-path›

definition ag :: 's pred ⇒ 's rel ⇒ 's rel ⇒ ('v ⇒ 's pred) ⇒ (sequential, 's, 'v) spec (‹{|-|}, -/ ⊢ -, {|-|}› [0,0,0,0]
100) where
  {|P|}, A ⊢ G, {|Q|} = spec.pre P ⊓ ag.assm A ⟶H ag.guar G ⊓ spec.post Q
```

setup ‹*Sign.mandatory-path spec.invmap*›

lemma *ag*: — Note *af* = *id*
fixes *sf* :: *'s* ⇒ *'t*
fixes *vf* :: *'v* ⇒ *'w*
fixes *A* :: *'t rel*
fixes *G* :: *'t rel*
fixes *P* :: *'t pred*
fixes *Q* :: *'w* ⇒ *'t pred*
shows *spec.invmap id sf vf* (⦃*P*⦄, *A* ⊢ *G*, ⦃*Q*⦄) = ⦃λ*s*. *P* (*sf s*)⦄, *inv-image* (*A*=) *sf* ⊢ *inv-image* (*G*=) *sf*, ⦃λ*v s*. *Q* (*vf v*) (*sf s*)⦄
proof −
have {*self*} × *UNIV* ∪ ({*env*} × *inv-image A sf* ∪ *UNIV* × *inv-image Id sf*) = {*self*} × *UNIV* ∪ {*env*} × *inv-image* (*A*=) *sf*
and {*env*} × *UNIV* ∪ ({*self*} × *inv-image G sf* ∪ *UNIV* × *inv-image Id sf*) = {*env*} × *UNIV* ∪ {*self*} × *inv-image* (*G*=) *sf*
by *auto*
then show *?thesis*
by (*simp add*: *ag-def spec.invmap.heyting spec.invmap.inf spec.invmap.rel spec.invmap.pre spec.invmap.post*
ac-simps map-prod-vimage-Times Sigma-Un-distrib2
flip: *inv-image-alt-def*)
qed

setup ‹*Sign.parent-path*›

setup ‹*Sign.mandatory-path ag*›

lemma *refinement*:
shows ⦃*P*⦄, *A* ⊢ *G*, ⦃*Q*⦄ = ⦃*P*⦄, *ag.assm A* ⊨ *ag.guar G*, ⦃*Q*⦄
proof −
have *constrains-heyting-ag*: *ag.assm A* ⟶+ *ag.guar G* = *ag.assm A* ⟶*H* *ag.guar G*
apply (*rule antisym*[*OF spec.next-imp.heyting-le*])
apply (*simp add*: *spec.next-imp.heyting heyting*)
apply (*subst inf.commute*)
apply (*rule spec.composition-principle-half*[**where** *a*1={*self*} **and** *a*2={*env*}];
force simp: *spec.idle-le spec.term.closed.rel*)
done
have *constrains-heyting-post*: *P* ⟶+ *spec.post Q* = *P* ⟶*H* *spec.post Q*
if *P* ∈ *spec.term.closed* -
for *P* :: (*sequential*, -, -) *spec*
apply (*rule antisym*[*OF spec.next-imp.heyting-le*])
apply (*clarsimp simp*: *spec.next-imp.heyting*)
apply (*metis spec.term.all.closed-conv*[*OF that*] *heyting.topL order-refl spec.term.all.post*
spec.term.all-none spec.term.heyting-noneL-allR-mono spec.term.lower-upper-lower(*2*))
done
show *?thesis*
by (*simp add*: *ag-def refinement-def*
spec.pre.next-imp-eq-heyting spec.idle-le
constrains-heyting-ag spec.next-imp.infR spec.term.closed.rel
constrains-heyting-post[*OF spec.term.closed.rel*] *heyting.infR heyting.curry-conv*)
qed

lemma *E*:
assumes *c* ≤ ⦃*P*⦄, *A* ⊢ *G*, ⦃*Q*⦄
obtains *c* ≤ *spec.pre P* ⊓ *ag.assm A* ⟶*H* *ag.guar G*
and *c* ≤ *spec.pre P* ⊓ *ag.assm A* ⟶*H* *spec.post Q*
using *assms* **by** (*simp add*: *ag-def heyting.infR*)

lemma *pre-post-cong*:
  assumes $P = P'$
  assumes $Q = Q'$
  shows $\{\!|P|\!\}, A \vdash G, \{\!|Q|\!\} = \{\!|P'|\!\}, A \vdash G, \{\!|Q'|\!\}$
using *assms* **by** *simp*

lemma *pre-bot*:
  shows $\{\!|\bot|\!\}, A \vdash G, \{\!|Q|\!\} = \top$
    and $\{\!|\langle\bot\rangle|\!\}, A \vdash G, \{\!|Q|\!\} = \top$
    and $\{\!|\langle False\rangle|\!\}, A \vdash G, \{\!|Q|\!\} = \top$
by (*simp-all add*: *ag-def heyting.botL*)

lemma *post-top*:
  shows $\{\!|P|\!\}, A \vdash UNIV, \{\!|\top|\!\} = \top$
    and $\{\!|P|\!\}, A \vdash UNIV, \{\!|\langle\top\rangle|\!\} = \top$
    and $\{\!|P|\!\}, A \vdash UNIV, \{\!|\lambda\text{- -}.\ True|\!\} = \top$
by (*simp-all add*: *ag-def spec.rel.upper-top flip*: *Sigma-Un-distrib1*)

lemma *mono*:
  assumes $P' \leq P$
  assumes $A' \leq A$
  assumes $G \leq G'$
  assumes $Q \leq Q'$
  shows $\{\!|P|\!\}, A \vdash G, \{\!|Q|\!\} \leq \{\!|P'|\!\}, A' \vdash G', \{\!|Q'|\!\}$
unfolding *ag-def*
apply (*strengthen ord-to-strengthen*(*1*)[*OF assms*(*1*)])
apply (*strengthen ord-to-strengthen*(*1*)[*OF assms*(*2*)])
apply (*strengthen ord-to-strengthen*(*1*)[*OF assms*(*3*)])
apply (*strengthen ord-to-strengthen*(*1*)[*OF assms*(*4*)])
apply *simp*
done

lemma *strengthen*[*strg*]:
  assumes *st-ord* $(\neg\ F)\ P\ P'$
  assumes *st-ord* $(\neg\ F)\ A\ A'$
  assumes *st-ord* $F\ G\ G'$
  assumes *st-ord* $F\ Q\ Q'$
  shows *st-ord* F $(\{\!|P|\!\}, A \vdash G, \{\!|Q|\!\})$ $(\{\!|P'|\!\}, A' \vdash G', \{\!|Q'|\!\})$
using *assms* **by** (*cases F*; *simp add*: *ag.mono*)

lemma *strengthen-pre*:
  assumes *st-ord* $(\neg\ F)\ P\ P'$
  shows *st-ord* F $(\{\!|P|\!\}, A \vdash G, \{\!|Q|\!\})$ $(\{\!|P'|\!\}, A \vdash G, \{\!|Q|\!\})$
using *assms* **by** (*cases F*; *simp add*: *ag.mono*)

lemmas *pre-ag* = *order.trans*[*OF - ag.mono*[*OF order.refl - - order.refl*], *of c*] **for** *c*
lemmas *pre-a* = *ag.pre-ag*[*OF - - order.refl*]
lemmas *pre-g* = *ag.pre-ag*[*OF - order.refl*]

lemma *pre*:
  assumes $c \leq \{\!|P|\!\}, A \vdash G, \{\!|Q|\!\}$
  assumes $\bigwedge s.\ P'\ s \Longrightarrow P\ s$
  assumes $A' \subseteq A$
  assumes $G \subseteq G'$
  assumes $\bigwedge v\ s.\ Q\ v\ s \Longrightarrow Q'\ v\ s$
  shows $c \leq \{\!|P'|\!\}, A' \vdash G', \{\!|Q'|\!\}$
using *assms*(*2*−) **by** (*blast intro*: *order.trans*[*OF assms*(*1*) *ag.mono*])

lemmas *pre-pre-post* = *ag.pre*[*OF - - order.refl order.refl*, *of c*] **for** *c*

lemma *pre-imp*:
  assumes $\bigwedge s.\ P\ s \Longrightarrow P'\ s$
  assumes $c \leq \{\!|P'|\!\}, A \vdash G, \{\!|Q|\!\}$
  shows $c \leq \{\!|P|\!\}, A \vdash G, \{\!|Q|\!\}$
using *assms ag.pre* **by** *blast*

lemmas *pre-pre* = *ag.pre-imp*[*rotated*]

lemma *post-imp*:
  assumes $\bigwedge v\ s\ .\ Q\ v\ s \Longrightarrow Q'\ v\ s$
  assumes $c \leq \{\!|P|\!\}, A \vdash G, \{\!|Q|\!\}$
  shows $c \leq \{\!|P|\!\}, A \vdash G, \{\!|Q'|\!\}$
using *assms ag.pre* **by** *blast*

lemmas *pre-post* = *ag.post-imp*[*rotated*]
lemmas *strengthen-post* = *ag.pre-post*

lemmas *reflcl-ag* = *spec.invmap.ag*[**where** *sf*=*id* **and** *vf*=*id*, *simplified spec.invmap.id*, *simplified*]

lemma
  shows *reflcl-a*: $\{\!|P|\!\}, A \vdash G, \{\!|Q|\!\} = \{\!|P|\!\}, A^{=} \vdash G, \{\!|Q|\!\}$
    and *reflcl-g*: $\{\!|P|\!\}, A \vdash G, \{\!|Q|\!\} = \{\!|P|\!\}, A \vdash G^{=}, \{\!|Q|\!\}$
by (*metis ag.reflcl-ag sup.left-idem sup-commute*)+

lemma *gen-asm-base*:
  assumes $P \Longrightarrow c \leq \{\!|P' \wedge P''|\!\}, A \vdash G, \{\!|Q|\!\}$
  shows $c \leq \{\!|P' \wedge \langle P\rangle \wedge P''|\!\}, A \vdash G, \{\!|Q|\!\}$
using *assms* **by** (*simp add*: *ag-def spec.pre.conj spec.pre.K heyting.botL*)

lemmas *gen-asm* =
  ag.gen-asm-base[**where** P'=$\langle True\rangle$ **and** P''=$\langle True\rangle$, *simplified*]
  ag.gen-asm-base[**where** P'=$\langle True\rangle$, *simplified*]
  ag.gen-asm-base[**where** P''=$\langle True\rangle$, *simplified*]
  ag.gen-asm-base

lemma *post-conj*:
  assumes $c \leq \{\!|P|\!\}, A \vdash G, \{\!|Q|\!\}$
  assumes $c \leq \{\!|P|\!\}, A \vdash G, \{\!|Q'|\!\}$
  shows $c \leq \{\!|P|\!\}, A \vdash G, \{\!|\lambda v.\ Q\ v \wedge Q'\ v|\!\}$
using *assms* **by** (*simp add*: *ag-def spec.post.conj heyting*)

lemma *pre-disj*:
  assumes $c \leq \{\!|P|\!\}, A \vdash G, \{\!|Q|\!\}$
  assumes $c \leq \{\!|P'|\!\}, A \vdash G, \{\!|Q|\!\}$
  shows $c \leq \{\!|P \vee P'|\!\}, A \vdash G, \{\!|Q|\!\}$
using *assms* **by** (*simp add*: *ag-def spec.pre.disj inf-sup-distrib heyting*)

lemma *drop-imp*:
  assumes $c \leq \{\!|P|\!\}, A \vdash G, \{\!|Q|\!\}$
  shows $c \leq \{\!|P|\!\}, A \vdash G, \{\!|\lambda v.\ Q'\ v \longrightarrow Q\ v|\!\}$
using *assms ag.strengthen-post* **by** *fastforce*

lemma *prop*:
  shows $c \leq \{\!|\langle P\rangle|\!\}, A \vdash UNIV, \{\!|\lambda v.\ \langle P\rangle|\!\}$
by (*simp add*: *ag.gen-asm*(*1*) *ag.post-top*(*3*))

lemma *name-pre-state*:
assumes $\bigwedge s.\ P\ s \Longrightarrow c \leq \{\!|(=)\ s|\!\},\ A \vdash G,\ \{\!|Q|\!\}$
shows $c \leq \{\!|P|\!\},\ A \vdash G,\ \{\!|Q|\!\}$
by (*metis assms ag.refinement refinement.name-pre-state spec.idle.rel-le*)

lemma *conj-lift*:
assumes $c \leq \{\!|P|\!\},\ A \vdash G,\ \{\!|Q|\!\}$
assumes $c \leq \{\!|P'|\!\},\ A \vdash G,\ \{\!|Q'|\!\}$
shows $c \leq \{\!|P \wedge P'|\!\},\ A \vdash G,\ \{\!|\lambda v.\ Q\ v \wedge Q'\ v|\!\}$
using *assms* **by** (*blast intro*: *ag.post-conj ag.pre*)

lemma *disj-lift*:
assumes $c \leq \{\!|P|\!\},\ A \vdash G,\ \{\!|Q|\!\}$
assumes $c \leq \{\!|P'|\!\},\ A \vdash G,\ \{\!|Q'|\!\}$
shows $c \leq \{\!|P \vee P'|\!\},\ A \vdash G,\ \{\!|\lambda v.\ Q\ v \vee Q'\ v|\!\}$
using *assms* **by** (*simp add*: *ag-def spec.post.disj spec.pre.disj heyting inf-sup-distrib le-supI1 le-supI2*)

lemma *all-lift*:
assumes $\bigwedge x.\ c \leq \{\!|P\ x|\!\},\ A \vdash G,\ \{\!|Q\ x|\!\}$
shows $c \leq \{\!|\forall x.\ P\ x|\!\},\ A \vdash G,\ \{\!|\lambda v.\ \forall x.\ Q\ x\ v|\!\}$
using *assms*
by (*auto simp*: *ag-def spec.pre.All spec.post.All le-Inf-iff heyting*
simp flip: *INF-inf-const1 INF-inf-const2*)

lemma *interference-le*:
shows *spec.rel* $(\{env\} \times UNIV) \leq \{\!|P|\!\},\ A \vdash G,\ \{\!|\top|\!\}$
and *spec.rel* $(\{env\} \times UNIV) \leq \{\!|P|\!\},\ A \vdash G,\ \{\!|\lambda\text{-}.\ \top|\!\}$
and *spec.rel* $(\{env\} \times UNIV) \leq \{\!|P|\!\},\ A \vdash G,\ \{\!|\lambda\text{-}\ \text{-}.\ True|\!\}$
by (*auto simp*: *ag-def heyting spec.term.all.rel intro*: *spec.rel.mono inf.coboundedI1*)

lemma *assm-heyting*:
fixes $Q :: 'v \Rightarrow 's\ pred$
shows $ag.assm\ r \longrightarrow_H \{\!|P|\!\},\ A \vdash G,\ \{\!|Q|\!\} = \{\!|P|\!\},\ A \cap r \vdash G,\ \{\!|Q|\!\}$
by (*simp add*: *ag-def ac-simps Int-Un-distrib Times-Int-Times flip*: *heyting.curry-conv spec.rel.inf*)

lemma *augment-a*: — instantiate A'
assumes $c \leq \{\!|P|\!\},\ A \vdash G,\ \{\!|Q|\!\}$
shows $c \leq \{\!|P|\!\},\ A \cap A' \vdash G,\ \{\!|Q|\!\}$
by (*blast intro*: *ag.pre-a*[*OF assms*])

lemma *augment-post*: — instantiate Q
assumes $c \leq \{\!|P|\!\},\ A \vdash G,\ \{\!|\lambda v.\ Q'\ v \wedge Q\ v|\!\}$
shows $c \leq \{\!|P|\!\},\ A \vdash G,\ \{\!|Q'|\!\}$
by (*blast intro*: *ag.pre-post*[*OF assms*])

lemma *augment-post-imp*: — instantiate Q
assumes $c \leq \{\!|P|\!\},\ A \vdash G,\ \{\!|\lambda v.\ (Q\ v \longrightarrow Q'\ v) \wedge Q\ v|\!\}$
shows $c \leq \{\!|P|\!\},\ A \vdash G,\ \{\!|Q'|\!\}$
by (*blast intro*: *ag.pre-post*[*OF assms*])

setup ‹*Sign.parent-path*›

setup ‹*Sign.mandatory-path spec.term.none*›

lemma *ag-le*:
shows *spec.term.none* $(\{\!|P|\!\},\ A \vdash G,\ \{\!|Q|\!\}) \leq \{\!|P|\!\},\ A \vdash G,\ \{\!|\bot|\!\}$
by (*simp add*: *ag.refinement spec.term.all.rel order.trans*[*OF spec.term.none.refinement-le*])

setup ‹*Sign.parent-path*›

setup ‹*Sign.mandatory-path ag.spec.term*›

lemmas *none-inteference* =
  order.trans[*OF spec.term.none.mono*,
        OF ag.interference-le(*1*) *ag.pre-post*[**where** $Q'{=}Q$ **for** Q, *OF spec.term.none.ag-le*, *simplified*]]

setup ‹*Sign.parent-path*›

setup ‹*Sign.mandatory-path ag.spec*›

lemma *bind*:
  assumes *g*: $\bigwedge v.\ g\ v \leq \{\!|Q'\ v|\!\}, A \vdash G, \{\!|Q|\!\}$
  assumes *f*: $f \leq \{\!|P|\!\}, A \vdash G, \{\!|Q'|\!\}$
  shows $f \gg\!= g \leq \{\!|P|\!\}, A \vdash G, \{\!|Q|\!\}$
apply (*subst ag.refinement*)
apply (*rule refinement.spec.bind*[**where** $f'{=}$*ag.guar G* **and** $g'{=}\langle$*ag.guar G*$\rangle$, *simplified spec.rel.wind-bind*])
 apply (*rule refinement.pre-a*[*OF g*[*unfolded ag.refinement*]])
 apply (*simp-all add*: *refinement.spec.bind.res.rel-le spec.term.all.rel f*[*unfolded ag.refinement*])
done

lemma *action*:
  fixes $F :: ('v \times sequential \times 's \times 's)\ set$
  assumes *Q*: $\bigwedge v\ a\ s\ s'.\ [\![P\ s; (v, a, s, s') \in F]\!] \Longrightarrow Q\ v\ s'$
  assumes *G*: $\bigwedge v\ s\ s'.\ [\![P\ s; (v, self, s, s') \in F; s \neq s']\!] \Longrightarrow (s, s') \in G$
  shows *spec.action F* $\leq \{\!|P|\!\}, A \vdash G, \{\!|Q|\!\}$
proof −
  from *G* **have** ∗: *spec.pre P* ⊓ *spec.action F* ≤ *spec.rel* ({*env*} × *UNIV* ∪ {*self*} × *G*)
    by − (*rule order.trans*[*OF spec.pre.inf-action-le*(*1*) *spec.action.rel-le*]; *auto*)
  show *?thesis*
    by (*fastforce intro*: *order.trans*[*OF - refinement.mono-stronger*[*OF order.refl - - order.refl*]]
                refinement.spec.action Q
              simp: *ag.refinement order.trans*[*OF* ∗] *spec.next-imp.closure.expansive spec.idle.rel-le*)
qed

lemma *return*:
  shows *spec.return v* ≤ $\{\!|Q\ v|\!\}, A \vdash G, \{\!|Q|\!\}$
by (*auto simp*: *spec.return-def intro*: *ag.spec.action*)

lemma *Parallel-assm*:
  shows *refinement.spec.env-hyp P* (*ag.assm A*) *as* (*ag.guar* ∘ *G*) *a* ≤ *ag.assm* ($A \cup \bigcup$ (*G* ‘ (*as* − {*a*})))
by (*simp add*: *refinement.spec.env-hyp-def spec.invmap.rel flip*: *spec.rel.upper-INF spec.rel.inf*)
   (*force intro*!: *le-infI2 order.trans*[*OF spec.map.rel-le*] *spec.rel.mono-reflcl*)

lemma *Parallel-guar*:
  shows *spec.Parallel as* (*ag.guar* ∘ *G*) = *ag.guar* ($\bigcup a{\in}as.\ G\ a$)
proof −
  have ∗: {*self*} × *Id* ∪ (*insert env* ((λ*x*. *self*) ‘ *as*) × *Id* ∪ *map-prod toSequential-fn id* ‘ (*insert env* (*proc* ‘ *as*)
× *UNIV* ∩ ($\bigcap x{\in}as$. {*proc x*} × *G x* ∪ (− {*proc x*}) × *UNIV*)))
        = {*env*} × *UNIV* ∪ ({*self*} × *Id* ∪ {*self*} × $\bigcup$ (*G* ‘ *as*))
    by (*rule antisym*, *force simp*: *toSequential-fn-def*, (*safe*; *force simp*: *map-prod-conv*))
  show *?thesis*
    apply (*simp add*: *spec.Parallel-def spec.invmap.rel*
          flip: *spec.rel.INF spec.rel.inf*)
    apply (*subst spec.map.rel*)
     apply (*clarsimp*; *blast*)

apply (*subst* (*1 2*) *spec.rel.reflcl*[**where** *A*={*self*}, *symmetric*])
apply (*clarsimp simp*: *ac-simps inf-sup-distrib image-Un image-image* ∗
map-prod-image-Times map-prod-vimage-Times Times-Int-Times)
done
qed

lemma *Parallel*:
fixes *A* :: *'s rel*
fixes *G* :: *'a* ⇒ *'s rel*
fixes *Q* :: *'a* ⇒ *'s* ⇒ *bool*
fixes *Ps* :: *'a* ⇒ (*sequential*, *'s*, *unit*) *spec*
assumes *proc-ag*: ⋀*a*. *a* ∈ *as* ⟹ *Ps a* ≤ ⦃*P a*⦄, *A* ∪ (⋃ *a'*∈*as*−{*a*}. *G a'*) ⊢ *G a*, ⦃λ*v*. *Q a*⦄
shows *spec.Parallel as Ps* ≤ ⦃⋂ *a*∈*as*. *P a*⦄, *A* ⊢ ⋃ *a*∈*as*. *G a*, ⦃λ*rv*. ⋂ *a*∈*as*. *Q a*⦄
unfolding *ag.refinement*
apply (*rule order.trans*[*OF - refinement.mono*[*OF order.refl - - order.refl*]])
apply (*rule refinement.spec.Parallel*[**where** *A*=*ag.assm A* **and** *Ps'*=*ag.guar* ∘ *G*])
apply (*rule order.trans*[*OF proc-ag*, *unfolded ag.refinement*], *assumption*)
apply (*rule refinement.mono*[*OF order.refl - - order.refl*])
apply (*force intro*: *ag.spec.Parallel-assm simp*: *ag.spec.Parallel-guar*)+
done

setup ‹*Sign.parent-path*›

12.2.1 Stability rules

setup ‹*Sign.mandatory-path spec*›

lemma *stable-pre-post*:
fixes *S* :: (*'a*, *'s*, *'v*) *spec*
assumes *stable* (*snd* ‘ *r*) *P*
assumes *spec.steps S* ⊆ *r*
shows *S* ≤ *spec.pre P* ⟶*H* *spec.post* ⟨*P*⟩
proof −
have *P* (*trace.final' s xs*)
if *P s*
and *trace.steps* (*trace.T s xs v*) ⊆ *r*
for *s* :: *'s* **and** *xs* :: (*'a* × *'s*) *list* **and** *v* :: *'v option*
using *that*
proof(*induct xs arbitrary*: *s*)
case (*Cons x xs*) **with** ‹*stable* (*snd* ‘ *r*) *P*› **show** *?case*
by (*cases x*; *cases snd x* = *s*;
force simp: *stable-def monotone-def dest*: *le-boolD*)
qed *simp*
from *this* ‹*spec.steps S* ⊆ *r*› **show** *?thesis*
by − (*rule spec.singleton-le-extI*;
auto dest: *order.trans*[**where** *b*=*S*]
simp: *spec.singleton.le-conv spec.singleton.heyting-le-conv trace.split-all spec.rel.galois*
split: *option.split*)
qed

lemma *pre-post-stable*:
fixes *P* :: *'s* ⇒ *bool*
assumes *stable* (*snd* ‘ *r*) *P*
shows *spec.rel r* ≤ *spec.pre P* ⟶*H* *spec.post* ⟨*P*⟩
by (*rule spec.stable-pre-post*[*OF assms spec.rel.lower-upper-contractive*])

setup ‹*Sign.parent-path*›

setup ‹*Sign.mandatory-path ag*›

lemma *stable-lift*:
  assumes *stable* $(A \cup G)$ P' — anything stable over $A \cup G$ is invariant
  shows $\{\!|P \wedge P'|\!\}, A \vdash G, \{\!|\lambda v.\ P' \longrightarrow Q\ v|\!\} \leq \{\!|P \wedge P'|\!\}, A \vdash G, \{\!|\lambda v.\ Q\ v \wedge P'|\!\}$
apply (*simp add*: *ag-def spec.pre.conj heyting heyting.detachment le-infI2 flip*: *spec.heyting.post*)
apply (*simp add*: *ac-simps Sigma-Un-distrib2 Int-Un-distrib Times-Int-Times flip*: *spec.rel.inf*)
apply (*rule order.subgoal*)
 apply (*rule order.trans*[*OF - spec.pre-post-stable*[**where** *r*=$\{env\} \times A \cup \{self\} \times G$, *simplified image-Un*, *simplified*, *OF assms*]])
 apply (*simp add*: *le-infI2*; *fail*)
apply (*simp add*: *ac-simps spec.post.conj*)
apply (*simp add*: *heyting.discharge le-infI1 flip*: *inf.assoc*)
done

lemma *stable-augment-base*:
  assumes $c \leq \{\!|P \wedge P'|\!\}, A \vdash G, \{\!|\lambda v.\ P' \longrightarrow Q\ v|\!\}$
  assumes *stable* $(A \cup G)$ P' — anything stable over $A \cup G$ is invariant
  shows $c \leq \{\!|P \wedge P'|\!\}, A \vdash G, \{\!|\lambda v.\ Q\ v \wedge P'|\!\}$
using *order.trans*[*OF - ag.stable-lift*] *assms* **by** *blast*

lemma *stable-augment*:
  assumes $c \leq \{\!|P'|\!\}, A \vdash G, \{\!|Q'|\!\}$
  assumes $\bigwedge v\ s.\ [\![P\ s;\ Q'\ v\ s]\!] \Longrightarrow Q\ v\ s$
  assumes *stable* $(A \cup G)$ P
  shows $c \leq \{\!|P' \wedge P|\!\}, A \vdash G, \{\!|Q|\!\}$
apply (*rule ag.strengthen-post*)
 apply (*rule ag.stable-augment-base*[**where** *Q*=*Q*, *OF - assms*(*3*)])
 apply (*auto intro*: *assms*(*2*) *ag.pre*[*OF assms*(*1*)])
done

lemma *stable-augment-post*:
  assumes $c \leq \{\!|P'|\!\}, A \vdash G, \{\!|Q'|\!\}$ — resolve before application
  assumes $\bigwedge v.$ *stable* $(A \cup G)$ $(Q'\ v \longrightarrow Q\ v)$
  shows $c \leq \{\!|(\forall v.\ Q'\ v \longrightarrow Q\ v) \wedge P'|\!\}, A \vdash G, \{\!|Q|\!\}$
apply (*rule ag.pre-pre-post*)
 apply (*rule ag.stable-augment-base*[**where** *P*=*P'* **and** *Q*=*Q'* **and** *P'*=$(\forall v.\ Q'\ v \longrightarrow Q\ v)$])
  apply (*rule ag.pre-pre-post*[*OF assms*(*1*)])
  using *assms*(*2*) **apply** (*fast intro*: *stable.allI*)+
done

lemma *stable-augment-frame*: — anything stable over $A \cup G$ is invariant
  assumes $c \leq \{\!|P|\!\}, A \vdash G, \{\!|Q|\!\}$
  assumes *stable* $(A \cup G)$ P'
  shows $c \leq \{\!|P \wedge P'|\!\}, A \vdash G, \{\!|\lambda v.\ Q\ v \wedge P'|\!\}$
using *assms* **by** (*blast intro*: *ag.stable-augment*[*OF assms*(*1*)])

setup ‹*Sign.parent-path*›

setup ‹*Sign.mandatory-path ag.spec*›

lemma *stable-interference*:
  assumes *stable* $(A \cap r)$ P
  shows *spec.rel* $(\{env\} \times r) \leq \{\!|P|\!\}, A \vdash G, \{\!|\langle P \rangle|\!\}$
using *assms*
by (*auto simp*: *ag-def ac-simps heyting Int-Un-distrib Times-Int-Times*
    simp flip: *spec.rel.inf*
        intro: *le-infI2 spec.rel.mono spec.pre-post-stable*[*simplified heyting ac-simps*])

setup ‹*Sign.parent-path*›

setup ‹*Sign.mandatory-path spec*›

setup ‹*Sign.mandatory-path cam*›

lemma *closed-ag*:
  shows ⦃*P*⦄, *A* ⊢ *G*, ⦃*Q*⦄ ∈ *spec.cam.closed* ({*env*} × *r*)
unfolding *ag-def heyting.infR*
by (*blast intro*: *subsetD*[*OF spec.cam.closed.antimono*, *rotated*])

setup ‹*Sign.parent-path*›

setup ‹*Sign.mandatory-path interference*›

lemma *cl-ag-le*:
  assumes *P*: *stable* (*A* ∩ *r*) *P*
  assumes *Q*: ⋀*v*. *stable* (*A* ∩ *r*) (*Q v*)
  shows *spec.interference.cl* ({*env*} × *r*) (⦃*P*⦄, *A* ⊢ *G*, ⦃*Q*⦄) ≤ ⦃*P*⦄, *A* ⊢ *G*, ⦃*Q*⦄
unfolding *spec.interference.cl-def*
by (*rule ag.spec.bind ag.spec.return ag.spec.stable-interference spec.cam.least*[*OF - spec.cam.closed-ag*] *assms order.refl*)+

lemma *closed-ag*:
  assumes *P*: *stable* (*A* ∩ *r*) *P*
  assumes *Q*: ⋀*v*. *stable* (*A* ∩ *r*) (*Q v*)
  shows ⦃*P*⦄, *A* ⊢ *G*, ⦃*Q*⦄ ∈ *spec.interference.closed* ({*env*} × *r*)
by (*rule spec.interference.closed-clI*[*OF spec.interference.cl-ag-le*[*OF assms*]])

setup ‹*Sign.parent-path*›

setup ‹*Sign.parent-path*›

13 A programming language

The (*'a*, *'s*, *'v*) *spec* lattice of §8.2 is adequate for logic but is deficient as a programming language. In particular we wish to interpret the parallel composition as intersection (§9.5) which requires processes to contain enough interference opportunities. Similarly we want the customary "laws of programming" (Hoare, Hayes, He, Morgan, Roscoe, Sanders, Sørensen, Spivey, and Sufrin 1987a) to hold without side conditions.

These points are discussed at some length by Zwiers (1989, §3.2) and also Foster, Baxter, Cavalcanti, Woodcock, and Zeyda (2020, Lemma 6.7).

Our (*'v*, *'s*) *prog* lattice (§13.1) therefore handles the common case of the familiar constructs for sequential programming, and we lean on our (*'a*, *'s*, *'v*) *spec* lattice for other constructions such as interleaving parallel composition (§9.5) and local state (§15). It allows arbitrary interference by the environment before and after every program action.

13.1 The *('s, 'v) prog* lattice

According to Müller-Olm (1997, §2.1), (*'s*, *'v*) *prog* is a *sub-lattice* of (*'a*, *'s*, *'v*) *spec* as the corresponding (⊓) and (⊔) operations coincide. However it is not a *complete* sub-lattice as *Sup* in (*'s*, *'v*) *prog* needs to account for the higher bottom of that lattice.

typedef (*'s*, *'v*) *prog* = *spec.interference.closed* ({*env*} × *UNIV*) :: (*sequential*, *'s*, *'v*) *spec set*
morphisms *p2s Abs-t*
by *blast*

hide-const (**open**) *p2s*

setup-lifting *type-definition-prog*

instantiation *prog* :: (*type*, *type*) *complete-distrib-lattice*
begin

lift-definition *bot-prog* :: ('*s*, '*v*) *prog* **is** *spec.interference.cl* ({*env*} × *UNIV*) ⊥ **..**
lift-definition *top-prog* :: ('*s*, '*v*) *prog* **is** ⊤ **..**
lift-definition *sup-prog* :: ('*s*, '*v*) *prog* ⇒ ('*s*, '*v*) *prog* ⇒ ('*s*, '*v*) *prog* **is** *sup* **..**
lift-definition *inf-prog* :: ('*s*, '*v*) *prog* ⇒ ('*s*, '*v*) *prog* ⇒ ('*s*, '*v*) *prog* **is** *inf* **..**
lift-definition *less-eq-prog* :: ('*s*, '*v*) *prog* ⇒ ('*s*, '*v*) *prog* ⇒ *bool* **is** *less-eq* **.**
lift-definition *less-prog* :: ('*s*, '*v*) *prog* ⇒ ('*s*, '*v*) *prog* ⇒ *bool* **is** *less* **.**
lift-definition *Inf-prog* :: ('*s*, '*v*) *prog set* ⇒ ('*s*, '*v*) *prog* **is** *Inf* **..**
lift-definition *Sup-prog* :: ('*s*, '*v*) *prog set* ⇒ ('*s*, '*v*) *prog* **is** λ*X*. *Sup X* ⊔ *spec.interference.cl* ({*env*} × *UNIV*) ⊥ **..**

instance
by *standard* (*transfer*; *auto simp*: *Inf-lower InfI SupI le-supI1 spec.interference.least*)+

end

13.2 Morphisms to and from the *(sequential, 's, 'v) spec* lattice

We can readily convert a ('*s*, '*v*) *prog* into a ('*a agent*, '*s*, '*v*) *spec*. More interestingly, on ('*s*, '*v*) *prog* we have a Galois connection that embeds specifications into programs. (This connection is termed a *Galois insertion* by Melton et al. (1985) as we also have *prog.s2p.p2s*; Cousot says "Galois retraction".)
See also §13.4.2 and §13.5.1.

setup ‹*Sign.mandatory-path spec.interference.closed*›

lemmas *p2s*[*iff*] = *prog.p2s*

setup ‹*Sign.parent-path*›

setup ‹*Sign.mandatory-path spec.interference.cl*›

lemmas *p2s* = *spec.interference.closed-conv*[*OF spec.interference.closed.p2s*, *symmetric*, *of P* **for** *P*]

setup ‹*Sign.parent-path*›

setup ‹*Sign.mandatory-path spec.idle*›

lemmas *p2s-le*[*spec.idle-le*]
  = *spec.interference.le-closedE*[*OF spec.idle.interference.cl-le spec.interference.closed.p2s*, *of P* **for** *P*]
lemmas *p2s-minimal*[*iff*] = *order.trans*[*OF spec.idle.minimal-le spec.idle.p2s-le*]

setup ‹*Sign.parent-path*›

setup ‹*Sign.mandatory-path prog*›

lemma *p2s-leI*:
  assumes *prog.p2s c* ≤ *prog.p2s d*
  shows *c* ≤ *d*
by (*simp add*: *assms less-eq-prog.rep-eq*)

setup ‹*Sign.mandatory-path p2s*›

named-theorems *simps ‹simp rules for* ***const****‹p2s››*

lemmas *bot = bot-prog.rep-eq*
lemmas *top = top-prog.rep-eq*
lemmas *inf = inf-prog.rep-eq*
lemmas *sup = sup-prog.rep-eq*
lemmas *Inf = Inf-prog.rep-eq*
lemmas *Sup = Sup-prog.rep-eq*

lemma *Sup-not-empty*:
  assumes $X \neq \{\}$
  shows *prog.p2s* $(\bigsqcup X) = \bigsqcup$ *(prog.p2s ‘ X)*
using *assms* **by** *transfer* (*simp add*: *sup.absorb1 less-eq-Sup spec.interference.least*)

lemma *SUP-not-empty*:
  assumes $X \neq \{\}$
  shows *prog.p2s* $(\bigsqcup x{\in}X.\ f\ x) = (\bigsqcup x{\in}X.$ *prog.p2s* $(f\ x))$
by (*simp add*: *assms prog.p2s.Sup-not-empty*[**where** *X=f ‘ X*, *simplified image-image*])

lemma *monotone*:
  shows *mono prog.p2s*
by (*rule monoI*) (*transfer*; *simp*)

lemmas *strengthen*[*strg*] = *st-monotone*[*OF prog.p2s.monotone*]
lemmas *mono = monotoneD*[*OF prog.p2s.monotone*]
lemmas *mono2mono*[*cont-intro*, *partial-function-mono*] = *monotone2monotone*[*OF prog.p2s.monotone*, *simplified*, *of orda P* **for** *orda P*]

lemma *mcont*: — Morally *galois.complete-lattice.mcont-lower*
  shows *mcont Sup* $(\leq)$ *Sup* $(\leq)$ *prog.p2s*
by (*simp add*: *contI mcontI prog.p2s.Sup-not-empty*)

lemmas *mcont2mcont*[*cont-intro*] = *mcont2mcont*[*OF prog.p2s.mcont*, *of luba orda P* **for** *luba orda P*]

lemmas *Let-distrib = Let-distrib*[**where** *f=prog.p2s*]

lemmas [*prog.p2s.simps*] =
  prog.p2s.bot
  prog.p2s.top
  prog.p2s.inf
  prog.p2s.sup
  prog.p2s.Inf
  prog.p2s.Sup-not-empty
  spec.interference.cl.p2s
  prog.p2s.Let-distrib

lemma *interference-wind-bind*:
  shows *spec.rel* $(\{env\} \times UNIV) \gg\!=$ $(\lambda$-::*unit. prog.p2s P*$) =$ *prog.p2s P*
by (*subst* (*1 2*) *spec.interference.closed-conv*[*OF prog.p2s*])
   (*simp add*: *spec.interference.cl-def spec.rel.wind-bind flip*: *spec.bind.bind*)

setup *‹Sign.parent-path›*

definition *s2p* :: (*sequential*, *'s*, *'v*) *spec* ⇒ (*'s*, *'v*) *prog* **where** — Morally the upper of a Galois connection
  s2p P $= \bigsqcup\{c.$ *prog.p2s c* $\leq P\}$

setup *‹Sign.mandatory-path s2p›*

lemma *bottom*:
  shows *prog.s2p* $\bot = \bot$
by (*simp add*: *prog.s2p-def bot.extremum-uniqueI less-eq-prog.rep-eq*)

lemma *top*:
  shows *prog.s2p* $\top = \top$
by (*simp add*: *prog.s2p-def*)

lemma *monotone*:
  shows *mono prog.s2p*
by (*fastforce simp*: *prog.s2p-def intro*: *monotoneI Sup-subset-mono*)

lemmas *strengthen*[*strg*] = *st-monotone*[*OF prog.s2p.monotone*]
lemmas *mono* = *monotoneD*[*OF prog.s2p.monotone*]
lemmas *mono2mono*[*cont-intro*, *partial-function-mono*] = *monotone2monotone*[*OF prog.s2p.monotone*, *simplified*]

lemma *p2s*:
  shows *prog.s2p* (*prog.p2s P*) = *P*
by (*auto simp*: *prog.s2p-def simp flip*: *less-eq-prog.rep-eq intro*: *Sup-eqI*)

lemma *Sup-le*:
  shows $\bigsqcup$(*prog.s2p* ‘ *X*) $\leq$ *prog.s2p* ($\bigsqcup$ *X*)
by (*simp add*: *prog.s2p-def Collect-mono SUPE Sup-subset-mono Sup-upper2*)

lemma *sup-le*:
  shows *prog.s2p x* $\sqcup$ *prog.s2p y* $\leq$ *prog.s2p* (*x* $\sqcup$ *y*)
using *prog.s2p.Sup-le*[**where** *X*={*x*, *y*}] **by** *simp*

lemma *Inf*:
  shows *prog.s2p* ($\bigsqcap$ *X*) = $\bigsqcap$(*prog.s2p* ‘ *X*) (**is** *?lhs* = *?rhs*)
proof(*rule antisym*)
  show *?lhs* $\leq$ *?rhs*
    by (*simp add*: *prog.s2p-def SupI Sup-le-iff le-Inf-iff*)
  show *?rhs* $\leq$ *?lhs*
    by (*fastforce simp*: *prog.s2p-def prog.p2s.mono*
                  Inf-Sup[**where** *A*=(λx. {*c*. *prog.p2s c* $\leq$ *x*}) ‘ *X*, *simplified image-image*]
                  le-Inf-iff INF-lower2
                elim: *order.trans*[*rotated*]
               intro: *Sup-subset-mono*)
qed

lemma *inf*:
  shows *prog.s2p* (*x* $\sqcap$ *y*) = *prog.s2p x* $\sqcap$ *prog.s2p y*
using *prog.s2p.Inf*[**where** *X*={*x*, *y*}] **by** *simp*

setup ‹*Sign.parent-path*›

setup ‹*Sign.mandatory-path p2s-s2p*›

lemma *galois*: — the Galois connection
  shows *prog.p2s c* $\leq$ *S*
    $\longleftrightarrow$ *c* $\leq$ *prog.s2p S* $\wedge$ *spec.term.none* (*spec.rel* ({*env*} $\times$ *UNIV*) :: (-, -, *unit*) *spec*) $\leq$ *S* (**is** *?lhs* $\longleftrightarrow$ *?rhs*)
proof(*rule iffI*)
  show *?lhs* $\Longrightarrow$ *?rhs*
    by (*metis order.trans prog.s2p.mono prog.s2p.p2s spec.interference.closed.p2s*
          spec.term.none.interference.closed.rel-le)

show *?rhs* ⟹ *?lhs*
unfolding *prog.s2p-def* **by** *transfer* (*force simp*: *spec.interference.cl.bot elim*: *order.trans*)
qed

lemma *le*:
shows *prog.p2s* (*prog.s2p S*) ≤ *spec.interference.cl* ({*env*} × *UNIV*) *S*
by (*metis bot-prog.rep-eq prog.p2s-s2p.galois prog.s2p.mono spec.interference.cl-bot-least*
spec.interference.expansive)

lemma *insertion*:
fixes *S* :: (*sequential*, *'s*, *'v*) *spec*
assumes *S* ∈ *spec.interference.closed* ({*env*} × *UNIV*)
shows *prog.p2s* (*prog.s2p S*) = *S*
by (*metis assms prog.p2s-cases prog.s2p.p2s*)

setup ‹*Sign.parent-path*›

setup ‹*Sign.parent-path*›

13.3 Programming language constructs

We lift the combinators directly from the (*'a*, *'s*, *'v*) *spec* lattice (§8), but need to interference-close primitive actions. Control flow is expressed via HOL's *if−then−else* construct and other case combinators where the scrutinee is a pure value. This means that the atomicity of a process is completely determined by occurrences of *prog.action*.

setup ‹*Sign.mandatory-path prog*›

lift-definition *bind* :: (*'s*, *'v*) *prog* ⇒ (*'v* ⇒ (*'s*, *'w*) *prog*) ⇒ (*'s*, *'w*) *prog* **is**
spec.bind **..**

adhoc-overloading
Monad-Syntax.bind ⇌ *prog.bind*

lift-definition *action* :: (*'v* × *'s* × *'s*) *set* ⇒ (*'s*, *'v*) *prog* **is**
λ*F*. *spec.interference.cl* ({*env*} × *UNIV*) (*spec.action* (*map-prod id* (*Pair self*) ‘ *F*)) **..**

abbreviation (*input*) *det-action* :: (*'s* ⇒ (*'v* × *'s*)) ⇒ (*'s*, *'v*) *prog* **where**
det-action f ≡ *prog.action* {(*v*, *s*, *s'*). (*v*, *s'*) = *f s*}

definition *return* :: *'v* ⇒ (*'s*, *'v*) *prog* **where**
return v = *prog.action* ({*v*} × *Id*)

definition *guard* :: *'s pred* ⇒ (*'s*, *unit*) *prog* **where**
guard g ≡ *prog.action* ({()} × *Diag g*)

abbreviation (*input*) *read* :: (*'s* ⇒ *'v*) ⇒ (*'s*, *'v*) *prog* **where**
read F ≡ *prog.action* {(*F s*, *s*, *s*) |*s*. *True*}

abbreviation (*input*) *write* :: (*'s* ⇒ *'s*) ⇒ (*'s*, *unit*) *prog* **where**
write F ≡ *prog.action* {((), *s*, *F s*) |*s*. *True*}

lift-definition *Parallel* :: *'a set* ⇒ (*'a* ⇒ (*'s*, *unit*) *prog*) ⇒ (*'s*, *unit*) *prog* **is** *spec.Parallel*
by (*rule spec.interference.closed.Parallel*)

lift-definition *parallel* :: (*'s*, *unit*) *prog* ⇒ (*'s*, *unit*) *prog* ⇒ (*'s*, *unit*) *prog* **is** *spec.parallel*
by (*simp add*: *spec.parallel-def spec.interference.closed.Parallel*)

lift-definition *vmap* :: (*'v* ⇒ *'w*) ⇒ (*'s*, *'v*) *prog* ⇒ (*'s*, *'w*) *prog* **is** *spec.vmap*

by (*rule subsetD*[*OF spec.interference.closed.antimono spec.interference.closed.map-sf-id*, *rotated*])
auto

adhoc-overloading
Parallel ⇌ *prog.Parallel*
adhoc-overloading
parallel ⇌ *prog.parallel*

lemma *return-alt-def*:
shows *prog.return v* = *prog.read* ⟨*v*⟩
by (*auto simp*: *prog.return-def intro*: *arg-cong*[**where** *f*=*prog.action*])

lemma *parallel-alt-def*:
shows *prog.parallel P Q* = *prog.Parallel UNIV* (λ*a*::*bool*. *if a then P else Q*)
by *transfer* (*simp add*: *spec.parallel-def*)

lift-definition *rel* :: *'s rel* ⇒ (*'s*, *'v*) *prog* **is** λ*r*. *spec.rel* ({*env*} × *UNIV* ∪ {*self*} × *r*)
by (*simp add*: *spec.interference.closed.rel*)

lift-definition *steps* :: (*'s*, *'v*) *prog* ⇒ *'s rel* **is** λ*P*. *spec.steps P* `` {*self*} **.**

lift-definition *invmap* :: (*'s* ⇒ *'t*) ⇒ (*'v* ⇒ *'w*) ⇒ (*'t*, *'w*) *prog* ⇒ (*'s*, *'v*) *prog* **is**
spec.invmap id
by (*rule subsetD*[*OF spec.interference.closed.antimono spec.interference.closed.invmap*, *rotated*])
auto

abbreviation *sinvmap* ::(*'s* ⇒ *'t*) ⇒ (*'t*, *'v*) *prog* ⇒ (*'s*, *'v*) *prog* **where**
sinvmap sf ≡ *prog.invmap sf id*
abbreviation *vinvmap* ::(*'v* ⇒ *'w*) ⇒ (*'s*, *'w*) *prog* ⇒ (*'s*, *'v*) *prog* **where**
vinvmap vf ≡ *prog.invmap id vf*

declare *prog.bind-def*[*code del*]
declare *prog.action-def*[*code del*]
declare *prog.return-def*[*code del*]
declare *prog.Parallel-def*[*code del*]
declare *prog.parallel-def*[*code del*]
declare *prog.vmap-def*[*code del*]
declare *prog.rel-def*[*code del*]
declare *prog.steps-def*[*code del*]
declare *prog.invmap-def*[*code del*]

13.3.1 Laws of programming

setup ‹*Sign.mandatory-path p2s*›

lemma *bind*[*prog.p2s.simps*]:
shows *prog.p2s* (*f* ⤜ *g*) = *prog.p2s f* ⤜ (λ*x*. *prog.p2s* (*g x*))
by *transfer simp*

lemmas *action* = *prog.action.rep-eq*

lemma *return*:
shows *prog.p2s* (*prog.return v*) = *spec.interference.cl* ({*env*} × *UNIV*) (*spec.return v*)
by (*simp add*: *prog.return-def prog.p2s.action map-prod-image-Times Pair-image*
flip: *spec.return-alt-def*)

lemma *guard*:
shows *prog.p2s* (*prog.guard g*) = *spec.interference.cl* ({*env*} × *UNIV*) (*spec.guard g*)

by (*simp add*: *prog.guard-def prog.p2s.action map-prod-image-Times Pair-image*
flip: *spec.guard.alt-def*[**where** *A*={*self*}])

lemmas *Parallel*[*prog.p2s.simps*] = *prog.Parallel.rep-eq*[*simplified*, *of as Ps* **for** *as Ps*, *unfolded comp-def*]
lemmas *parallel*[*prog.p2s.simps*] = *prog.parallel.rep-eq*
lemmas *invmap*[*prog.p2s.simps*] = *prog.invmap.rep-eq*
lemmas *rel*[*prog.p2s.simps*] = *prog.rel.rep-eq*

setup ‹*Sign.parent-path*›

setup ‹*Sign.mandatory-path return*›

lemma *transfer*[*transfer-rule*]:
shows *rel-fun* (=) *cr-prog* (λ*v*. *spec.interference.cl* ({*env*} × *UNIV*) (*spec.return v*)) *prog.return*
by (*simp add*: *rel-funI cr-prog-def prog.p2s.return*)

lemma *cong*:
fixes *F* :: ('*v* × '*s* × '*s*) *set*
assumes ⋀*v s s'*. (*v*, *s*, *s'*) ∈ *F* ⟹ *s'* = *s*
assumes ⋀*v s s' s''*. *v* ∈ *fst* ‘ *F* ⟹ (*v*, *s*, *s*) ∈ *F*
shows *prog.action F* = (⨆(*v*, *s*, *s'*)∈*F*. *prog.return v*)
using *assms*
by *transfer*
(*subst spec.return.cong*;
fastforce simp: *spec.interference.cl.action spec.interference.cl.return*
spec.interference.cl.Sup spec.interference.cl.sup spec.interference.cl.idle
spec.interference.cl.bot image-image split-def
intro: *map-prod-imageI*[**where** *f*=*id*, *simplified*])

lemma *rel-le*:
shows *prog.return v* ≤ *prog.rel r*
by *transfer* (*simp add*: *spec.interference.least spec.interference.closed.rel spec.return.rel-le*)

setup ‹*Sign.parent-path*›

setup ‹*Sign.mandatory-path action*›

lemma *empty*:
shows *prog.action* {} = ⊥
by *transfer*
(*simp add*: *spec.action.empty spec.interference.cl.bot spec.interference.cl.idle*
flip: *bot-fun-def spec.bind.botR*)

lemma *monotone*:
shows *mono* (*prog.action* :: - ⇒ ('*s*, '*v*) *prog*)
proof(*transfer*, *rule monotoneI*)
show *spec.interference.cl* ({*env*} × *UNIV*) (*spec.action* (*map-prod id* (*Pair self*) ‘ *F*))
≤ *spec.interference.cl* ({*env*} × *UNIV*) (*spec.action* (*map-prod id* (*Pair self*) ‘ *F'*))
if *F* ⊆ *F'* **for** *F F'* :: ('*v* × '*s* × '*s*) *set*
by (*strengthen ord-to-strengthen*(*1*)[*OF* ‹*F* ⊆ *F'*›]) *simp*
qed

lemmas *strengthen*[*strg*] = *st-monotone*[*OF prog.action.monotone*]
lemmas *mono* = *monotoneD*[*OF prog.action.monotone*]
lemmas *mono2mono*[*cont-intro*, *partial-function-mono*] = *monotone2monotone*[*OF prog.action.monotone*, *simplified*]

lemma *Sup*:

shows *prog.action* ($\bigsqcup Fs$) = ($\bigsqcup F{\in}Fs.$ *prog.action* F)
by *transfer*
(*simp add*: *spec.interference.cl.bot spec.interference.cl.Sup spec.interference.cl.sup*
spec.interference.cl.idle spec.interference.cl.action spec.action.Sup image-Union image-image
flip: *bot-fun-def spec.bind.botR*)

lemmas *sup* = *prog.action.Sup*[**where** *Fs*={*F*, *G*} **for** *F G*, *simplified*]

lemma *Inf-le*:
shows *prog.action* ($\bigsqcap Fs$) ≤ ($\bigsqcap F{\in}Fs.$ *prog.action* F)
apply *transfer*
apply (*strengthen ord-to-strengthen*(*1*)[*OF image-Inter-subseteq*])
apply (*strengthen ord-to-strengthen*(*1*)[*OF spec.action.Inf-le*])
apply (*strengthen ord-to-strengthen*(*1*)[*OF spec.interference.cl-Inf-le*])
apply (*blast intro*: *Inf-mono*)
done

lemma *inf-le*:
shows *prog.action* ($F \sqcap G$) ≤ *prog.action* $F \sqcap$ *prog.action* G
using *prog.action.Inf-le*[**where** *Fs*={*F*, *G*}] **by** *simp*

lemma *sinvmap-le*: — a strict refinement
shows *prog.p2s* (*prog.action* (*map-prod id* (*map-prod sf sf*) −‘ F))
≤ *spec.sinvmap sf* (*prog.p2s* (*prog.action* F))
by (*force intro*: *order.trans*[*OF* - *spec.interference.cl.mono*[*OF order.refl spec.action.invmap-le*]]
spec.interference.cl.mono spec.action.mono
simp: *prog.p2s.action spec.invmap.interference.cl*)

lemma *return-const*:
fixes F :: *'s rel*
fixes V :: *'v set*
fixes W :: *'w set*
assumes $V \neq \{\}$
assumes $W \neq \{\}$
shows *prog.action* ($V \times F$) = *prog.action* ($W \times F$) ≫ ($\bigsqcup v{\in}V.$ *prog.return* v)
using *assms*(*1*)
by (*simp add*: *prog.p2s.simps prog.p2s.action prog.p2s.return image-image*
map-prod-image-Times spec.action.return-const[*OF assms*]
spec.bind.SUPR-not-empty spec.interference.cl.bind-return
spec.interference.cl.return spec.interference.cl-Sup-not-empty
spec.interference.closed.bind-relR
flip: *prog.p2s-inject*)

lemma *rel-le*:
assumes $\bigwedge v\ s\ s'.\ (v, s, s') \in F \Longrightarrow (s, s') \in r \vee s = s'$
shows *prog.action* $F \leq$ *prog.rel* r
by (*auto intro*: *order.trans*[*OF spec.interference.cl.mono*[*OF order.refl*
spec.action.rel-le[**where** *r*={*self*} × *r* ∪ {*env*} × *UNIV*]]]
dest: *assms*
simp: *less-eq-prog.rep-eq prog.p2s.simps prog.p2s.action spec.interference.cl.rel ac-simps*)

lemma *invmap-le*:
shows *prog.action* (*map-prod vf* (*map-prod sf sf*) −‘ F) ≤ *prog.invmap sf vf* (*prog.action* F)
by *transfer*
(*force simp*: *spec.invmap.interference.cl*
intro: *spec.interference.cl.mono*
spec.action.mono order.trans[*OF* - *spec.interference.cl.mono*[*OF order.refl spec.action.invmap-le*]])

lemma *action-le*:
  shows *spec.action* (*map-prod id* (*Pair self*) ‘ *F*) ≤ *prog.p2s* (*prog.action F*)
by (*simp add*: *prog.p2s.action spec.interference.expansive*)

setup ‹*Sign.parent-path*›

setup ‹*Sign.mandatory-path bind*›

lemmas *if-distrL* = *if-distrib*[**where** *f*=λ*x*. *x* ⤜ *g* **for** *g* :: - ⇒ (-, -) *prog*]

lemma *mono*:
  assumes *f* ≤ *f*′
  assumes ⋀*x*. *g x* ≤ *g*′ *x*
  shows *prog.bind f g* ≤ *prog.bind f*′ *g*′
using *assms* **by** *transfer* (*simp add*: *spec.bind.mono*)

lemma *strengthen*[*strg*]:
  assumes *st-ord F f f*′
  assumes ⋀*x*. *st-ord F* (*g x*) (*g*′ *x*)
  shows *st-ord F* (*prog.bind f g*) (*prog.bind f*′ *g*′)
using *assms* **by** (*cases F*; *clarsimp intro*!: *prog.bind.mono*)

lemma *mono2mono*[*cont-intro*, *partial-function-mono*]:
  assumes *monotone orda* (≤) *f*
  assumes ⋀*x*. *monotone orda* (≤) (λ*y*. *g y x*)
  shows *monotone orda* (≤) (λ*x*. *prog.bind* (*f x*) (*g x*))
using *assms* **by** *transfer simp*

The monad laws hold unconditionally in the (′*s*, ′*v*) *prog* lattice.

lemma *bind*:
  shows *f* ⤜ *g* ⤜ *h* = *prog.bind f* (λ*x*. *g x* ⤜ *h*)
by *transfer* (*simp add*: *spec.bind.bind*)

lemma *return*:
  shows *returnL*: (⤜) (*prog.return v*) = (λ*g* :: ′*v* ⇒ (′*s*, ′*w*) *prog*. *g v*) (**is** *?thesis1*)
    and *returnR*: *f* ⤜ *prog.return* = *f* (**is** *?thesis2*)
proof −
  have *prog.return v* ⤜ *g* = *g v* **for** *g* :: ′*v* ⇒ (′*s*, ′*w*) *prog*
    by *transfer*
      (*simp add*: *map-prod-image-Times Pair-image spec.action.read-agents*
          spec.interference.cl.return spec.bind.bind
          spec.bind.returnL spec.idle-le prog.p2s-induct spec.interference.closed.bind-relL
        flip: *spec.return-def*)
  then show *?thesis1*
    by (*simp add*: *fun-eq-iff*)
  show *?thesis2*
    by *transfer*
      (*simp add*: *map-prod-image-Times Pair-image spec.action.read-agents*
        flip: *spec.interference.cl.bindL spec.return-def spec.interference.closed-conv*)
qed

lemma *botL*:
  shows *prog.bind* ⊥ = ⊥
by (*simp add*: *fun-eq-iff prog.p2s.simps spec.interference.cl.bot*
    flip: *prog.p2s-inject*)

lemma *botR-le*:
  shows *prog.bind f* ⟨⊥⟩ ≤ *f* (**is** *?thesis1*)

and *prog.bind f* $\bot \le f$ (**is** *?thesis2*)
proof −
show *?thesis1*
by (*metis bot.extremum dual-order.refl prog.bind.mono prog.bind.returnR*)
then show *?thesis2*
by (*simp add*: *bot-fun-def*)
qed

lemma
fixes f :: (-, -) *prog*
fixes f_1 :: (-, -) *prog*
shows *supL*: $(f_1 \sqcup f_2) \gg\!= g = (f_1 \gg\!= g) \sqcup (f_2 \gg\!= g)$
and *supR*: $f \gg\!= (\lambda x.\ g_1\ x \sqcup g_2\ x) = (f \gg\!= g_1) \sqcup (f \gg\!= g_2)$
by (*transfer*; *blast intro*: *spec.bind.supL spec.bind.supR*)+

lemma *SUPL*:
fixes X :: - *set*
fixes f :: - ⇒ (-, -) *prog*
shows $(\bigsqcup x{\in}X.\ f\ x) \gg\!= g = (\bigsqcup x{\in}X.\ f\ x \gg\!= g)$
by *transfer*
(*simp add*: *spec.interference.cl.bot spec.bind.supL spec.bind.bind spec.bind.SUPL*
flip: *spec.bind.botR bot-fun-def*)

lemma *SUPR*:
fixes X :: - *set*
fixes f :: (-, -) *prog*
shows $f \gg\!= (\lambda v.\ \bigsqcup x{\in}X.\ g\ x\ v) = (\bigsqcup x{\in}X.\ f \gg\!= g\ x) \sqcup (f \gg\!= \bot)$
unfolding *bot-fun-def*
by *transfer*
(*simp add*: *spec.bind.supL spec.bind.supR spec.bind.bind spec.bind.SUPR ac-simps le-supI2*
spec.interference.closed.bind-rel-botR
sup.absorb2 spec.interference.closed.bind spec.interference.least spec.bind.mono
flip: *spec.bind.botR*)

lemma *SupR*:
fixes X :: - *set*
fixes f :: (-, -) *prog*
shows $f \gg (\bigsqcup X) = (\bigsqcup x{\in}X.\ f \gg x) \sqcup (f \gg\!= \bot)$
by (*simp add*: *prog.bind.SUPR*[**where** *g*=$\lambda x\ v.\ x$, *simplified*])

lemma *SUPR-not-empty*:
fixes f :: (-, -) *prog*
assumes $X \neq \{\}$
shows $f \gg\!= (\lambda v.\ \bigsqcup x{\in}X.\ g\ x\ v) = (\bigsqcup x{\in}X.\ f \gg\!= g\ x)$
using *iffD2*[*OF ex-in-conv assms*]
by (*subst trans*[*OF prog.bind.SUPR sup.absorb1*]; *force intro*: *SUPI prog.bind.mono*)

lemma *mcont2mcont*[*cont-intro*]:
assumes *mcont luba orda Sup* (≤) *f*
assumes $\bigwedge v.$ *mcont luba orda Sup* (≤) $(\lambda x.\ g\ x\ v)$
shows *mcont luba orda Sup* (≤) $(\lambda x.$ *prog.bind* $(f\ x)\ (g\ x))$
proof(*rule ccpo.mcont2mcont'*[*OF complete-lattice-ccpo - - assms*(*1*)])
show *mcont Sup* (≤) *Sup* (≤) $(\lambda f.$ *prog.bind f* $(g\ x))$ **for** x
by (*intro mcontI contI monotoneI*) (*simp-all add*: *prog.bind.mono flip*: *prog.bind.SUPL*)
show *mcont luba orda Sup* (≤) $(\lambda x.$ *prog.bind f* $(g\ x))$ **for** f
by (*intro mcontI monotoneI contI*)
(*simp-all add*: *mcont-monoD*[*OF assms*(*2*)] *prog.bind.mono*
flip: *prog.bind.SUPR-not-empty contD*[*OF mcont-cont*[*OF assms*(*2*)]])

qed

lemma *inf-rel*:
  shows *prog.rel r* ⊓ (*f* ⤜ *g*) = *prog.rel r* ⊓ *f* ⤜ (λ*x*. *prog.rel r* ⊓ *g x*)
    and (*f* ⤜ *g*) ⊓ *prog.rel r* = *prog.rel r* ⊓ *f* ⤜ (λ*x*. *prog.rel r* ⊓ *g x*)
by (*transfer*; *simp add*: *spec.bind.inf-rel*)+

setup ‹*Sign.parent-path*›

setup ‹*Sign.mandatory-path guard*›

lemma *bot*:
  shows *prog.guard* ⊥ = ⊥
    and *prog.guard* ⟨*False*⟩ = ⊥
by (*simp-all add*: *prog.guard-def prog.action.empty*)

lemma *top*:
  shows *prog.guard* (⊤::′*a pred*) = *prog.return* () (**is** *?thesis1*)
    and *prog.guard* (⟨*True*⟩::′*a pred*) = *prog.return* () (**is** *?thesis2*)
proof −
  show *?thesis1*
    unfolding *prog.guard-def prog.return-def* **by** *transfer* (*simp add*: *Id-def*)
  then show *?thesis2*
    by (*simp add*: *top-fun-def*)
qed

lemma *return-le*:
  shows *prog.guard g* ≤ *prog.return* ()
unfolding *prog.guard-def Diag-def prog.return-def*
by *transfer* (*blast intro*: *spec.interference.cl.mono spec.action.mono*)

lemma *monotone*:
  shows *mono* (*prog.guard* :: ′*s pred* ⇒ -)
proof(*rule monoI*)
  show *prog.guard g* ≤ *prog.guard h* **if** *g* ≤ *h* **for** *g h* :: ′*s pred*
    unfolding *prog.guard-def*
    by (*strengthen ord-to-strengthen*(*1*)[*OF that*]) (*rule order.refl*)
qed

lemmas *strengthen*[*strg*] = *st-monotone*[*OF prog.guard.monotone*]
lemmas *mono* = *monotoneD*[*OF prog.guard.monotone*]
lemmas *mono2mono*[*cont-intro*, *partial-function-mono*] = *monotone2monotone*[*OF prog.guard.monotone*, *simplified*]

lemma *less*: — Non-triviality
  assumes *g* < *g*′
  shows *prog.guard g* < *prog.guard g*′
proof(*rule le-neq-trans*)
  show *prog.guard g* ≤ *prog.guard g*′
    by (*strengthen ord-to-strengthen*(*1*)[*OF order-less-imp-le*[*OF assms*]]) *simp*
  from *assms* **obtain** *s* **where** *g*′ *s* ¬*g s* **by** (*metis leD predicate1I*)
  from ‹¬*g s*› **have** ¬⦇*trace.T s* [] (*Some* ())⦈ ≤ *prog.p2s* (*prog.guard g*)
    by (*fastforce simp*: *trace.split-all prog.p2s.guard spec.guard-def spec.interference.cl.action*
                spec.singleton.le-conv spec.singleton.action-le-conv trace.steps′.step-conv
          elim: *spec.singleton.bind-le*)
  moreover
  from ‹*g*′ *s*› **have** ⦇*trace.T s* [] (*Some* ())⦈ ≤ *prog.p2s* (*prog.guard g*′)
    by (*force simp*: *prog.p2s.guard prog.p2s.action spec.guard-def*

intro: *order.trans*[*OF - spec.interference.expansive*] *spec.action.stutterI*)
ultimately show *prog.guard g* ≠ *prog.guard g'* **by** *metis*
qed

lemma *if*:
shows (*if b then t else e*) = (*prog.guard* ⟨*b*⟩ ≫ *t*) ⊔ (*prog.guard* ⟨¬*b*⟩ ≫ *e*)
by (*auto simp*: *prog.guard.bot prog.guard.top prog.bind.returnL prog.bind.botL*)

setup ‹*Sign.parent-path*›

setup ‹*Sign.mandatory-path Parallel*›

lemma *bot*:
assumes ⋀*a*. *a* ∈ *bs* ⟹ *Ps a* = ⊥
assumes *as* ∩ *bs* ≠ {}
shows *prog.Parallel as Ps* = *prog.Parallel* (*as* − *bs*) *Ps* ≫= ⊥
by (*auto simp*: *assms*(*1*)
prog.p2s.simps spec.interference.cl.bot
spec.interference.closed.bind-rel-unitR spec.interference.closed.Parallel
spec.term.none.Parallel-some-agents[*OF - assms*(*2*), **where** *Ps'*=⟨*spec.rel* ({*env*} × *UNIV*)⟩]
spec.Parallel.discard-interference[**where** *as*=*as* **and** *bs*=*as* ∩ *bs*]
simp del: *Int-iff*
simp flip: *prog.p2s-inject spec.bind.botR spec.bind.bind*
intro: *arg-cong*[*OF spec.Parallel.cong*])

lemma *mono*:
assumes ⋀*a*. *a* ∈ *as* ⟹ *Ps a* ≤ *Ps' a*
shows *prog.Parallel as Ps* ≤ *prog.Parallel as Ps'*
using *assms* **by** *transfer* (*blast intro*: *spec.Parallel.mono*)

lemma *strengthen-Parallel*[*strg*]:
assumes ⋀*a*. *a* ∈ *as* ⟹ *st-ord F* (*Ps a*) (*Ps' a*)
shows *st-ord F* (*prog.Parallel as Ps*) (*prog.Parallel as Ps'*)
using *assms* **by** (*cases F*) (*auto simp*: *prog.Parallel.mono*)

lemma *mono2mono*[*cont-intro*, *partial-function-mono*]:
assumes ⋀*a*. *a* ∈ *as* ⟹ *monotone orda* (≤) (*F a*)
shows *monotone orda* (≤) (λ*f*. *prog.Parallel as* (λ*a*. *F a f*))
using *assms* **by** *transfer* (*simp add*: *spec.Parallel.mono2mono*)

lemma *cong*:
assumes *as* = *as'*
assumes ⋀*a*. *a* ∈ *as'* ⟹ *Ps a* = *Ps' a*
shows *prog.Parallel as Ps* = *prog.Parallel as' Ps'*
using *assms* **by** *transfer* (*rule spec.Parallel.cong*; *simp*)

lemma *no-agents*:
shows *prog.Parallel* {} *Ps* = *prog.return* ()
by *transfer*
(*simp add*: *spec.Parallel.no-agents spec.interference.cl.return map-prod-image-Times Pair-image*
spec.action.read-agents)

lemma *singleton-agents*:
shows *prog.Parallel* {*a*} *Ps* = *Ps a*
by *transfer* (*rule spec.Parallel.singleton-agents*)

lemma *rename-UNIV*:
assumes *inj-on f as*

shows *prog.Parallel as Ps*
= prog.Parallel UNIV (*λb. if b ∈ f ‘ as then Ps* (*inv-into as f b*) *else prog.return* ())
by (*simp add*: *prog.p2s.simps if-distrib prog.p2s.return spec.interference.cl.return*
spec.Parallel.rename-UNIV[*OF assms*]
flip: *prog.p2s-inject*
cong: *spec.Parallel.cong if-cong*)

lemmas *rename = spec.Parallel.rename*[*transferred*]

lemma *return*:
assumes ⋀*a. a ∈ bs ⟹ Ps a = prog.return* ()
shows *prog.Parallel as Ps = prog.Parallel* (*as − bs*) *Ps*
by (*subst* (*1 2*) *prog.Parallel.rename-UNIV*[**where** *f=id*, *simplified*])
(*auto intro*: *arg-cong*[**where** *f=prog.Parallel UNIV*]
simp: *assms fun-eq-iff f-inv-into-f*[**where** *f=id*, *simplified*])

lemma *unwind*:
assumes *a*: *f* ⤜ *g ≤ Ps a* — The selected process starts with action *f*
assumes *a ∈ as*
shows *f* ⤜ (*λv. prog.Parallel as* (*Ps*(*a*:=*g v*))) *≤ prog.Parallel as Ps*
using *assms* **by** *transfer* (*simp add*: *spec.Parallel.unwind spec.interference.closed.bind-relL*)

setup ‹*Sign.parent-path*›

setup ‹*Sign.mandatory-path parallel*›

lemmas *commute = spec.parallel.commute*[*transferred*]
lemmas *assoc = spec.parallel.assoc*[*transferred*]
lemmas *mono = spec.parallel.mono*[*transferred*]

lemma *strengthen*[*strg*]:
assumes *st-ord F P P′*
assumes *st-ord F Q Q′*
shows *st-ord F* (*prog.parallel P Q*) (*prog.parallel P′ Q′*)
using *assms* **by** (*cases F*; *simp add*: *prog.parallel.mono*)

lemma *mono2mono*[*cont-intro*, *partial-function-mono*]:
assumes *monotone orda* (*≤*) *F*
assumes *monotone orda* (*≤*) *G*
shows *monotone orda* (*≤*) (*λf. prog.parallel* (*F f*) (*G f*))
using *assms* **by** (*simp add*: *monotone-def prog.parallel.mono*)

lemma *bot*:
shows *botL*: *prog.parallel ⊥ P = P* ⤜ *⊥* (**is** *?thesis1*)
and *botR*: *prog.parallel P ⊥ = P* ⤜ *⊥* (**is** *?thesis2*)
proof −
show *?thesis1*
by (*simp add*: *prog.parallel-alt-def prog.Parallel.bot*[**where** *bs*={*True*}, *simplified*]
prog.Parallel.singleton-agents
cong: *prog.Parallel.cong*)
then show *?thesis2*
by (*simp add*: *prog.parallel.commute*)
qed

lemma *return*:
shows *returnL*: *prog.return* () ∥ *P = P* (**is** *?thesis1*)
and *returnR*: *P* ∥ *prog.return* () *= P* (**is** *?thesis2*)
proof −

show *?thesis1*
by (*simp add*: *prog.parallel-alt-def prog.Parallel.return*[**where** *bs*={*True*}, *simplified*]
prog.Parallel.singleton-agents)
then show *?thesis2*
by (*simp add*: *prog.parallel.commute*)
qed

lemma *Sup-not-empty*:
fixes X :: (-, *unit*) *prog set*
assumes $X \neq \{\}$
shows *SupL-not-empty*: $\bigsqcup X \parallel Q = (\bigsqcup P{\in}X.\ P \parallel Q)$ (**is** *?thesis1 Q*)
and *SupR-not-empty*: $P \parallel \bigsqcup X = (\bigsqcup Q{\in}X.\ P \parallel Q)$ (**is** *?thesis2*)
proof −
from *assms* **show** *?thesis1 Q* **for** Q
by (*simp add*: *prog.p2s.parallel prog.p2s.Sup-not-empty*[*OF assms*] *image-image*
spec.parallel.Sup prog.p2s.SUP-not-empty
flip: *prog.p2s-inject*)
then show *?thesis2*
by (*simp add*: *prog.parallel.commute*)
qed

lemma *sup*:
fixes P :: (-, *unit*) *prog*
shows *supL*: $P \sqcup Q \parallel R = (P \parallel R) \sqcup (Q \parallel R)$
and *supR*: $P \parallel Q \sqcup R = (P \parallel Q) \sqcup (P \parallel R)$
using *prog.parallel.SupL-not-empty*[**where** X={P, Q}] *prog.parallel.SupR-not-empty*[**where** X={Q, R}] **by** *simp-all*

lemma *mcont2mcont*[*cont-intro*]:
assumes *mcont luba orda Sup* (≤) P
assumes *mcont luba orda Sup* (≤) Q
shows *mcont luba orda Sup* (≤) (λ*x*. *prog.parallel* (*P x*) (*Q x*))
proof(*rule ccpo.mcont2mcont'*[*OF complete-lattice-ccpo - - assms*(*1*)])
show *mcont Sup* (≤) *Sup* (≤) (λ*y*. *prog.parallel y* (*Q x*)) **for** x
by (*intro mcontI contI monotoneI*) (*simp-all add*: *prog.parallel.mono prog.parallel.SupL-not-empty*)
show *mcont luba orda Sup* (≤) (λ*x*. *prog.parallel y* (*Q x*)) **for** y
by (*simp add*: *mcontI monotoneI contI mcont-monoD*[*OF assms*(*2*)]
spec.parallel.mono mcont-contD[*OF assms*(*2*)] *prog.parallel.SupR-not-empty image-image*)
qed

lemma *unwindL*:
fixes f :: ('*s*, '*v*) *prog*
assumes *a*: $f \gg\!= g \leq P$ — The selected process starts with action f
shows $f \gg\!= (\lambda v.\ g\ v \parallel Q) \leq P \parallel Q$
unfolding *prog.parallel-alt-def*
by (*strengthen ord-to-strengthen*[*OF prog.Parallel.unwind*[**where** *a*=*True*]])
(*auto simp*: *prog.Parallel.mono prog.bind.mono intro*: *assms*)

lemma *unwindR*:
fixes f :: ('*s*, '*v*) *prog*
assumes *a*: $f \gg\!= g \leq Q$ — The selected process starts with action f
shows $f \gg\!= (\lambda v.\ P \parallel g\ v) \leq P \parallel Q$
by (*subst* (*1 2*) *prog.parallel.commute*) (*rule prog.parallel.unwindL*[*OF assms*])

setup ‹*Sign.parent-path*›

setup ‹*Sign.mandatory-path bind*›

lemma *parallel-le*:

fixes P :: (-, -) *prog*
shows $P \gg Q \leq P \parallel Q$
by (*strengthen ord-to-strengthen*[*OF prog.parallel.unwindL*[**where** *g*=*prog.return*, *simplified prog.bind.returnR*, *OF order.refl*]])
(*simp add*: *prog.parallel.return*)

setup ‹*Sign.parent-path*›

setup ‹*Sign.mandatory-path invmap*›

lemma *bot*:
shows *prog.invmap sf vf* $\bot$ = (*prog.rel* (*map-prod sf sf* −‘ *Id*) :: (-, *unit*) *prog*) $\gg\!=$ $\bot$
by (*auto simp*: *prog.p2s.simps spec.interference.cl.bot spec.rel.wind-bind-trailing*
spec.invmap.bind spec.invmap.rel spec.invmap.bot
simp flip: *prog.p2s-inject spec.bind.botR spec.bind.bind bot-fun-def*
intro: *arg-cong*[**where** *f*=λ*r*. *spec.rel r* $\gg\!=$ $\bot$])

lemma *id*:
shows *prog.invmap id id P* = *P*
and *prog.invmap* (λ*x*. *x*) (λ*x*. *x*) *P* = *P*
by (*transfer*; *simp add*: *spec.invmap.id id-def*)+

lemma *comp*:
shows *prog.invmap sf vf* (*prog.invmap sg vg P*) = *prog.invmap* (λ*s*. *sg* (*sf s*)) (λ*s*. *vg* (*vf s*)) *P* (**is** *?thesis1 P*)
and *prog.invmap sf vf* ∘ *prog.invmap sg vg* = *prog.invmap* (*sg* ∘ *sf*) (*vg* ∘ *vf*) (**is** *?thesis2*)
proof −
show *?thesis1 P* **for** *P* **by** *transfer* (*simp add*: *spec.invmap.comp id-def*)
then show *?thesis2* **by** (*simp add*: *comp-def*)
qed

lemma *monotone*:
shows *mono* (*prog.invmap sf vf*)
unfolding *monotone-def* **by** *transfer* (*simp add*: *spec.invmap.mono*)

lemmas *strengthen*[*strg*] = *st-monotone*[*OF prog.invmap.monotone*]
lemmas *mono* = *monotoneD*[*OF prog.invmap.monotone*]

lemma *mono2mono*[*cont-intro*, *partial-function-mono*]:
assumes *monotone orda* (≤) *t*
shows *monotone orda* (≤) (λ*x*. *prog.invmap sf vf* (*t x*))
by (*rule monotone2monotone*[*OF prog.invmap.monotone assms*]) *simp-all*

lemma *Sup*:
fixes *sf* :: *'s* ⇒ *'t*
fixes *vf* :: *'v* ⇒ *'w*
shows *prog.invmap sf vf* (⨆ *X*) = ⨆(*prog.invmap sf vf* ‘ *X*) ⊔ *prog.invmap sf vf* $\bot$
by *transfer*
(*simp add*: *spec.invmap.bot spec.invmap.Sup spec.invmap.sup spec.invmap.bind spec.invmap.rel*
spec.interference.cl.bot map-prod-vimage-Times ac-simps
sup.absorb2 spec.bind.mono[*OF spec.rel.mono order.refl*]
flip: *spec.bind.botR spec.bind.bind spec.rel.unwind-bind-trailing bot-fun-def inv-image-alt-def*)

lemma *Sup-not-empty*:
assumes *X* ≠ {}
shows *prog.invmap sf vf* (⨆ *X*) = ⨆(*prog.invmap sf vf* ‘ *X*)
using *iffD2*[*OF ex-in-conv assms*]
by (*clarsimp simp*: *prog.invmap.Sup sup.absorb1 SUPI prog.invmap.mono*[*OF bot-least*])

lemma *mcont*:
  shows *mcont Sup* (≤) *Sup* (≤) (*prog.invmap sf vf*)
by (*simp add*: *contI mcontI prog.invmap.monotone prog.invmap.Sup-not-empty*)

lemmas *mcont2mcont*[*cont-intro*] = *mcont2mcont*[*OF prog.invmap.mcont*, *of luba orda P* **for** *luba orda P*]

lemma *bind*:
  shows *prog.invmap sf vf* (*f* ⤜ *g*) = *prog.sinvmap sf f* ⤜ (λ*v*. *prog.invmap sf vf* (*g v*))
by *transfer* (*simp add*: *spec.invmap.bind*)

lemma *parallel*:
  shows *prog.invmap sf vf* (*P* ∥ *Q*) = *prog.invmap sf vf P* ∥ *prog.invmap sf vf Q*
by *transfer* (*simp add*: *spec.invmap.parallel*)

lemma *invmap-image-vimage-commute*:
  shows *map-prod id* (*map-prod id sf*) −‘ *map-prod id* (*Pair self*) ‘ *F*
      = *map-prod id* (*Pair self*) ‘ *map-prod id sf* −‘ *F*
by (*auto simp*: *map-prod-conv*)

lemma *action*:
  shows *prog.invmap sf vf* (*prog.action F*)
      = *prog.rel* (*map-prod sf sf* −‘ *Id*)
        ⤜ (λ-::*unit*. *prog.action* (*map-prod id* (*map-prod sf sf*) −‘ *F*)
          ⤜ (λ*v*. *prog.rel* (*map-prod sf sf* −‘ *Id*)
            ⤜ (λ-::*unit*. ⨆ *v*′∈*vf* −‘ {*v*}. *prog.return v*′)))
proof −
  have ∗: {*env*} × *UNIV* ∪ {*self*} × *map-prod sf sf* −‘ *Id*
        = {*env*} × *UNIV* ∪ *UNIV* × *map-prod sf sf* −‘ *Id* **by** *auto*
  show *?thesis*
    by (*simp add*: *ac-simps prog.p2s.simps prog.p2s.action*
          spec.interference.cl.action spec.invmap.bind spec.invmap.rel spec.invmap.action spec.invmap.return
          spec.bind.bind spec.bind.return
          prog.invmap.invmap-image-vimage-commute map-prod-vimage-Times ∗
        flip: *prog.p2s-inject*)
      (*simp add*: *prog.p2s.Sup image-image prog.p2s.simps prog.p2s.return spec.interference.cl.return*
          spec.interference.cl.bot spec.bind.supR
          spec.rel.wind-bind-leading spec.rel.wind-bind-trailing
        flip: *spec.bind.botR spec.bind.SUPR spec.bind.bind*)
qed

setup ‹*Sign.parent-path*›

setup ‹*Sign.mandatory-path vmap*›

lemma *bot*:
  shows *prog.vmap vf* ⊥ = ⊥
by *transfer*
  (*simp add*: *spec.interference.cl.bot spec.vmap.unit-rel*
    flip: *spec.term.none.map-gen*[**where** *vf*=⟨()⟩])

lemma *unitL*:
  shows *f* ≫ *g* = *prog.vmap* ⟨()⟩ *f* ≫ *g*
by *transfer* (*metis spec.vmap.unitL*)

lemma *eq-return*:
  shows *prog.vmap vf P* = *P* ⤜ *prog.return* ∘ *vf* (**is** *?thesis1*)
   and *prog.vmap vf P* = *P* ⤜ (λ*v*. *prog.return* (*vf v*)) (**is** *?thesis2*)
proof −

show *?thesis1*
by *transfer*
(*simp add*: *comp-def spec.vmap.eq-return spec.interference.cl.return spec.interference.closed.bind-relR*)
then show *?thesis2*
by (*simp add*: *comp-def*)
qed

lemma *action*:
shows *prog.vmap vf* (*prog.action F*) = *prog.action* (*map-prod vf id* ‘ *F*)
by *transfer* (*simp add*: *spec.map.interference.cl-sf-id spec.map.surj-sf-action image-comp*)

lemma *return*:
shows *prog.vmap vf* (*prog.return v*) = *prog.return* (*vf v*)
by (*simp add*: *prog.return-def prog.vmap.action map-prod-image-Times*)

setup ‹*Sign.parent-path*›

interpretation *kleene*: *kleene prog.return* () λ*x y*. *prog.bind x* ⟨*y*⟩
by *standard*
(*simp-all add*: *prog.bind.bind prog.bind.return prog.bind.botL prog.bind.supL prog.bind.supR*)

interpretation *rel*: *galois.complete-lattice-class prog.steps prog.rel*
proof
show *prog.steps P* ⊆ *r* ⟷ *P* ≤ *prog.rel r* **for** *P* :: (′*a*, ′*b*) *prog* **and** *r* :: ′*a rel*
by *transfer* (*auto simp flip*: *spec.rel.galois*)
qed

setup ‹*Sign.mandatory-path rel*›

lemma *empty*:
shows *prog.rel* {} = ⨆ *range prog.return*
by (*simp add*: *prog.p2s.simps prog.p2s.return image-image*
spec.interference.cl.bot spec.interference.cl.return
spec.term.closed.bind-all-return[*OF spec.term.closed.rel*] *spec.term.all.rel*
sup.absorb1 spec.term.galois
flip: *prog.p2s-inject spec.bind.SUPR-not-empty*)

lemmas *monotone* = *prog.rel.monotone-upper*
lemmas *strengthen*[*strg*] = *st-monotone*[*OF prog.rel.monotone*]
lemmas *mono* = *monotoneD*[*OF prog.rel.monotone*]

lemmas *Inf* = *prog.rel.upper-Inf*
lemmas *inf* = *prog.rel.upper-inf*

lemma *reflcl*:
shows *prog.rel* (*r* ∪ *Id*) = (*prog.rel r* :: (′*s*, ′*v*) *prog*) (**is** *?thesis1*)
and *prog.rel* (*Id* ∪ *r*) = (*prog.rel r* :: (′*s*, ′*v*) *prog*) (**is** *?thesis2*)
proof −
show *?thesis1*
by *transfer*
(*subst* (*2*) *spec.rel.reflcl*[**where** *A*=*UNIV*, *symmetric*];
auto intro: *arg-cong*[**where** *f*=*spec.rel*])
then show *?thesis2*
by (*simp add*: *ac-simps*)
qed

lemma *minus-Id*:
shows *prog.rel* (*r* − *Id*) = *prog.rel r*

by (*metis Un-Diff-cancel2 prog.rel.reflcl*(*1*))

lemma *Id*:
shows *prog.rel Id* = ⨆ *range prog.return*
by (*simp add*: *prog.rel.reflcl*(*1*)[**where** *r*={}, *simplified*] *prog.rel.empty*)

lemma *unfoldL*:
fixes *r* :: *'s rel*
assumes *Id* ⊆ *r*
shows *prog.rel r* = *prog.action* ({()} × *r*) ≫ *prog.rel r*
proof −
have ∗: *spec.rel* ({*env*} × *UNIV*)
≫= (λ*v*::*unit*. *spec.action* (*map-prod id* (*Pair self*) ‘ ({()} × *r*))
≫= (λ*v*::*unit*. *spec.rel* ({*env*} × *UNIV* ∪ {*self*} × *r*)))
= *spec.rel* ({*env*} × *UNIV* ∪ {*self*} × *r*) (**is** *?lhs* = *?rhs*)
if *Id* ⊆ *r*
for *r* :: *'s rel*
proof(*rule antisym*)
let *?r'* = {*env*} × *UNIV* ∪ {*self*} × *r*
have *?lhs* ≤ *spec.rel ?r'* ≫= (λ-::*unit*. *spec.rel ?r'* ≫= (λ-::*unit*. *spec.rel ?r'*))
by (*fastforce intro*: *spec.bind.mono spec.rel.mono spec.action.mono*
order.trans[*OF* - *spec.rel.monomorphic-act-le*]
simp: *spec.rel.act-def*)
also have ... = *?rhs*
by (*simp add*: *spec.rel.wind-bind*)
finally show *?lhs* ≤ *?rhs* **.**
from *that* **show** *?rhs* ≤ *?lhs*
apply −
apply (*rule order.trans*[*OF* -
spec.bind.mono[*OF spec.return.rel-le*
spec.bind.mono[*OF spec.action.mono*[**where** *x*={()} × {*self*} × *Id*] *order.refl*]]])
apply (*subst spec.return.cong*; *simp add*: *image-image spec.bind.supL spec.bind.supR spec.bind.returnL spec.idle-le*)
apply (*fastforce simp*: *map-prod-image-Times*)
done
qed
from *assms* **show** *?thesis*
by *transfer* (*simp add*: ∗ *spec.interference.cl.action spec.bind.bind spec.rel.wind-bind-leading*)
qed

lemma *wind-bind*: — arbitrary interstitial return type
shows *prog.rel r* ≫ *prog.rel r* = *prog.rel r*
by *transfer* (*simp add*: *spec.rel.wind-bind*)

lemma *wind-bind-leading*: — arbitrary interstitial return type
assumes *r'* ⊆ *r*
shows *prog.rel r'* ≫ *prog.rel r* = *prog.rel r*
using *assms* **by** *transfer* (*subst spec.rel.wind-bind-leading*; *blast*)

lemma *wind-bind-trailing*: — arbitrary interstitial return type
assumes *r'* ⊆ *r*
shows *prog.rel r* ≫ *prog.rel r'* = *prog.rel r* (**is** *?lhs* = *?rhs*)
using *assms* **by** *transfer* (*subst spec.rel.wind-bind-trailing*; *blast*)

Interstitial unit, for unfolding

lemmas *unwind-bind* = *prog.rel.wind-bind*[**where** *'c*=*unit*, *symmetric*]
lemmas *unwind-bind-leading* = *prog.rel.wind-bind-leading*[**where** *'c*=*unit*, *symmetric*]
lemmas *unwind-bind-trailing* = *prog.rel.wind-bind-trailing*[**where** *'c*=*unit*, *symmetric*]

lemma *mono-conv*:
shows *prog.rel r = prog.kleene.star (prog.action ({()} × r⁼)) (***is** *?lhs = ?rhs)*
proof(*rule antisym*)
have *spec.kleene.star (spec.rel.act ({env} × UNIV ∪ {self} × r)) ≤ prog.p2s ?rhs*
proof(*induct rule*: *spec.kleene.star.fixp-induct*[*case-names adm bot step*])
case (*step R*) **show** *?case*
proof(*induct rule*: *le-supI*[*case-names act-step ret*])
case *act-step*
have ∗: *spec.rel.act ({env} × UNIV ∪ {self} × r) ≤ prog.p2s (prog.action ({()} × r⁼))*
by (*auto simp*: *spec.rel.act-alt-def Times-Un-distrib2 spec.action.sup*
prog.p2s.sup prog.p2s.return spec.interference.cl.return prog.action.sup map-prod-conv
simp flip: *prog.return-def*
intro: *spec.action.mono le-supI2 spec.action.rel-le spec.return.rel-le*
le-supI1[*OF order.trans*[*OF spec.action.mono prog.action.action-le*]])
show *?case*
apply (*subst prog.kleene.star.simps*)
apply (*strengthen ord-to-strengthen*(*1*)[*OF step*])
apply (*simp add*: *prog.p2s.simps le-supI1*[*OF spec.bind.mono*[*OF ∗ order.refl*]])
done
next
case *ret* **show** *?case*
by (*simp add*: *order.trans*[*OF - prog.p2s.mono*[*OF prog.kleene.epsilon-star-le*]]
prog.p2s.return spec.interference.expansive)
qed
qed *simp-all*
then show *?lhs ≤ ?rhs*
by (*simp add*: *prog.p2s-leI prog.p2s.simps spec.rel.monomorphic-conv*)
show *?rhs ≤ ?lhs*
proof(*induct rule*: *prog.kleene.star.fixp-induct*[*case-names adm bot step*])
case (*step R*) **show** *?case*
apply (*strengthen ord-to-strengthen*(*1*)[*OF step*])
apply (*simp add*: *prog.return.rel-le*)
apply (*subst* (*2*) *prog.rel.unwind-bind*)
apply (*auto intro*: *prog.bind.mono prog.action.rel-le*)
done
qed *simp-all*
qed

setup ‹*Sign.parent-path*›

setup ‹*Sign.mandatory-path action*›

lemma *inf-rel*:
assumes *refl r*
shows *prog.action F ⊓ prog.rel r = prog.action (F ∩ UNIV × r) (***is** *?thesis1*)
and *prog.rel r ⊓ prog.action F = prog.action (F ∩ UNIV × r) (***is** *?thesis2*)
proof −
from *assms* **have** *refl (({env} × UNIV ∪ {self} × r) `` {a}) for a*
by (*fastforce dest*: *reflD intro*: *reflI*)
then show *?thesis1*
by *transfer* (*simp add*: *spec.interference.cl.inf-rel spec.action.inf-rel*; *rule arg-cong*; *blast*)
then show *?thesis2*
by (*rule inf-commute-conv*)
qed

lemma *inf-rel-reflcl*:
shows *prog.action F ⊓ prog.rel r = prog.action (F ∩ UNIV × r⁼)*

and *prog.rel r* ⊓ *prog.action F* = *prog.action* (*F* ∩ *UNIV* × *r*=)
by (*simp-all add*: *refl-on-def prog.rel.reflcl ac-simps flip*: *prog.action.inf-rel*)

setup ‹*Sign.parent-path*›

setup ‹*Sign.mandatory-path return*›

lemma *not-bot*:
shows *prog.return v* ≠ (⊥ :: ('*s*, '*v*) *prog*)
using *prog.guard.less*[**where** *g*=⊥::'*s pred* **and** *g'*=*top*]
by (*force dest*: *arg-cong*[**where** *f*=*prog.vmap* (λ-::'*v*. ())]
simp: *prog.vmap.return prog.vmap.bot fun-eq-iff prog.guard.bot prog.guard.top*
simp flip: *top.not-eq-extremum*)

setup ‹*Sign.parent-path*›

setup ‹*Sign.mandatory-path invmap*›

lemma *return*:
shows *prog.invmap sf vf* (*prog.return v*)
= *prog.rel* (*map-prod sf sf* −‘ *Id*) ⤜ (λ-::*unit*. ⨆ *v'*∈*vf* −‘ {*v*}. *prog.return v'*)
apply (*simp add*: *prog.return-def prog.invmap.action map-prod-vimage-Times*)
apply (*simp add*: *prog.action.return-const*[**where** *V*={*v*} **and** *W*={()}] *prog.bind.bind prog.bind.return*)
apply (*subst prog.bind.bind*[*symmetric*], *subst prog.rel.unfoldL*[*symmetric*];
force simp: *prog.rel.wind-bind simp flip*: *prog.bind.bind*)
done

lemma *split-vinvmap*:
fixes *P* :: ('*s*, '*v*) *prog*
shows *prog.invmap sf vf P* = *prog.sinvmap sf P* ⤜ (λ*v*. ⨆ *v'*∈*vf* −‘ {*v*}. *prog.return v'*)
proof −
note *sic-invmap* = *spec.interference.closed.invmap*[**where** *af*=*id* **and** *r*={*env*} × *UNIV*,
simplified map-prod-vimage-Times, *simplified*]
show *?thesis*
apply *transfer*
apply (*simp add*: *spec.bind.supR sup.absorb1 spec.interference.cl.bot bot-fun-def*
spec.interference.closed.bind-relR sic-invmap spec.bind.mono
flip: *spec.bind.botR*)
apply (*subst* (*1*) *spec.invmap.split-vinvmap*)
apply (*subst* (*1*) *spec.interference.closed.bind-relR*[*symmetric*], *erule sic-invmap*)
apply (*simp add*: *spec.bind.SUPR spec.bind.supR spec.interference.cl.return*
sup.absorb1 bot-fun-def spec.interference.closed.bind-relR sic-invmap spec.bind.mono)
done
qed

setup ‹*Sign.parent-path*›

setup ‹*Sign.parent-path*›

13.4 Refinement for *('s, 'v) prog*

We specialize the rules of §12.1 to the ('*s*, '*v*) *prog* lattice. Observe that, as preconditions, postconditions and assumes are not interference closed, we apply the *prog.p2s* morphism and work in the more capacious (*sequential*, '*s*, '*v*) *spec* lattice. This syntactic noise could be elided with another definition.

13.4.1 Introduction rules

Refinement is a way of showing inequalities and equalities between programs.

setup ‹*Sign.mandatory-path refinement.prog*›

lemma *leI*:
  assumes *prog.p2s c* ≤ ⦃⟨*True*⟩⦄, ⊤ ⊩ *prog.p2s d*, ⦃λ-. ⟨*True*⟩⦄
  shows *c* ≤ *d*
using *assms* **by** (*simp add*: *refinement-def prog.p2s-leI*)

lemma *eqI*:
  assumes *prog.p2s c* ≤ ⦃⟨*True*⟩⦄, ⊤ ⊩ *prog.p2s d*, ⦃λ-. ⟨*True*⟩⦄
  assumes *prog.p2s d* ≤ ⦃⟨*True*⟩⦄, ⊤ ⊩ *prog.p2s c*, ⦃λ-. ⟨*True*⟩⦄
  shows *c* = *d*
by (*rule antisym*; *simp add*: *assms refinement.prog.leI*)

setup ‹*Sign.parent-path*›

13.4.2 Galois considerations

Refinement quadruples ⦃*P*⦄, *A* ⊩ *G*, ⦃*Q*⦄ denote points in the (′*s*, ′*v*) *prog* lattice provided *G* is suitably interference closed.

setup ‹*Sign.mandatory-path refinement.prog*›

lemma *galois*:
  assumes *spec.term.none* (*spec.rel* ({*env*} × *UNIV*) :: (-, -, *unit*) *spec*) ≤ *G*
  shows *prog.p2s c* ≤ ⦃*P*⦄, *A* ⊩ *G*, ⦃*Q*⦄ ⟷ *c* ≤ *prog.s2p* (⦃*P*⦄, *A* ⊩ *G*, ⦃*Q*⦄)
by (*simp add*: *assms prog.p2s-s2p.galois refinement-def spec.next-imp.contains spec.term.none.post-le*)

lemmas *s2p-refinement* = *iffD1*[*OF refinement.prog.galois*, *rotated*]

lemma *p2s-s2p*:
  assumes *spec.term.none* (*spec.rel* ({*env*} × *UNIV*) :: (-, -, *unit*) *spec*) ≤ *G*
  shows *prog.p2s* (*prog.s2p* (⦃*P*⦄, *A* ⊩ *G*, ⦃*Q*⦄)) ≤ ⦃*P*⦄, *A* ⊩ *G*, ⦃*Q*⦄
using *assms* **by** (*simp add*: *refinement.prog.galois*)

setup ‹*Sign.parent-path*›

13.4.3 Rules

setup ‹*Sign.mandatory-path refinement.prog*›

lemma *bot*[*iff*]:
  shows *prog.p2s* ⊥ ≤ ⦃*P*⦄, *A* ⊩ *prog.p2s c*′, ⦃*Q*⦄
by (*simp add*: *refinement.prog.galois spec.term.none.interference.closed.rel-le*)

lemma *sup-conv*:
  shows *prog.p2s* (*c*₁ ⊔ *c*₂) ≤ ⦃*P*⦄, *A* ⊩ *G*, ⦃*Q*⦄
    ⟷ *prog.p2s c*₁ ≤ ⦃*P*⦄, *A* ⊩ *G*, ⦃*Q*⦄ ∧ *prog.p2s c*₂ ≤ ⦃*P*⦄, *A* ⊩ *G*, ⦃*Q*⦄
by (*simp add*: *prog.p2s.simps*)

lemmas *sup* = *iffD2*[*OF refinement.prog.sup-conv*, *unfolded conj-explode*]

lemma *if*:
  assumes *i* ⟹ *prog.p2s t* ≤ ⦃*P*⦄, *A* ⊩ *prog.p2s t*′, ⦃*Q*⦄
  assumes ¬*i* ⟹ *prog.p2s e* ≤ ⦃*P*′⦄, *A* ⊩ *prog.p2s e*′, ⦃*Q*⦄
  shows *prog.p2s* (*if i then t else e*) ≤ ⦃*if i then P else P*′⦄, *A* ⊩ *prog.p2s* (*if i then t*′ *else e*′), ⦃*Q*⦄
using *assms* **by** *fastforce*

lemmas *if*′ = *refinement.prog.if*[**where** *P*=*P* **and** *P*′=*P*, *simplified*] **for** *P*

lemma *case-option*:
assumes *opt* = *None* ⟹ *prog.p2s none* ≤ ⦃P_n⦄, *A* ⊨ *prog.p2s none′*, ⦃*Q*⦄
assumes ⋀*v*. *opt* = *Some v* ⟹ *prog.p2s* (*some v*) ≤ ⦃P_s *v*⦄, *A* ⊨ *prog.p2s* (*some′ v*), ⦃*Q*⦄
shows *prog.p2s* (*case-option none some opt*) ≤ ⦃*case opt of None* ⇒ P_n | *Some v* ⇒ P_s *v*⦄, *A* ⊨ *prog.p2s* (*case-option none′ some′ opt*), ⦃*Q*⦄
using *assms* **by** (*simp add*: *option.case-eq-if*)

lemma *case-sum*:
assumes ⋀*v*. *x* = *Inl v* ⟹ *prog.p2s* (*left v*) ≤ ⦃P_l *v*⦄, *A* ⊨ *prog.p2s* (*left′ v*), ⦃*Q*⦄
assumes ⋀*v*. *x* = *Inr v* ⟹ *prog.p2s* (*right v*) ≤ ⦃P_r *v*⦄, *A* ⊨ *prog.p2s* (*right′ v*), ⦃*Q*⦄
shows *prog.p2s* (*case-sum left right x*) ≤ ⦃*case-sum* P_l P_r *x*⦄, *A* ⊨ *prog.p2s* (*case-sum left′ right′ x*), ⦃*Q*⦄
using *assms* **by** (*simp add*: *sum.case-eq-if*)

lemma *case-list*:
assumes *x* = [] ⟹ *prog.p2s nil* ≤ ⦃P_n⦄, *A* ⊨ *prog.p2s nil′*, ⦃*Q*⦄
assumes ⋀*v vs*. *x* = *v* # *vs* ⟹ *prog.p2s* (*cons v vs*) ≤ ⦃P_c *v vs*⦄, *A* ⊨ *prog.p2s* (*cons′ v vs*), ⦃*Q*⦄
shows *prog.p2s* (*case-list nil cons x*) ≤ ⦃*case-list* P_n P_c *x*⦄, *A* ⊨ *prog.p2s* (*case-list nil′ cons′ x*), ⦃*Q*⦄
using *assms* **by** (*simp add*: *list.case-eq-if*)

lemma *action*:
fixes *F* :: (′*v* × ′*s* × ′*s*) *set*
assumes ⋀*v s s′*. ⟦*P s*; (*v*, *s*, *s′*) ∈ *F*; (*self*, *s*, *s′*) ∈ *spec.steps A* ∨ *s* = *s′*⟧ ⟹ *Q v s′*
assumes ⋀*v s s′*. ⟦*P s*; (*v*, *s*, *s′*) ∈ *F*⟧ ⟹ (*v*, *s*, *s′*) ∈ *F′*
assumes *sP*: *stable* (*spec.steps A* ‘‘ {*env*}) *P*
assumes ⋀*v s s′*. ⟦*P s*; (*v*, *s*, *s′*) ∈ *F*⟧ ⟹ *stable* (*spec.steps A* ‘‘ {*env*}) (*Q v*)
shows *prog.p2s* (*prog.action F*) ≤ ⦃*P*⦄, *A* ⊨ *prog.p2s* (*prog.action F′*), ⦃*Q*⦄
unfolding *prog.p2s.action spec.interference.cl.action*
apply (*rule refinement.pre-a*[*OF - spec.rel.upper-lower-expansive*])
apply (*rule refinement.spec.bind*[*rotated*])
apply (*rule refinement.spec.rel-mono*[*OF order.refl*];
fastforce simp: *spec.term.all.rel spec.steps.rel*
intro: *sP antimonoD*[*OF stable.antimono-rel*, *unfolded le-bool-def*, *rule-format*, *rotated*])
apply (*strengthen ord-to-strengthen*(*1*)[*OF inf-le2*])
apply (*strengthen ord-to-strengthen*(*1*)[*OF refinement.spec.bind.res.rel-le*[*OF order.refl*]])
apply (*rule refinement.spec.bind*[*rotated*, **where** *Q′*=λ*v s*. *Q v s* ∧ *stable* (*spec.steps A* ‘‘ {*env*}) (*Q v*)])
apply (*rule refinement.spec.action*;
fastforce simp: *spec.initial-steps.term.all spec.initial-steps.rel*
intro: *assms*)
apply (*rule refinement.gen-asm*)
apply (*rule refinement.spec.bind*[*rotated*])
apply (*rule refinement.spec.rel-mono*[*OF order.refl*];
fastforce simp: *spec.steps.term.all spec.rel.lower-upper-lower*
elim: *antimonoD*[*OF stable.antimono-rel*, *unfolded le-bool-def*, *rule-format*, *rotated*]
dest: *subsetD*[*OF spec.steps.refinement.spec.bind.res-le*])
apply (*rule refinement.spec.return*)
apply (*simp only*: *spec.idle-le*)
done

lemma *return*:
assumes *sQ*: *stable* (*spec.steps A* ‘‘ {*env*}) (*Q v*)
shows *prog.p2s* (*prog.return v*) ≤ ⦃*Q v*⦄, *A* ⊨ *prog.p2s* (*prog.return v*), ⦃*Q*⦄
unfolding *prog.return-def* **using** *assms* **by** (*blast intro*: *refinement.prog.action*)

lemma *invmap-return*:
assumes *sQ*: *stable* (*spec.steps A* ‘‘ {*env*}) (*Q v*)
assumes *vf v* = *v′*
shows *prog.p2s* (*prog.return v*) ≤ ⦃*Q v*⦄, *A* ⊨ *prog.p2s* (*prog.invmap sf vf* (*prog.return v′*)), ⦃*Q*⦄
unfolding *prog.invmap.return*

by (*strengthen ord-to-strengthen*(2)[*OF prog.return.rel-le*])
(*simp add*: *assms*(2) *refinement.pre-g*[*OF refinement.prog.return*[**where** *Q*=*Q*, *OF sQ*]]
SUP-upper prog.bind.return prog.p2s.mono)

lemma *bind-abstract*:
fixes *f* :: ('*s*, '*v*) *prog*
fixes *f'* :: ('*s*, '*v'*) *prog*
fixes *g* :: '*v* ⇒ ('*s*, '*w*) *prog*
fixes *g'* :: '*v'* ⇒ ('*s*, '*w*) *prog*
fixes *vf* :: '*v* ⇒ '*v'*
assumes ⋀*v*. *prog.p2s* (*g v*) ≤ ⦃*Q'* (*vf v*)⦄, *refinement.spec.bind.res* (*spec.pre P* ⊓ *spec.term.all A* ⊓ *prog.p2s f'*) *A* (*vf v*) ⊨ *prog.p2s* (*g'* (*vf v*)), ⦃*Q*⦄
assumes *prog.p2s f* ≤ ⦃*P*⦄, *spec.term.all A* ⊨ *spec.vinvmap vf* (*prog.p2s f'*), ⦃λ*v*. *Q'* (*vf v*)⦄
shows *prog.p2s* (*f* ⤜ *g*) ≤ ⦃*P*⦄, *A* ⊨ *prog.p2s* (*f'* ⤜ *g'*), ⦃*Q*⦄
by (*simp add*: *prog.p2s.simps refinement.spec.bind-abstract*[*OF assms*])

lemma *bind*:
assumes ⋀*v*. *prog.p2s* (*g v*) ≤ ⦃*Q'* *v*⦄, *refinement.spec.bind.res* (*spec.pre P* ⊓ *spec.term.all A* ⊓ *prog.p2s f'*) *A* *v* ⊨ *prog.p2s* (*g'* *v*), ⦃*Q*⦄
assumes *prog.p2s f* ≤ ⦃*P*⦄, *spec.term.all A* ⊨ *prog.p2s f'*, ⦃*Q'*⦄
shows *prog.p2s* (*f* ⤜ *g*) ≤ ⦃*P*⦄, *A* ⊨ *prog.p2s* (*f'* ⤜ *g'*), ⦃*Q*⦄
by (*simp add*: *prog.p2s.simps refinement.spec.bind*[*OF assms*])

lemmas *rev-bind* = *refinement.prog.bind*[*rotated*]

lemma *Parallel*:
fixes *A* :: (*sequential*, '*s*, *unit*) *spec*
fixes *Q* :: '*a* ⇒ '*s pred*
fixes *Ps* :: '*a* ⇒ ('*s*, *unit*) *prog*
fixes *Ps'* :: '*a* ⇒ ('*s*, *unit*) *prog*
assumes ⋀*a*. *a* ∈ *as* ⟹ *prog.p2s* (*Ps a*) ≤ ⦃*P a*⦄, *refinement.spec.env-hyp P A as* (*prog.p2s* ∘ *Ps'*) *a* ⊨ *prog.p2s* (*Ps'* *a*), ⦃λ*rv*. *Q a*⦄
shows *prog.p2s* (*prog.Parallel as Ps*) ≤ ⦃⨅ *a*∈*as*. *P a*⦄, *A* ⊨ *prog.p2s* (*prog.Parallel as Ps'*), ⦃λ*rv*. ⨅ *a*∈*as*. *Q a*⦄
using *assms* **by** *transfer* (*simp add*: *refinement.spec.Parallel comp-def*)

lemma *parallel*:
assumes *prog.p2s* c_1 ≤ ⦃P_1⦄, *refinement.spec.env-hyp* (λ*a*. *if a then* P_1 *else* P_2) *A UNIV* (λ*a*. *if a then prog.p2s* c_1' *else prog.p2s* c_2') *True* ⊨ *prog.p2s* c_1', ⦃Q_1⦄
assumes *prog.p2s* c_2 ≤ ⦃P_2⦄, *refinement.spec.env-hyp* (λ*a*. *if a then* P_1 *else* P_2) *A UNIV* (λ*a*. *if a then prog.p2s* c_1' *else prog.p2s* c_2') *False* ⊨ *prog.p2s* c_2', ⦃Q_2⦄
shows *prog.p2s* (*prog.parallel* c_1 c_2) ≤ ⦃P_1 ∧ P_2⦄, *A* ⊨ *prog.p2s* (*prog.parallel* c_1' c_2'), ⦃λ*v*. Q_1 *v* ∧ Q_2 *v*⦄
unfolding *prog.parallel-alt-def*
by (*rule refinement.pre*[*OF refinement.prog.Parallel*[**where** *A*=*A* **and** *P*=λ*a*. *if a then* P_1 *else* P_2 **and** *Ps'*=λ*a*. *if a then* c_1' *else* c_2' **and** *Q*=λ*a*. *if a then* Q_1 () *else* Q_2 ()]])
(*use assms* **in** ‹*force simp*: *if-distrib comp-def*›)+

setup ‹*Sign.parent-path*›

13.5 A relational assume/guarantee program logic for the *('s, 'v) prog* lattice

Similarly we specialize the assume/guarantee program logic of §12.2 to ('*s*, '*v*) *prog*.
References:

- de Roever, de Boer, Hannemann, Hooman, Lakhnech, Poel, and Zwiers (2001); Xu, de Roever, and He (1997)
- Prensa Nieto (2003, §7)
- Vafeiadis (2008, §3)

13.5.1 Galois considerations

For suitably stable P, Q, $\{\!|P|\!\}$, $A \vdash G$, $\{\!|Q|\!\}$ is interference closed and hence denotes a point in $('s, 'v)$ *prog*. In other words we can replace programs with their specifications.

setup ‹*Sign.mandatory-path ag.prog*›

lemma *galois*:
  shows *prog.p2s c* ≤ $\{\!|P|\!\}$, $A \vdash G$, $\{\!|Q|\!\}$ ⟷ *c* ≤ *prog.s2p* ($\{\!|P|\!\}$, $A \vdash G$, $\{\!|Q|\!\}$)
by (*simp add*: *prog.p2s-s2p.galois ag.spec.term.none-inteference*)

lemmas *s2p-ag* = *iffD1*[*OF ag.prog.galois*]

lemma *p2s-s2p-ag*:
  shows *prog.p2s* (*prog.s2p* ($\{\!|P|\!\}$, $A \vdash G$, $\{\!|Q|\!\}$)) ≤ $\{\!|P|\!\}$, $A \vdash G$, $\{\!|Q|\!\}$
by (*simp add*: *ag.prog.galois*)

lemma *p2s-s2p-ag-stable*:
  assumes *stable A P*
  assumes ⋀*v*. *stable A* (*Q v*)
  shows *prog.p2s* (*prog.s2p* ($\{\!|P|\!\}$, $A \vdash G$, $\{\!|Q|\!\}$)) = $\{\!|P|\!\}$, $A \vdash G$, $\{\!|Q|\!\}$
by (*rule prog.p2s-s2p.insertion*[*OF spec.interference.closed-ag*[**where** *r*=*UNIV*, *simplified*, *OF assms*]])

setup ‹*Sign.parent-path*›

setup ‹*Sign.mandatory-path prog.ag*›

lemma *bot*[*iff*]:
  shows *prog.p2s* ⊥ ≤ $\{\!|P|\!\}$, $A \vdash G$, $\{\!|Q|\!\}$
by (*simp add*: *ag.prog.galois*)

setup ‹*Sign.parent-path*›

setup ‹*Sign.mandatory-path ag*›

setup ‹*Sign.mandatory-path prog*›

lemma *sup-conv*:
  shows *prog.p2s* (*c*$_1$ ⊔ *c*$_2$) ≤ $\{\!|P|\!\}$, $A \vdash G$, $\{\!|Q|\!\}$ ⟷ *prog.p2s c*$_1$ ≤ $\{\!|P|\!\}$, $A \vdash G$, $\{\!|Q|\!\}$ ∧ *prog.p2s c*$_2$ ≤ $\{\!|P|\!\}$, $A \vdash G$, $\{\!|Q|\!\}$
by (*simp add*: *prog.p2s.simps*)

lemmas *sup* = *iffD2*[*OF ag.prog.sup-conv*, *unfolded conj-explode*]

lemma *bind*: — Assumptions in weakest-pre order
  assumes ⋀*v*. *prog.p2s* (*g v*) ≤ $\{\!|Q'\ v|\!\}$, $A \vdash G$, $\{\!|Q|\!\}$
  assumes *prog.p2s f* ≤ $\{\!|P|\!\}$, $A \vdash G$, $\{\!|Q'|\!\}$
  shows *prog.p2s* (*f* ⤜ *g*) ≤ $\{\!|P|\!\}$, $A \vdash G$, $\{\!|Q|\!\}$
by (*simp add*: *prog.p2s.simps*) (*rule ag.spec.bind*; *fact*)

lemma *action*: — Conclusion is insufficiently instantiated for use
  fixes *F* :: (*'v* × *'s* × *'s*) *set*
  assumes *Q*: ⋀*v s s'*. ⟦*P s*; (*v*, *s*, *s'*) ∈ *F*⟧ ⟹ *Q v s'*
  assumes *G*: ⋀*v s s'*. ⟦*P s*; *s* ≠ *s'*; (*v*, *s*, *s'*) ∈ *F*⟧ ⟹ (*s*, *s'*) ∈ *G*
  assumes *sP*: *stable A P*
  assumes *sQ*: ⋀*s s' v*. ⟦*P s*; (*v*, *s*, *s'*) ∈ *F*⟧ ⟹ *stable A* (*Q v*)
  shows *prog.p2s* (*prog.action F*) ≤ $\{\!|P|\!\}$, $A \vdash G$, $\{\!|Q|\!\}$
unfolding *prog.p2s.action spec.interference.cl.action* — sp proof
by (*rule ag.gen-asm*

ag.spec.bind[*rotated*] *ag.spec.stable-interference ag.spec.return*
ag.spec.action[**where** $Q{=}\lambda v\ s.\ Q\ v\ s \wedge (\exists s\ s'.\ P\ s \wedge (v,\ s,\ s') \in F)$]
| *use assms* **in** *auto*)+

lemma *guard*:
assumes $\bigwedge s.\ [\![P\ s;\ g\ s]\!] \Longrightarrow Q\ ()\ s$
assumes *stable A P*
assumes *stable A* (*Q* ())
shows *prog.p2s* (*prog.guard g*) $\leq \{\!|P|\!\}, A \vdash G, \{\!|Q|\!\}$
using *assms* **by** (*fastforce simp*: *prog.guard-def intro*: *ag.prog.action split*: *if-splits*)

lemma *Parallel*:
assumes $\bigwedge a.\ a \in as \Longrightarrow$ *prog.p2s* (*Ps a*) $\leq \{\!|P\ a|\!\}, A \cup (\bigcup a' {\in} as - \{a\}.\ G\ a') \vdash G\ a, \{\!|\lambda v.\ Q\ a|\!\}$
shows *prog.p2s* (*prog.Parallel as Ps*) $\leq \{\!|\prod a{\in}as.\ P\ a|\!\}, A \vdash \bigcup a{\in}as.\ G\ a, \{\!|\lambda v.\ \prod a{\in}as.\ Q\ a|\!\}$
using *assms* **by** *transfer* (*fast intro*: *ag.spec.Parallel*)

lemma *parallel*:
assumes *prog.p2s* $c_1 \leq \{\!|P_1|\!\}, A \cup G_2 \vdash G_1, \{\!|Q_1|\!\}$
assumes *prog.p2s* $c_2 \leq \{\!|P_2|\!\}, A \cup G_1 \vdash G_2, \{\!|Q_2|\!\}$
shows *prog.p2s* (*prog.parallel* $c_1\ c_2$) $\leq \{\!|P_1 \wedge P_2|\!\}, A \vdash G_1 \cup G_2, \{\!|\lambda v.\ Q_1\ v \wedge Q_2\ v|\!\}$
unfolding *prog.parallel-alt-def*
by (*rule ag.pre*[*OF ag.prog.Parallel*[**where** $A{=}A$ **and** $G{=}\lambda a.$ *if a then* G_1 *else* G_2 **and** $P{=}\langle P_1 \wedge P_2\rangle$ **and** $Q{=}\lambda a.$ *if a then* Q_1 () *else* Q_2 ()]])
(*use assms* **in** ‹*auto intro*: *ag.pre-imp*›)

lemma *return*:
assumes *sQ*: *stable A* (*Q v*)
shows *prog.p2s* (*prog.return v*) $\leq \{\!|Q\ v|\!\}, A \vdash G, \{\!|Q|\!\}$
using *assms* **by** (*auto simp*: *prog.return-def intro*: *ag.prog.action*)

lemma *if*:
assumes $b \Longrightarrow$ *prog.p2s* $c_1 \leq \{\!|P_1|\!\}, A \vdash G, \{\!|Q|\!\}$
assumes $\neg b \Longrightarrow$ *prog.p2s* $c_2 \leq \{\!|P_2|\!\}, A \vdash G, \{\!|Q|\!\}$
shows *prog.p2s* (*if b then* c_1 *else* c_2) $\leq \{\!|$*if b then* P_1 *else* $P_2|\!\}, A \vdash G, \{\!|Q|\!\}$
using *assms* **by** (*fastforce intro*: *ag.pre-ag*)

lemma *case-option*:
assumes $x = None \Longrightarrow$ *prog.p2s none* $\leq \{\!|P_n|\!\}, A \vdash G, \{\!|Q|\!\}$
assumes $\bigwedge v.\ x = Some\ v \Longrightarrow$ *prog.p2s* (*some v*) $\leq \{\!|P_s\ v|\!\}, A \vdash G, \{\!|Q|\!\}$
shows *prog.p2s* (*case-option none some x*) $\leq \{\!|$*case x of None* $\Rightarrow P_n$ | *Some v* $\Rightarrow P_s\ v|\!\}, A \vdash G, \{\!|Q|\!\}$
using *assms* **by** (*fastforce intro*: *ag.pre-ag split*: *option.split*)

lemma *case-sum*:
assumes $\bigwedge v.\ x = Inl\ v \Longrightarrow$ *prog.p2s* (*left v*) $\leq \{\!|P_l\ v|\!\}, A \vdash G, \{\!|Q|\!\}$
assumes $\bigwedge v.\ x = Inr\ v \Longrightarrow$ *prog.p2s* (*right v*) $\leq \{\!|P_r\ v|\!\}, A \vdash G, \{\!|Q|\!\}$
shows *prog.p2s* (*case-sum left right x*) $\leq \{\!|$*case-sum* $P_l\ P_r\ x|\!\}, A \vdash G, \{\!|Q|\!\}$
using *assms* **by** (*fastforce intro*: *ag.pre-ag split*: *sum.split*)

lemma *case-list*:
assumes $x = [] \Longrightarrow$ *prog.p2s nil* $\leq \{\!|P_n|\!\}, A \vdash G, \{\!|Q|\!\}$
assumes $\bigwedge v\ vs.\ x = v \#\ vs \Longrightarrow$ *prog.p2s* (*cons v vs*) $\leq \{\!|P_c\ v\ vs|\!\}, A \vdash G, \{\!|Q|\!\}$
shows *prog.p2s* (*case-list nil cons x*) $\leq \{\!|$*case-list* $P_n\ P_c\ x|\!\}, A \vdash G, \{\!|Q|\!\}$
using *assms* **by** (*fastforce intro*: *ag.pre-ag split*: *list.split*)

setup ‹*Sign.parent-path*›

setup ‹*Sign.parent-path*›

13.5.2 A proof of the parallel rule using Abadi and Plotkin's composition principle

Here we show that the key rule for *Parallel* (*ag.spec.Parallel*) can be established using the *spec.ag-circular* rule (§9.2).

The following proof is complicated by the need to discard a lot of contextual information.

notepad
begin

have *imp-discharge-leL1*:
  $x' \le x \Longrightarrow x' \sqcap (x \sqcap y \longrightarrow_H z) = x' \sqcap (y \longrightarrow_H z)$ **for** $x\ x'\ y\ z$
by (*simp add*: *heyting.curry-conv heyting.discharge*(*1*))

have *LHS-rel*:
  $\{proc\ x\} \times UNIV \cup (-\{proc\ x\}) \times (A \cup (Id \cup \bigcup\ (G\ `\ (as - \{x\}))))$
  $= ((-(proc\ `\ as)) \times (A \cup (Id \cup \bigcup\ (G\ `\ (as - \{x\}))))$
    $\cup (\{proc\ x\} \times UNIV \cup proc\ `\ (as - \{x\}) \times (A \cup (Id \cup \bigcup\ (G\ `\ (as - \{x\})))))))$ **for** $A\ G\ as\ x$
by *blast*

have *rel-agents-split*:
  spec.rel $(as \times r \cup s) = spec.rel\ (as \times r \cup fst\ `\ s \times UNIV) \sqcap spec.rel\ (as \times UNIV \cup s)$
  if $fst\ `\ s \cap as = \{\}$ **for** $as\ r\ s$
using *that* **by** (*fastforce simp*: *image-iff simp flip*: *spec.rel.inf intro*: *arg-cong*[**where** *f*=*spec.rel*])

— *ag.spec.Parallel*
fix *as* :: *'a set*
fix *A* :: *'s rel*
fix *G* :: *'a* ⇒ *'s rel*
fix *P* :: *'a* ⇒ *'s pred*
fix *Q* :: *'a* ⇒ *'s pred*
fix *Ps* :: *'a* ⇒ (*sequential*, *'s*, *unit*) *spec*
assume *proc-ag*: $\bigwedge a.\ a \in as \Longrightarrow Ps\ a \le \{\!|P\ a|\!\}, A \cup (\bigcup a' \in as - \{a\}.\ G\ a') \vdash G\ a, \{\!|\lambda v.\ Q\ a|\!\}$
have *spec.Parallel as Ps* $\le \{\!|\bigsqcap a \in as.\ P\ a|\!\}, A \vdash \bigcup a \in as.\ G\ a, \{\!|\lambda v.\ \bigsqcap a \in as.\ Q\ a|\!\}$ (**is** *?lhs* ≤ *?rhs*)
proof(*cases as* = {})
  case *True* **then show** *?thesis*
    by (*simp add*: *spec.Parallel.no-agents ag.interference-le*)
next
  case *False* **then show** *?thesis*
apply −
supply *inf.bounded-iff*[*simp del*] — preserve RHS

— replace *Ps* with a/g specs. guard against empty *A*, *G*
apply (*strengthen ord-to-strengthen*(*1*)[*OF proc-ag*], *assumption*)
apply (*subst ag.reflcl-ag*)
apply (*strengthen ord-to-strengthen*(*2*)[*OF reflcl-cl.sup-cl-le*])

— Circular concurrent reasoning
unfolding *ag-def*

— Move a/g hypotheses to LHS, normalize
apply (*simp add*: *heyting ac-simps*)

— Discharge *spec.pre P*
apply (*subst inf-assoc*[*symmetric*])
apply (*subst inf-commute*)
apply (*subst inf-assoc*)
apply (*subst* (*2*) *inf-commute*)
apply (*subst spec.Parallel.inf-pre*, *assumption*)

apply (*simp add*: *ac-simps*)
— Idiom for rewriting under a quantifier, here *Parallel*
apply (*rule order.trans*)
apply (*rule inf.mono*[*OF order.refl*])
apply (*rule spec.Parallel.mono*)
apply (*subst imp-discharge-leL1*)
apply (*simp add*: *Inf-lower spec.pre.INF*; *fail*)
— Discard *spec.pre* hypothesis
apply (*rule inf-le2*)

— Move environment assumption A hypothesis under *spec.toSequential* and the *Inf*
unfolding *spec.Parallel-def*
apply (*subst inf.commute*)
apply (*subst spec.map.inf-distr*)
apply (*subst spec.invmap.rel*)
apply (*simp add*: *ac-simps flip*: *INF-inf-const1*)

— Eradicate *spec.toSequential*: move to parallel space
apply (*simp add*: *spec.map-invmap.galois spec.invmap.inf spec.invmap.post spec.invmap.rel*
flip: *spec.term.all.invmap spec.term.all.map*)

— Eradicate *spec.toConcurrent*
apply (*simp add*: *ac-simps spec.invmap.heyting spec.invmap.inf spec.invmap.rel*
spec.invmap.pre spec.invmap.post)

— Normalize the relations
apply (*simp add*: *inf-sup-distrib1 Times-Int-Times map-prod-vimage-Times ac-simps spec.rel.reflcl*
flip: *spec.rel.inf image-Int inf.assoc*)

— Discharge environment assumption A and that for agents in $-as$
apply (*subst LHS-rel*)
— Idiom for rewriting under a quantifier, here *Inf*
apply (*rule order.trans*)
apply (*rule INF-mono*[**where** B=as])
apply (*rule rev-bexI*, *assumption*)
apply (*subst* (*2*) *rel-agents-split*, *fastforce*)
apply (*subst imp-discharge-leL1*)
apply (*rule spec.rel.mono*, *fastforce simp*: *image-Un*)
apply (*rule order.refl*)
apply (*simp flip*: *sup.assoc Times-Un-distrib1*)
apply (*simp add*: *ac-simps INF-inf-const1*)

— Apply Abadi/Plotkin

— Change coordinates
apply (*subst INF-rename-bij*[**where** X=*proc* ‘ *as* **and** π=*the-inv proc*])
apply (*fastforce simp*: *bij-betw-iff-bijections*)
apply (*simp add*: *image-comp cong*: *INF-cong*)

— The circular reasoning principle only applies to the relational part as *spec.post* is not termination closed. Therefore split the goal
apply (*subst heyting.infR*)
apply (*subst INF-inf-distrib*[*symmetric*])
apply (*rule order.trans*)
apply (*rule inf-mono*[*OF order.refl*])+
apply (*rule order.trans*[*rotated*])
apply (*rule spec.ag-circular*[**where**
as=*proc* ‘ *as*

and *Ps*=λ*a*. *spec.rel* ({*a*} × (*Id* ∪ *G* (*the-inv proc a*)) ∪ *insert env* (*proc* ‘ (− {*the-inv proc a*})) × *UNIV*),
simplified spec.idle-le spec.term.closed.rel, *simplified*,
OF subsetD[*OF spec.cam.closed.antimono spec.cam.closed.rel*[*OF order.refl*]]])
apply (*clarsimp simp*: *image-iff*)
apply (*metis ComplI agent.exhaust singletonD*)
apply (*rule INFI*)
apply (*simp add*: *heyting ac-simps flip*: *spec.rel.INF INF-inf-const1*)
— Idiom for rewriting under a quantifier, here *Inf*
apply (*rule order.trans*)
apply (*rule INF-mono*[**where** *B*=*as*])
apply (*rule rev-bexI*, *assumption*)
apply (*subst heyting.discharge*)
apply (*rule spec.rel.mono-reflcl*)
apply *fastforce*
apply (*simp flip*: *spec.rel.inf*)
apply (*rule order.refl*)
apply (*simp flip*: *spec.rel.INF*)
apply (*rule spec.rel.mono*)
apply (*clarsimp simp*: *image-iff*)
apply (*metis ComplI agent.exhaust singletonD*)
apply (*simp add*: *ac-simps flip*: *spec.rel.INF*)
apply (*subst inf.assoc*[*symmetric*])
apply (*simp flip*: *spec.rel.inf*)

— Conclude guarantee *G*
apply (*rule le-infI*[*rotated*])
apply (*rule le-infI1*)
apply (*rule spec.rel.mono-reflcl*, *blast*)

— Conclude *spec.post Q*
apply (*subst* (*2*) *INF-inf-const1*[*symmetric*], *force*)
— Idiom for rewriting under a quantifier, here *Inf*
apply (*rule order.trans*)
apply (*rule INF-mono*[**where** *B*=*as*])
apply (*rule rev-bexI*, *blast*)
apply (*subst heyting.discharge*)
apply (*force intro*: *spec.rel.mono*)
apply (*rule order.refl*)
apply (*simp add*: *spec.post.Ball flip*: *INF-inf-distrib*)
done
qed

end

13.6 Specification inhabitation

setup ‹*Sign.mandatory-path inhabits.prog*›

lemma *Sup*:
assumes *prog.p2s P* −*s*, *xs*→ *P*′
assumes *P* ∈ *X*
shows *prog.p2s* (⨆ *X*) −*s*, *xs*→ *P*′
by (*auto simp*: *prog.p2s.Sup intro*: *inhabits.Sup inhabits.supL assms*)

lemma *supL*:
assumes *prog.p2s P* −*s*, *xs*→ *P*′
shows *prog.p2s* (*P* ⊔ *Q*) −*s*, *xs*→ *P*′
by (*simp add*: *prog.p2s.simps assms inhabits.supL*)

lemma *supR*:
  assumes *prog.p2s Q −s, xs→ Q′*
  shows *prog.p2s (P ⊔ Q) −s, xs→ Q′*
by (*simp add*: *prog.p2s.simps assms inhabits.supR*)

lemma *bind*:
  assumes *prog.p2s f −s, xs→ prog.p2s f′*
  shows *prog.p2s (f ⤜ g) −s, xs→ prog.p2s (f′ ⤜ g)*
by (*simp add*: *prog.p2s.simps inhabits.spec.bind assms*)

lemma *return*:
  shows *prog.p2s (prog.return v) −s, []→ spec.return v*
by (*metis prog.p2s.return inhabits.pre inhabits.tau*[*OF spec.idle.interference.cl-le*]
        spec.interference.expansive)

lemma *action-step*:
  fixes *F* :: (′*v* × ′*s* × ′*s*) *set*
  assumes (*v*, *s*, *s′*) ∈ *F*
  shows *prog.p2s (prog.action F) −s, [(self, s′)]→ prog.p2s (prog.return v)*
apply (*simp only*: *prog.p2s.action prog.p2s.return spec.interference.cl.action spec.interference.cl.return*)
apply (*rule inhabits.pre*[*OF - order.refl*])
 apply (*rule inhabits.spec.bind′*[*OF inhabits.spec.rel.term*])
 apply (*simp add*: *spec.bind.returnL spec.idle-le*)
 apply (*rule inhabits.spec.bind′*[*OF inhabits.spec.action.step*])
  using *assms* **apply** *force*
 apply (*simp add*: *spec.bind.returnL spec.idle-le*)
 apply (*rule inhabits.tau*)
 apply (*simp add*: *spec.idle-le*)
apply *simp*
done

lemma *action-stutter*:
  fixes *F* :: (′*v* × ′*s* × ′*s*) *set*
  assumes (*v*, *s*, *s*) ∈ *F*
  shows *prog.p2s (prog.action F) −s, []→ prog.p2s (prog.return v)*
apply (*simp only*: *prog.p2s.action prog.p2s.return spec.interference.cl.action spec.interference.cl.return*)
apply (*rule inhabits.pre*[*OF - order.refl*])
 apply (*rule inhabits.spec.bind′*[*OF inhabits.spec.rel.term*])
 apply (*simp add*: *spec.bind.returnL spec.idle-le*)
 apply (*rule inhabits.spec.bind′*[*OF inhabits.spec.action.stutter*])
  using *assms* **apply** *force*
 apply (*simp add*: *spec.bind.returnL spec.idle-le*)
 apply (*rule inhabits.tau*)
 apply (*simp add*: *spec.idle-le*)
apply *simp*
done

lemma *parallelL*:
  assumes *prog.p2s P −s, xs→ prog.p2s P′*
  shows *prog.p2s (P ∥ Q) −s, xs→ prog.p2s (P′ ∥ Q)*
by (*simp add*: *prog.p2s.simps inhabits.spec.parallelL assms prog.p2s.interference-wind-bind*)

lemma *parallelR*:
  assumes *prog.p2s Q −s, xs→ prog.p2s Q′*
  shows *prog.p2s (P ∥ Q) −s, xs→ prog.p2s (P ∥ Q′)*
by (*simp add*: *prog.p2s.simps inhabits.spec.parallelR assms prog.p2s.interference-wind-bind*)

setup ‹*Sign.parent-path*›

14 More combinators

Extra combinators:

- *prog.select* shows how we can handle arbitrary choice
- *prog.while* combinator expresses all tail-recursive computations. Its condition is a pure value.

setup ‹*Sign.mandatory-path prog*›

definition *select* :: *'v set* ⇒ (*'s*, *'v*) *prog* **where**
  select X = (⨆ *x*∈*X*. *prog.return x*)

context
  notes [[*function-internals*]]
begin

partial-function (*lfp*) *while* :: (*'k* ⇒ (*'s*, *'k* + *'v*) *prog*) ⇒ *'k* ⇒ (*'s*, *'v*) *prog* **where**
  while c k = *c k* ⤜ (λ*rv*. *case rv of Inl k'* ⇒ *while c k'* | *Inr v* ⇒ *prog.return v*)

end

abbreviation *loop* :: (*'s*, *unit*) *prog* ⇒ (*'s*, *'w*) *prog* **where**
  loop P ≡ *prog.while* (λ(). *P* ≫ *prog.return* (*Inl* ())) ()

abbreviation *guardM* :: *bool* ⇒ (*'s*, *unit*) *prog* **where**
  guardM b ≡ *if b then* ⊥ *else prog.return* ()

abbreviation *unlessM* :: *bool* ⇒ (*'s*, *unit*) *prog* ⇒ (*'s*, *unit*) *prog* **where**
  unlessM b c ≡ *if b then prog.return* () *else c*

abbreviation *whenM* :: *bool* ⇒ (*'s*, *unit*) *prog* ⇒ (*'s*, *unit*) *prog* **where**
  whenM b c ≡ *if b then c else prog.return* ()

definition *app* :: (*'a* ⇒ (*'s*, *unit*) *prog*) ⇒ *'a list* ⇒ (*'s*, *unit*) *prog* **where** — Haskell's *mapM-*
  app f xs = *foldr* (λ*x m*. *f x* ≫ *m*) *xs* (*prog.return* ())

definition *set-app* :: (*'a* ⇒ (*'s*, *unit*) *prog*) ⇒ *'a set* ⇒ (*'s*, *unit*) *prog* **where**
  set-app f =
    prog.while (λ*X*. *if X* = {} *then prog.return* (*Inr* ())
                          else prog.select X ⤜ (λ*x*. *f x* ≫ *prog.return* (*Inl* (*X* − {*x*}))))

primrec *foldM* :: (*'b* ⇒ *'a* ⇒ (*'s*, *'b*) *prog*) ⇒ *'b* ⇒ *'a list* ⇒ (*'s*, *'b*) *prog* **where**
  foldM f b [] = *prog.return b*
| *foldM f b* (*x* # *xs*) = *do* {
    b' ← *f b x*;
    foldM f b' xs
  }

primrec *fold-mapM* :: (*'a* ⇒ (*'s*, *'b*) *prog*) ⇒ *'a list* ⇒ (*'s*, *'b list*) *prog* **where**
  fold-mapM f [] = *prog.return* []
| *fold-mapM f* (*x* # *xs*) = *do* {
    y ← *f x*;
    ys ← *fold-mapM f xs*;
    prog.return (*y* # *ys*)
  }

setup ‹*Sign.mandatory-path select*›

lemma *empty*:
  shows *prog.select* {} = ⊥
by (*simp add*: *prog.select-def*)

lemma *singleton*:
  shows *prog.select* {*x*} = *prog.return x*
by (*simp add*: *prog.select-def*)

lemma *monotone*:
  shows *mono prog.select*
by (*simp add*: *monoI prog.select-def SUP-subset-mono*)

lemmas *strengthen*[*strg*] = *st-monotone*[*OF prog.select.monotone*]
lemmas *mono* = *monotoneD*[*OF prog.select.monotone*, *of P Q* **for** *P Q*]
lemmas *mono2mono*[*cont-intro*, *partial-function-mono*] = *monotone2monotone*[*OF prog.select.monotone*, *simplified*, *of orda P* **for** *orda P*]

lemma *Sup*:
  shows *prog.select* (⋃ *X*) = (⨆*x*∈*X*. *prog.select x*)
by (*simp add*: *prog.select-def flip*: *SUP-UNION*)

lemma *mcont*:
  shows *mcont* ⋃ (⊆) *Sup* (≤) *prog.select*
by (*simp add*: *mcontI contI prog.select.monotone prog.select.Sup*)

lemmas *mcont2mcont*[*cont-intro*] = *mcont2mcont*[*OF prog.select.mcont*, *of supa orda P* **for** *supa orda P*]

setup ‹*Sign.parent-path*›

setup ‹*Sign.mandatory-path return*›

lemma *select-le*:
  assumes *x* ∈ *X*
  shows *prog.return x* ≤ *prog.select X*
by (*simp add*: *assms prog.select-def SUP-upper*)

setup ‹*Sign.parent-path*›

setup ‹*Sign.mandatory-path bind*›

lemma *selectL*:
  shows *prog.select X* ⤜ *g* = (⨆*x*∈*X*. *g x*)
by (*simp add*: *prog.select-def prog.bind.SUPL prog.bind.returnL*)

setup ‹*Sign.parent-path*›

setup ‹*Sign.mandatory-path while*›

lemma *bot*:
  shows *prog.while* ⊥ = ⊥
by (*simp add*: *fun-eq-iff prog.while.simps prog.bind.botL*)

lemma *monotone*: — could hope to prove this with a *strengthen* rule for *lfp.fixp-fun*
  shows *mono* (λ*P*. *prog.while P s*)
by (*rule monoI*)

(*induct arbitrary*: *s rule*: *prog.while.fixp-induct*; *simp add*: *prog.bind.mono le-funD split*: *sum.split*)

lemmas *strengthen*[*strg*] = *st-monotone*[*OF prog.while.monotone*]
lemmas *mono'* = *monotoneD*[*OF prog.while.monotone*, *of P Q* **for** *P Q*] — compare with *prog.while.mono*
lemmas *mono2mono*[*cont-intro*, *partial-function-mono*] = *monotone2monotone*[*OF prog.while.monotone*, *simplified*, *of orda P* **for** *orda P*]

lemma *Sup-le*:
shows ($\bigsqcup P{\in}X.$ *prog.while P s*) $\leq$ *prog.while* ($\bigsqcup X$) *s*
by (*simp add*: *SUP-le-iff SupI prog.while.mono'*)

lemma *Inf-le*:
shows *prog.while* ($\bigsqcap X$) *s* $\leq$ ($\bigsqcap P{\in}X.$ *prog.while P s*)
by (*simp add*: *le-INF-iff Inf-lower prog.while.mono'*)

lemma *True-skip-eq-bot*:
shows *prog.while* ⟨*prog.return* (*Inl x*)⟩ *s* = ⊥
by (*induct arbitrary*: *s rule*: *prog.while.fixp-induct*) (*simp-all add*: *prog.bind.returnL*)

lemma *Inr-eq-return*:
shows *prog.while* ⟨*prog.return* (*Inr v*)⟩ *s* = *prog.return v*
by (*subst prog.while.simps*) (*simp add*: *prog.bind.returnL*)

lemma *kleene-star*:
shows *prog.kleene.star P*
= *prog.while* (λ-. (*P* ≫ *prog.return* (*Inl* ())) ⊔ *prog.return* (*Inr* ())) () (**is** *?lhs* = *?rhs*)
proof(*rule antisym*)
show *?lhs* $\leq$ *?rhs*
proof(*induct rule*: *prog.kleene.star.fixp-induct*[*case-names adm bot step*])
case (*step P*) **then show** *?case*
by (*subst prog.while.simps*) (*simp add*: *prog.bind.supL prog.bind.bind prog.bind.mono sup.coboundedI1 prog.bind.returnL*)
qed *simp-all*
show *?rhs* $\leq$ *?lhs*
proof(*induct rule*: *prog.while.fixp-induct*[*case-names adm bot step*])
case (*step k*) **then show** *?case*
by (*subst prog.kleene.star.simps*) (*simp add*: *prog.bind.supL prog.bind.bind prog.bind.mono prog.bind.returnL le-supI1*)
qed *simp-all*
qed

lemma *invmap-le*:
fixes *sf* :: *'s* ⇒ *'t*
fixes *vf* :: *'v* ⇒ *'w*
shows *prog.while* (λ*k*. *prog.invmap sf* (*map-sum id vf*) (*c k*)) *k*
$\leq$ *prog.invmap sf vf* (*prog.while c k*) (**is** *?lhs prog.while k* $\leq$ *?rhs k*)
proof(*rule spec*[**where** *x*=*k*],
induct rule: *prog.while.fixp-induct*[**where** *P*=λ*R*. ∀ *k*. *?lhs R k* $\leq$ *?rhs k*, *case-names adm bot step*])
case (*step k*) **show** *?case*
apply (*subst prog.while.simps*)
apply (*strengthen ord-to-strengthen*(*1*)[*OF step*[*rule-format*]])
apply (*auto intro*!: *SUPE prog.bind.mono*[*OF order.refl*]
split: *sum.splits*
simp: *prog.invmap.bind prog.invmap.return*
prog.invmap.split-vinvmap[**where** *sf*=*sf* **and** *vf*=*map-sum id vf*]
prog.bind.bind prog.bind.return prog.bind.SUPL
SUP-upper
order.trans[*OF - prog.bind.mono*[*OF prog.return.rel-le order.refl*]])

done
qed (*simp-all add*: *prog.invmap.bot*)

setup ‹*Sign.parent-path*›

setup ‹*Sign.mandatory-path loop*›

lemma *bindL*:
fixes *P* :: ('*s*, *unit*) *prog*
fixes *Q* :: ('*s*, '*w*) *prog*
shows *prog.loop P* ≫ *Q* = *prog.loop P* (**is** *?lhs* = *?rhs*)
proof(*rule antisym*)
show *?lhs* ≤ *?rhs*
by (*rule prog.while.fixp-induct*[**where** *P*=λ*R*. *R* (λ(). *P* ≫ *prog.return* (*Inl* ())) () ≫ *Q* ≤ *?rhs*]; *simp add*: *prog.bind.botL*)
(*subst prog.while.simps*; *simp add*: *prog.bind.bind prog.bind.mono lambda-unit-futzery prog.bind.returnL*)
show *?rhs* ≤ *?lhs*
by (*rule prog.while.fixp-induct*[**where** *P*=λ*R*. *R* (λ(). *P* ≫ *prog.return* (*Inl* ())) () ≤ *?lhs*]; *simp*)
(*subst prog.while.simps*; *simp add*: *prog.bind.bind prog.bind.mono lambda-unit-futzery prog.bind.returnL*)
qed

lemma *parallel-le*:
shows *prog.loop P* ≤ *lfp* (λ*R*. *P* ∥ *R*)
proof(*induct rule*: *prog.while.fixp-induct*[*case-names adm bot step*])
case (*step k*) **show** *?case*
apply (*subst lfp-unfold*, *simp*)
apply (*strengthen ord-to-strengthen*[*OF prog.bind.parallel-le*])
apply (*simp add*: *prog.bind.bind prog.bind.mono prog.bind.returnL step*)
done
qed *simp-all*

setup ‹*Sign.parent-path*›

setup ‹*Sign.mandatory-path foldM*›

lemma *append*:
shows *prog.foldM f b* (*xs* @ *ys*) = *prog.foldM f b xs* ≫= (λ*b*'. *prog.foldM f b*' *ys*)
by (*induct xs arbitrary*: *b*) (*simp-all add*: *prog.bind.returnL prog.bind.bind*)

setup ‹*Sign.parent-path*›

lemma *foldM-alt-def*:
shows *prog.foldM f b xs* = *foldr* (λ*x m*. *prog.bind m* (λ*b*. *f b x*)) (*rev xs*) (*prog.return b*)
by (*induct xs arbitrary*: *b rule*: *rev-induct*) (*simp-all add*: *prog.foldM.append prog.bind.returnR*)

setup ‹*Sign.mandatory-path fold-mapM*›

lemma *bot*:
shows *prog.fold-mapM* ⊥ = (λ*xs*. *case xs of* [] ⇒ *prog.return* [] | - ⇒ ⊥)
by (*simp add*: *fun-eq-iff prog.bind.botL split*: *list.split*)

lemma *append*:
shows *prog.fold-mapM f* (*xs* @ *ys*)
= *prog.fold-mapM f xs* ≫= (λ*xs*. *prog.fold-mapM f ys* ≫= (λ*ys*. *prog.return* (*xs* @ *ys*)))
by (*induct xs*) (*simp-all add*: *prog.bind.bind prog.bind.returnL prog.bind.returnR*)

setup ‹*Sign.parent-path*›

setup ‹*Sign.mandatory-path app*›

lemma *bot*:
  shows *prog.app* ⊥ = (λ*xs*. *case xs of* [] ⇒ *prog.return* () | - ⇒ ⊥)
    and *prog.app* (λ-. ⊥) = (λ*xs*. *case xs of* [] ⇒ *prog.return* () | - ⇒ ⊥)
by (*simp-all add*: *fun-eq-iff prog.app-def prog.bind.botL split*: *list.split*)

lemma *Nil*:
  shows *prog.app f* [] = *prog.return* ()
by (*simp add*: *prog.app-def*)

lemma *Cons*:
  shows *prog.app f* (*x* # *xs*) = *f x* ≫ *prog.app f xs*
by (*simp add*: *prog.app-def*)

lemmas *simps* =
  prog.app.bot
  prog.app.Nil
  prog.app.Cons

lemma *append*:
  shows *prog.app f* (*xs* @ *ys*) = *prog.app f xs* ≫ *prog.app f ys*
by (*induct xs arbitrary*: *ys*) (*simp-all add*: *prog.app.simps prog.bind.returnL prog.bind.bind*)

lemma *monotone*:
  shows *mono* (λ*f*. *prog.app f xs*)
by (*induct xs*) (*simp-all add*: *prog.app.simps le-fun-def monotone-on-def prog.bind.mono*)

lemmas *strengthen*[*strg*] = *st-monotone*[*OF prog.app.monotone*]
lemmas *mono* = *monotoneD*[*OF prog.app.monotone*]
lemmas *mono2mono*[*cont-intro*, *partial-function-mono*] = *monotone2monotone*[*OF prog.app.monotone*, *simplified*, *of orda P* **for** *orda P*]

lemma *Sup-le*:
  shows (⨆*f*∈*X*. *prog.app f xs*) ≤ *prog.app* (⨆ *X*) *xs*
by (*simp add*: *SUP-le-iff SupI prog.app.mono*)

setup ‹*Sign.parent-path*›

setup ‹*Sign.mandatory-path invmap*›

lemma *app*:
  fixes *sf* :: *'s* ⇒ *'t*
  fixes *vf* :: *'v* ⇒ *unit*
  shows *prog.invmap sf vf* (*prog.app f xs*)
      = *prog.app* (λ*x*. *prog.sinvmap sf* (*f x*)) *xs* ≫ *prog.invmap sf vf* (*prog.return* ())
by (*induct xs*)
   (*simp-all add*: *prog.app.simps prog.bind.return prog.invmap.bind prog.bind.bind id-def*)

setup ‹*Sign.parent-path*›

setup ‹*Sign.mandatory-path sinvmap*›

lemma *app-le*:
  fixes *sf* :: *'s* ⇒ *'t*
  fixes *vf* :: *'v* ⇒ *unit*
  shows *prog.app* (λ*x*. *prog.sinvmap sf* (*f x*)) *xs* ≤ *prog.sinvmap sf* (*prog.app f xs*)
by (*simp add*: *prog.invmap.app prog.invmap.return prog.bind.return*

order.trans[*OF - prog.bind.mono*[*OF order.refl prog.return.rel-le*]])

setup ‹*Sign.parent-path*›

setup ‹*Sign.mandatory-path set-app*›

lemma *bot*:
  shows $X \neq \{\} \Longrightarrow$ *prog.set-app* $\bot$ $X = \bot$
    and $X \neq \{\} \Longrightarrow$ *prog.set-app* (λ-. $\bot$) $X = \bot$
by (*simp-all add*: *prog.set-app-def prog.while.simps prog.bind.bind prog.bind.botL prog.bind.selectL*)

lemma *empty*:
  shows *prog.set-app f* {} = *prog.return* ()
by (*simp add*: *prog.set-app-def prog.while.simps prog.bind.returnL*)

lemma *not-empty*:
  assumes $X \neq \{\}$
  shows *prog.set-app f X* = *prog.select X* $\gg\!=$ (λx. *f x* $\gg$ *prog.set-app f* ($X - \{x\}$))
using *assms* **by** (*simp add*: *prog.set-app-def prog.while.simps prog.bind.returnL prog.bind.bind*)

lemmas *simps* =
  prog.set-app.bot
  prog.set-app.empty
  prog.set-app.not-empty

setup ‹*Sign.parent-path*›

setup ‹*Sign.mandatory-path app*›

lemma *set-app-le*:
  assumes *X* = *set xs*
  assumes *distinct xs*
  shows *prog.app f xs* $\leq$ *prog.set-app f X*
using *assms*
proof(*induct xs arbitrary*: *X*)
  case (*Cons x xs*) **then show** *?case*
    apply (*simp add*: *prog.set-app.simps prog.app.simps*)
    apply (*strengthen ord-to-strengthen*(*2*)[*OF prog.return.select-le*[*of x*]], *blast*)
    apply (*simp add*: *prog.bind.returnL prog.bind.mono*)
    done
qed (*simp add*: *prog.app.simps prog.set-app.simps*)

setup ‹*Sign.parent-path*›

lemma *set-app-alt-def*:
  assumes *finite X*
  shows *prog.set-app f X* = ($\bigsqcup$ *xs*$\in${*ys. set ys* = *X* $\wedge$ *distinct ys*}. *prog.app f xs*) (**is** *?lhs* = *?rhs*)
proof(*rule antisym*)
  from *assms* **show** *?lhs* $\leq$ *?rhs*
  proof(*induct rule*: *finite-remove-induct*)
    case (*remove X*)
    from ‹*finite X*› ‹*X* $\neq$ {}› **have** $*$: {*ys. set ys* = *X* $-$ {*x*} $\wedge$ *distinct ys*} $\neq$ {} **for** *x*
      by (*simp add*: *finite-distinct-list*)
    from ‹*X* $\neq$ {}› **show** *?case*
      apply (*clarsimp simp*: *prog.set-app.simps prog.bind.selectL*)
      apply (*strengthen ord-to-strengthen*[*OF remove.hyps*(*4*)], *blast*)
      apply (*fastforce simp*: *prog.app.simps prog.bind.SUPR-not-empty*[*OF* $*$] *Sup-le-iff*
                  intro: *rev-SUPI*[**where** *x*=*x* # *xs* **for** *x xs*])

```
    done
  qed (simp add: prog.set-app.simps prog.app.simps)
  show ?rhs ≤ ?lhs
    by (simp add: Sup-le-iff prog.app.set-app-le)
qed

setup ‹Sign.parent-path›

setup ‹Sign.parent-path›

setup ‹Sign.mandatory-path ag.prog›

lemma select-sp:
  assumes ⋀s x. ⟦P s; x ∈ X⟧ ⟹ Q x s
  assumes ⋀v. stable A (P ∧ Q v)
  shows prog.p2s (prog.select X) ≤ ⦃P⦄, A ⊢ G, ⦃λv. P ∧ Q v⦄
by (clarsimp simp: prog.select-def prog.p2s.Sup spec.interference.cl.bot ag.spec.term.none-inteference)
   (rule ag.pre[OF ag.prog.return[OF assms(2)]]; blast intro: assms(1))

lemma while:
  fixes c :: 'k ⇒ ('s, 'k + 'v) prog
  assumes c: ⋀k. prog.p2s (c k) ≤ ⦃P k⦄, A ⊢ G, ⦃case-sum I Q⦄
  assumes IP: ⋀s v. I v s ⟹ P v s
  assumes sQ: ⋀v. stable A (Q v)
  shows prog.p2s (prog.while c k) ≤ ⦃I k⦄, A ⊢ G, ⦃Q⦄
proof −
  have prog.p2s (prog.while c k) ≤ ⦃P k⦄, A ⊢ G, ⦃Q⦄
  proof(induct arbitrary: k rule: prog.while.fixp-induct[case-names adm bot step])
    case (step k) show ?case
      apply (rule ag.prog.bind[OF - c])
      apply (rule ag.pre-pre[OF ag.prog.case-sum[OF step ag.prog.return[OF sQ]]])
      apply (simp add: IP split: sum.splits)
      done
  qed (simp-all add: ag.prog.galois)
  then show ?thesis
    by (meson IP ag.pre-pre)
qed

lemma app:
  fixes xs :: 'a list
  fixes f :: 'a ⇒ ('s, unit) prog
  fixes P :: 'a list ⇒ 's pred
  assumes ⋀x ys zs. xs = ys @ x # zs ⟹ prog.p2s (f x) ≤ ⦃P ys⦄, A ⊢ G, ⦃λ-. P (ys @ [x])⦄
  assumes ⋀ys. prefix ys xs ⟹ stable A (P ys)
  shows prog.p2s (prog.app f xs) ≤ ⦃P []⦄, A ⊢ G, ⦃λ-. P xs⦄
using assms
by (induct xs rule: rev-induct;
    fastforce intro: ag.prog.bind ag.prog.return
              simp: prog.app.append prog.bind.returnR prog.app.simps)

lemma app-set:
  fixes X :: 'a set
  fixes f :: 'a ⇒ ('s, unit) prog
  fixes P :: 'a set ⇒ 's pred
  assumes ⋀Y x. ⟦Y ⊆ X; x ∈ X − Y⟧ ⟹ prog.p2s (f x) ≤ ⦃P Y⦄, A ⊢ G, ⦃λ-. P (insert x Y)⦄
  assumes ⋀Y. Y ⊆ X ⟹ Stability.stable A (P Y)
  shows prog.p2s (prog.set-app f X) ≤ ⦃P {}⦄, A ⊢ G, ⦃λ-. P X⦄
proof −
```

have $*$: $X - (Y - \{x\}) = insert\ x\ (X - Y)$ **if** $Y \subseteq X$ **and** $x \in Y$ **for** x **and** $X\ Y :: 'a\ set$
using *that* **by** *blast*
show *?thesis*
unfolding *prog.set-app-def*
apply (*rule ag.prog.while*[**where** $I{=}\lambda Y\ s.\ Y \subseteq X \wedge P\ (X - Y)\ s$ **and** $Q{=}\langle P\ X\rangle$ **and** $k{=}X$, *simplified*])
apply (*rename-tac k*)
apply (*rule ag.prog.if*)
apply (*rule ag.prog.return*)
apply (*simp add*: *assms*; *fail*)
apply (*rule-tac* $P{=}\lambda s.\ k \subseteq X \wedge P\ (X - k)\ s$ **in** *ag.prog.bind*[*rotated*])
apply (*rule-tac* $Q{=}\lambda x\ s.\ x \in k$ **in** *ag.prog.select-sp*, *assumption*)
apply (*simp add*: *assms*(*2*) *stable.conjI stable.const*; *fail*)
apply (*intro ag.gen-asm*)
apply (*rule ag.prog.bind*[*rotated*])
apply (*rule assms*(*1*); *force*)
apply (*rule ag.pre-pre*[*OF ag.prog.return*])
apply (*simp add*: *assms*(*2*) *stable.conjI stable.const*; *fail*)
using $*$ **apply** *fastforce*
apply *force*
apply (*simp add*: *assms*(*2*))
done
qed

lemma *foldM*:
fixes $xs :: 'a\ list$
fixes $f :: 'b \Rightarrow 'a \Rightarrow ('s, 'b)\ prog$
fixes $I :: 'b \Rightarrow 'a \Rightarrow 's\ pred$
fixes $P :: 'b \Rightarrow 's\ pred$
assumes f: $\bigwedge b\ x.\ x \in set\ xs \Longrightarrow prog.p2s\ (f\ b\ x) \leq \{\!|I\ b\ x|\!\}, A \vdash G, \{\!|P|\!\}$
assumes P: $\bigwedge b\ x\ s.\ [\![P\ b\ s; x \in set\ xs]\!] \Longrightarrow I\ b\ x\ s$
assumes sP: $\bigwedge b.\ stable\ A\ (P\ b)$
shows $prog.p2s\ (prog.foldM\ f\ b\ xs) \leq \{\!|P\ b|\!\}, A \vdash G, \{\!|P|\!\}$
using $f\ P$ **by** (*induct xs arbitrary*: *b*) (*fastforce intro*!: *ag.prog.bind intro*: *ag.pre-pre ag.prog.return*[*OF sP*])+

setup ‹*Sign.parent-path*›

15 Structural local state

15.1 *spec.local*

We develop a few combinators for structural local state. The goal is to encapsulate a local state of type *'ls* in a process (*'a agent*, *'ls* × *'s*, *'v*) *spec*. Applying *spec.smap snd* yields a process of type (*'a agent*, *'s*, *'v*) *spec*. We also constrain environment steps to not affect *'ls*, yielding a plausible data refinement rule (see §15.6.1).

abbreviation (*input*) *localize1* :: $('b \Rightarrow 's \Rightarrow 'a) \Rightarrow 'b \Rightarrow 'ls \times 's \Rightarrow 'a$ **where**
$localize1\ f\ b\ s \equiv f\ b\ (snd\ s)$

setup ‹*Sign.mandatory-path spec*›

setup ‹*Sign.mandatory-path local*›

definition *qrm* :: (*'a agent*, *'ls* × *'s*) *steps* **where** — cf *ag.assm*
$qrm = range\ proc \times UNIV \cup \{env\} \times (Id \times_R UNIV)$

abbreviation (*input*) *interference* ≡ *spec.rel spec.local.qrm*

setup ‹*Sign.parent-path*›

definition *local* :: (*'a agent*, *'ls* × *'s*, *'v*) *spec* ⇒ (*'a agent*, *'s*, *'v*) *spec* **where**
local P = *spec.smap snd* (*spec.local.interference* ⊓ *P*)

setup ‹*Sign.mandatory-path singleton*›

lemma *local-le-conv*:
shows ⦇*σ*⦈ ≤ *spec.local P*
⟷ (∃*σ'*. ⦇*σ'*⦈ ≤ *P*
∧ *trace.steps σ'* ⊆ *spec.local.qrm*
∧ ⦇*σ*⦈ ≤ ⦇*trace.map id snd id σ'*⦈)
by (*simp add*: *spec.local-def spec.singleton.le-conv ac-simps*)

setup ‹*Sign.parent-path*›

setup ‹*Sign.mandatory-path idle*›

lemma *local-le*[*spec.idle-le*]: — Converse does not hold
assumes *spec.idle* ≤ *P*
shows *spec.idle* ≤ *spec.local P*
by (*simp add*: *spec.local-def assms spec.idle-le*)

setup ‹*Sign.parent-path*›

setup ‹*Sign.mandatory-path local*›

setup ‹*Sign.mandatory-path qrm*›

lemma *refl*:
shows *refl* (*spec.local.qrm* `` {*a*})
by (*simp add*: *spec.local.qrm-def refl-onI*)

lemma *member*:
shows (*proc a*, *s*, *s'*) ∈ *spec.local.qrm*
and (*env*, *s*, *s'*) ∈ *spec.local.qrm* ⟷ *fst s* = *fst s'*
by (*auto simp*: *spec.local.qrm-def*)

lemma *inter*:
shows *UNIV* × *Id* ∩ *spec.local.qrm* = *UNIV* × *Id*
and *spec.local.qrm* ∩ *UNIV* × *Id* = *UNIV* × *Id*
and *spec.local.qrm* ∩ {*self*} × *Id* = {*self*} × *Id*
and *spec.local.qrm* ∩ {*env*} × *UNIV* = {*env*} × (*Id* ×$_R$ *UNIV*)
and *spec.local.qrm* ∩ {*env*} × (*UNIV* ×$_R$ *Id*) = {*env*} × *Id*
and *spec.local.qrm* ∩ *A* × (*Id* ×$_R$ *r*) = *A* × (*Id* ×$_R$ *r*)
by (*auto simp*: *spec.local.qrm-def*)

lemmas *simps*[*simp*] =
spec.local.qrm.refl
spec.local.qrm.member
spec.local.qrm.inter

setup ‹*Sign.parent-path*›

setup ‹*Sign.mandatory-path interference*›

lemma *smap-snd*:
shows *spec.smap snd spec.local.interference* = ⊤
by (*subst spec.map.rel*)
(*auto simp*: *spec.local.qrm-def spec.rel.UNIV*

image-Un map-prod-image-Times map-prod-image-relprod map-prod-surj
simp flip: *Sigma-Un-distrib1*)

setup ‹*Sign.parent-path*›

lemma *inf-interference*:
shows *spec.local P* = *spec.local* (*P* ⊓ *spec.local.interference*)
by (*simp add*: *spec.local-def ac-simps*)

lemma *bot*:
shows *spec.local* ⊥ = ⊥
by (*simp add*: *spec.local-def spec.map.bot*)

lemma *top*:
shows *spec.local* ⊤ = ⊤
by (*simp add*: *spec.local-def spec.local.interference.smap-snd*)

lemma *monotone*:
shows *mono spec.local*
proof(*rule monotoneI*)
show *spec.local P* ≤ *spec.local P'* **if** *P* ≤ *P'* **for** *P P'* :: ('*a agent*, '*s* × '*ls*, '*v*) *spec*
unfolding *spec.local-def* **by** (*strengthen ord-to-strengthen*(*1*)[*OF* ‹*P* ≤ *P'*›]) *simp*
qed

lemmas *strengthen*[*strg*] = *st-monotone*[*OF spec.local.monotone*]
lemmas *mono* = *monotoneD*[*OF spec.local.monotone*]
lemmas *mono2mono*[*cont-intro*, *partial-function-mono*]
= *monotone2monotone*[*OF spec.local.monotone*, *simplified*, *of orda P* **for** *orda P*]

lemma *Sup*:
shows *spec.local* (⨆ *X*) = (⨆ *x*∈*X*. *spec.local x*)
by (*simp add*: *spec.local-def inf-Sup spec.map.Sup image-image*)

lemmas *sup* = *spec.local.Sup*[**where** *X*={*X*, *Y*} **for** *X Y*, *simplified*]

lemma *mcont2mcont*[*cont-intro*]:
assumes *mcont luba orda Sup* (≤) *P*
shows *mcont luba orda Sup* (≤) (λ*x*. *spec.local* (*P x*))
by (*simp add*: *spec.local-def assms*)

lemma *idle*:
shows *spec.local spec.idle* = *spec.idle*
by (*simp add*: *spec.local-def inf.absorb2*[*OF spec.idle.rel-le*] *spec.map.idle*)

lemma *action*:
fixes *F* :: ('*v* × '*a agent* × ('*ls* × '*s*)) × ('*ls* × '*s*)) *set*
shows *spec.local* (*spec.action F*)
= *spec.action* (*map-prod id* (*map-prod id* (*map-prod snd snd*)) ‘
(*F* ∩ *UNIV* × *spec.local.qrm*))
by (*simp add*: *spec.local-def spec.action.inf-rel spec.map.surj-sf-action*)

lemma *return*:
shows *spec.local* (*spec.return v*) = *spec.return v*
by (*simp add*: *spec.return-def spec.local.action*
Times-Int-Times map-prod-image-Times map-prod-snd-snd-image-Id)

lemma *bind-le*: — Converse does not hold
shows *spec.local* (*f* ⤜ *g*) ≤ *spec.local f* ⤜ (λ*v*. *spec.local* (*g v*))

by (*simp add*: *spec.local-def spec.bind.inf-rel spec.map.bind-le*)

lemma *interference*:
  shows *spec.local* (*spec.rel* ({*env*} × *UNIV*)) = *spec.rel* ({*env*} × *UNIV*)
by (*simp add*: *spec.local-def spec.map.rel map-prod-image-Times map-prod-image-relprod*
      flip: *spec.rel.inf*)

setup ‹*Sign.parent-path*›

setup ‹*Sign.mandatory-path map*›

lemma *local-le*:
  shows *spec.map id sf vf* (*spec.local P*) ≤ *spec.local* (*spec.map id* (*map-prod id sf*) *vf P*)
by (*fastforce intro*: *spec.map.mono inf.mono spec.rel.mono*
          simp: *spec.local-def spec.map.comp spec.map.inf-rel spec.local.qrm-def*)

setup ‹*Sign.parent-path*›

setup ‹*Sign.mandatory-path vmap*›

lemma *local*:
  shows *spec.vmap vf* (*spec.local P*) = *spec.local* (*spec.vmap vf P*)
by (*simp add*: *spec.local-def spec.map.comp spec.map.inf-rel spec.rel.reflcl*(*1*)[**where** *A*=*UNIV*])

setup ‹*Sign.parent-path*›

setup ‹*Sign.mandatory-path invmap*›

lemma *smap-snd*:
  fixes *P* :: ('*a*, '*ls* × '*t*, '*w*) *spec*
  fixes *sf* :: '*s* ⇒ '*t*
  fixes *vf* :: '*v* ⇒ '*w*
  shows *spec.invmap id sf vf* (*spec.smap snd P*)
      = *spec.smap snd* (*spec.invmap id* (*map-prod id sf*) *vf P*) (**is** *?lhs* = *?rhs*)
proof(*rule spec.singleton.antisym*)
  have *smap-snd-aux*:
    ∃ *zs*. *trace.natural'* (*ls*, *sf s*) *xs* = *trace.natural'* (*ls*, *sf s*) (*map* (*map-prod id* (*map-prod id sf*)) *zs*)
       ∧ *trace.natural' s* (*map* (*map-prod id snd*) *zs*) = *trace.natural' s ys* (**is** ∃ *zs*. *?P ls s ys zs*)
    if *trace.natural'* (*sf s*) (*map* (*map-prod id sf*) *ys*) = *trace.natural'* (*sf s*) (*map* (*map-prod id snd*) *xs*)
   for *ls* **and** *s* :: '*s* **and** *sf* :: '*s* ⇒ '*t* **and** *xs* :: ('*a* × '*ls* × '*t*) *list* **and** *ys* :: ('*a* × '*s*) *list*
    using *that*
  proof(*induct xs arbitrary*: *ls s ys*)
    case (*Nil ls s ys*) **then show** *?case*
      by (*fastforce intro*: *exI*[**where** *x*=*map* (*map-prod id* (*Pair ls*)) *ys*]
                  simp: *comp-def trace.natural'.eq-Nil-conv*)
  next
    case (*Cons x xs ls s ys*) **show** *?case*
    proof(*cases snd* (*snd x*) = *sf s*)
      case *True* **with** *Cons.prems* **show** *?thesis*
        by (*cases x*) (*fastforce dest*: *Cons.hyps*[**where** *ls*=*fst* (*snd x*)]
                        intro: *exI*[**where** *x*=(*fst x*, *fst* (*snd x*), *s*) # *zs* **for** *zs*]
                    simp flip: *id-def*)
    next
      case *False*
      with *Cons.prems* **obtain** *a* s_x *us* s' *zs*
        where *x* = (*a*, s_x, *sf* s')
        and *sf* s' ≠ *sf s*
        and *snd* ‘ *map-prod id sf* ‘ *set us* ⊆ {*sf s*}

```
      and ys = us @ (a, s') # zs
      and trace.natural' (sf s') (map (map-prod id sf) zs) = trace.natural' (sf s') (map (map-prod id snd) xs)
      by (cases x) (clarsimp simp: trace.natural'.eq-Cons-conv map-eq-append-conv simp flip: id-def)
    with False show ?thesis
      by (fastforce simp: comp-def trace.natural'.append image-subset-iff trace.natural'.eq-Nil-conv
                intro: exI[where x=map (map-prod id (Pair ls)) us @ (a, (s_x, s')) # zs for zs]
                 dest: Cons.hyps[where ls=fst (snd x)])
  qed
 qed
 fix σ
 assume ⦇σ⦈ ≤ ?lhs
 then obtain ls xs v i
  where ∗: ⦇(ls, sf (trace.init σ)), xs, v⦈ ≤ P
    and ∗∗: trace.natural' (sf (trace.init σ)) (map (map-prod id sf) (trace.rest σ))
          = trace.natural' (sf (trace.init σ)) (map (map-prod id snd) (take i xs))
    and ∗∗∗: if i ≤ length xs then trace.term σ = None else map-option vf (trace.term σ) = v
  apply (clarsimp simp: spec.singleton.le-conv spec.singleton-le-conv)
  apply (erule trace.less-eq-takeE)
  apply (erule trace.take.naturalE)
  apply (clarsimp simp: trace.take.map trace.natural-def trace.split-all not-le
              split: if-split-asm)
  apply (metis order.strict-iff-not take-all)
  done
 from smap-snd-aux[OF ∗∗] obtain zs
  where trace.natural' (ls, sf (trace.init σ)) (take i xs)
      = trace.natural' (ls, sf (trace.init σ)) (map (map-prod id (map-prod id sf)) zs)
    and trace.natural' (trace.init σ) (map (map-prod id snd) zs)
      = trace.natural' (trace.init σ) (trace.rest σ)
  by blast
 with ∗ ∗∗∗ show ⦇σ⦈ ≤ ?rhs
  apply −
  unfolding spec.singleton.le-conv
  apply (rule exI[where x=trace.T (ls, trace.init σ) zs (if i ≤ length xs then None else trace.term σ)])
  apply (clarsimp simp: spec.singleton-le-conv trace.natural-def trace.less-eq-None
              split: if-splits
              elim!: order.trans[rotated])
  apply (metis append-take-drop-id prefixI trace.natural'.append)
  done
next
 show ⦇σ⦈ ≤ ?lhs if ⦇σ⦈ ≤ ?rhs for σ
  using that
  by (fastforce dest: spec.singleton.map-le[where af=id and sf=sf and vf=vf]
            simp: spec.singleton.le-conv)
qed

lemma local:
 fixes P :: ('a agent, 'ls × 't, 'v) spec
 fixes sf :: 's ⇒ 't
 shows spec.invmap id sf vf (spec.local P) = spec.local (spec.invmap id (map-prod id sf) vf P)
by (auto simp: spec.local-def spec.local.qrm-def ac-simps
          spec.invmap.smap-snd spec.invmap.inf spec.invmap.rel
 intro!: arg-cong[where f=λr. spec.smap snd (spec.invmap id (map-prod id sf) vf P ⊓ spec.rel r)])

setup ‹Sign.parent-path›

setup ‹Sign.mandatory-path term›

setup ‹Sign.mandatory-path none›
```

lemma *local*:
shows *spec.term.none* (*spec.local P*) = *spec.local* (*spec.term.none P*)
by (*simp add*: *spec.local-def spec.term.none.inf spec.term.none.inf-none-rel spec.term.none.map*)

setup ‹*Sign.parent-path*›

setup ‹*Sign.mandatory-path all*›

lemma *local*:
shows *spec.term.all* (*spec.local P*) = *spec.local* (*spec.term.all P*)
by (*simp add*: *spec.local-def spec.term.all.map spec.term.all.rel spec.term.all.inf*)

setup ‹*Sign.parent-path*›

setup ‹*Sign.mandatory-path closed*›

lemma *local*:
assumes *P* ∈ *spec.term.closed* -
shows *spec.local P* ∈ *spec.term.closed* -
by (*rule spec.term.closed-clI*)
(*simp add*: *spec.term.all.local spec.term.all.monomorphic*
flip: *spec.term.closed-conv*[*OF assms*, *simplified*])

setup ‹*Sign.parent-path*›

setup ‹*Sign.parent-path*›

setup ‹*Sign.parent-path*›

15.2 Local state transformations

We want to reorder, introduce and eliminate actions that affect local state while preserving observable behaviour under *spec.local*.

The closure that arises from *spec.local*, i.e.:

lemma
defines *cl* ≡ *spec.map-invmap.cl* - - - *id snd id*
assumes *spec.local.interference* ⊓ *P*
≤ *cl* (*spec.local.interference* ⊓ *Q*)
shows *spec.local P* ≤ *spec.local Q*
unfolding *spec.local-def*
by (*strengthen ord-to-strengthen*(*1*)[*OF assms*(*2*)])
(*simp add*: *spec.map-invmap.galois cl-def spec.map-invmap.cl-def*)

expresses all transformations, but does not decompose over (≫=); in other words we do not have *cl f* ≫= (λ*v*. *cl* (*g v*)) ≤ *cl* (*f* ≫= *g*) as the local states that *cl f* terminates with may not satisfy *g*. (Observe that we do not expect the converse to hold as then all local states would need to be preserved.)

We therefore define a closure that preserves the observable state and the initial and optionally final (if terminating) local states via a projection:

setup ‹*Sign.mandatory-path seq-ctxt*›

definition *prj* :: *bool* ⇒ ('*a*, '*ls* × '*s*, '*v*) *trace.t* ⇒ ('*a*, '*s*, '*v*) *trace.t* × '*ls* × '*ls option* **where**
prj T σ = (⇂(*trace.map id snd id* σ),
fst (*trace.init* σ),
if T then map-option ⟨*fst* (*trace.final* σ)⟩ (*trace.term* σ) *else None*)

setup ‹*Sign.mandatory-path prj*›

lemma *natural*:
  shows *seq-ctxt.prj T* (♮*σ*) = *seq-ctxt.prj T σ*
by (*simp add*: *seq-ctxt.prj-def trace.natural.map-natural*)

lemma *idle*:
  shows *seq-ctxt.prj T* (*trace.T s* [] *None*) = (*trace.T* (*snd s*) [] *None*, *fst s*, *None*)
by (*simp add*: *seq-ctxt.prj-def trace.natural.simps*)

lemmas *simps*[*simp*] =
  seq-ctxt.prj.natural

setup ‹*Sign.parent-path*›

setup ‹*Sign.parent-path*›

interpretation *seq-ctxt*: *galois.image-vimage seq-ctxt.prj T* **for** *T* **.**

setup ‹*Sign.mandatory-path seq-ctxt.equivalent*›

lemma *partial-sel-equivE*:
  assumes *seq-ctxt.equivalent T σ*$_1$ *σ*$_2$
  obtains *trace.init σ*$_1$ = *trace.init σ*$_2$
      and *trace.term σ*$_1$ = *trace.term σ*$_2$
      and ⟦*T*; ∃ *v*. *trace.term σ*$_1$ = *Some v*⟧ ⟹ *trace.final σ*$_1$ = *trace.final σ*$_2$
using *assms*
by (*cases σ*$_1$)
   (*force intro*: *prod-eqI*
          simp: *seq-ctxt.prj-def trace.natural.trace-conv*
      simp flip: *trace.final′.map*[**where** *af*=*id* **and** *sf*=*snd*]
          cong: *trace.final′.natural′-cong*)

lemma *downwards-existsE*:
  assumes *σ*$_1$′ ≤ *σ*$_1$
  assumes *seq-ctxt.equivalent T σ*$_1$ *σ*$_2$
  obtains *σ*$_2$′
    where *σ*$_2$′ ≤ *σ*$_2$
      and *seq-ctxt.equivalent T σ*$_1$′ *σ*$_2$′
using *assms*
apply *atomize-elim*
apply (*clarsimp simp*: *seq-ctxt.prj-def*)
apply (*rule trace.natural.less-eqE*[*OF trace.map.mono sym*], *assumption*, *assumption*)
apply (*clarsimp split*: *if-split-asm*)
 apply (*cases trace.term σ*$_1$′)
  apply (*fastforce simp*: *trace.natural-def elim*: *trace.less-eqE trace.map.less-eqR*)+
done

lemma *downwards-existsE2*:
  assumes *σ*$_1$′ ≤ *σ*$_1$
  assumes *seq-ctxt.equivalent T σ*$_1$′ *σ*$_2$′
  obtains *σ*$_2$
    where *σ*$_2$′ ≤ *σ*$_2$
      and *seq-ctxt.equivalent T σ*$_1$ *σ*$_2$
proof(*atomize-elim*, *use* ‹*σ*$_1$′ ≤ *σ*$_1$› **in** ‹*induct rule*: *trace.less-eqE*›)
  case *prefix*
  from *prefix*(*3*) **obtain** *zs*
    where ∗: *σ*$_1$ = *trace.T* (*trace.init σ*$_1$) (*trace.rest σ*$_1$′ @ *zs*) (*trace.term σ*$_1$)
    by (*cases σ*$_1$) (*auto elim*: *prefixE*)

show *?case*
proof(*cases trace.term* σ_1)
case *None* **with** *assms(2) prefix(1,2)* ∗ **show** *?thesis*
by (*cases* σ_1, *cases* $\sigma_2{}'$)
(*fastforce intro*!: *exI*[**where** *x=trace.T* (*trace.init* σ_1) (*trace.rest* $\sigma_2{}'$ @ *zs*) (*trace.term* σ_1)]
simp: *seq-ctxt.prj-def trace.natural-def*
trace.natural'.append trace.less-eq-same-append-conv
cong: *trace.final'.natural'-cong*)
next
case (*Some v*)
from *assms(2) prefix(2)*
have *snd* (*trace.final* $\sigma_1{}'$) = *trace.final* (*trace.map id snd id* $\sigma_2{}'$)
by (*cases* $\sigma_1{}'$, *cases* $\sigma_2{}'$)
(*clarsimp simp*: *seq-ctxt.prj-def trace.natural-def trace.final'.map*
dest!: *arg-cong*[**where** *f*=λ*xs. trace.final'* (*snd* (*trace.init* σ_1)) *xs*])
with *Some assms(2) prefix(1,2)* ∗ **show** *?thesis*
apply (*cases* σ_1)
apply (*cases* $\sigma_2{}'$)
apply (*rule exI*[**where** *x=trace.T* (*trace.init* σ_1) (*trace.rest* $\sigma_2{}'$ @ (*undefined*, *trace.final* $\sigma_1{}'$) # *zs*) (*trace.term* σ_1)])
apply (*clarsimp simp*: *seq-ctxt.prj-def trace.natural-def trace.natural'.append trace.less-eq-same-append-conv*)
apply (*clarsimp cong*: *trace.final'.natural'-cong*)
done
qed
next
case (*maximal v*) **with** *assms(2)* **show** *?case*
by *blast*
qed

lemma *map-sf-eq-id*:
assumes *seq-ctxt.equivalent True* σ_1 σ_2
shows *seq-ctxt.equivalent True* (*trace.map af id vf* σ_1) (*trace.map af id vf* σ_2)
using *assms*
by (*auto simp*: *seq-ctxt.prj-def comp-def trace.final'.map*[**where** *sf=id*, *simplified*] *trace.natural-def*
simp flip: *trace.natural'.map-inj-on-sf*
dest: *arg-cong*[**where** *f=map* (*map-prod af id*)])

lemma *mono*:
assumes *T* ⟹ *T'*
assumes *seq-ctxt.equivalent T'* σ_1 σ_2
shows *seq-ctxt.equivalent T* σ_1 σ_2
using *assms* **by** (*clarsimp simp*: *seq-ctxt.prj-def*)

lemma *append*:
assumes *seq-ctxt.equivalent True* (*trace.T s xs* (*Some v*)) (*trace.T s' xs' v'*)
assumes *seq-ctxt.equivalent T* (*trace.T* (*trace.final' s xs*) *ys w*) (*trace.T t' ys' w'*)
shows *seq-ctxt.equivalent T* (*trace.T s* (*xs* @ *ys*) *w*) (*trace.T s'* (*xs'* @ *ys'*) *w'*)
using *assms*
by (*clarsimp simp*: *seq-ctxt.prj-def trace.natural-def trace.natural'.append*
simp flip: *trace.final'.map*[**where** *af=id* **and** *sf=snd*]
cong: *trace.final'.natural'-cong if-cong*)
(*simp*; *metis trace.final'.map*[**where** *af=id* **and** *sf=fst*])

setup ‹*Sign.parent-path*›

setup ‹*Sign.mandatory-path spec*›

setup ‹*Sign.mandatory-path seq-ctxt*›

definition *cl* :: *bool* ⇒ ('*a*, '*ls* × '*s*, '*v*) *spec* ⇒ ('*a*, '*ls* × '*s*, '*v*) *spec* **where**
cl T P = ⨆(*spec.singleton* ‘ {σ_1. ∃σ_2. ⟪σ_2⟫ ≤ *P* ∧ *seq-ctxt.equivalent T* σ_1 σ_2})

setup ‹*Sign.parent-path*›

setup ‹*Sign.mandatory-path singleton.seq-ctxt*›

lemma *cl-le-conv*[*spec.singleton.le-conv*]:
shows ⟪σ⟫ ≤ *spec.seq-ctxt.cl T P* ⟷ (∃σ'. ⟪σ'⟫ ≤ *P* ∧ *seq-ctxt.equivalent T* σ σ') (**is** *?lhs* ⟷ *?rhs*)
proof(*rule iffI*)
show *?lhs* ⟹ *?rhs*
by (*clarsimp simp*: *spec.seq-ctxt.cl-def spec.singleton-le-conv*)
(*force elim*: *seq-ctxt.equivalent.downwards-existsE*[**where** *T*=*T*]
dest: *order.trans*[*OF spec.singleton.mono*])
show *?rhs* ⟹ *?lhs*
unfolding *spec.seq-ctxt.cl-def spec.singleton-le-conv* **by** *blast*
qed

setup ‹*Sign.parent-path*›

interpretation *seq-ctxt*: *closure-complete-distrib-lattice-distributive-class spec.seq-ctxt.cl T* **for** *F*
proof *standard*
show *P* ≤ *spec.seq-ctxt.cl T Q* ⟷ *spec.seq-ctxt.cl T P* ≤ *spec.seq-ctxt.cl T Q* (**is** *?lhs* ⟷ *?rhs*)
for *P Q* :: ('*a*, '*ls* × '*s*, '*v*) *spec*
proof(*rule iffI*)
show *?lhs* ⟹ *?rhs*
by (*rule spec.singleton-le-extI*)
(*force simp*: *spec.singleton.seq-ctxt.cl-le-conv dest*: *order.trans*[*rotated*])
show *?rhs* ⟹ *?lhs*
by (*metis spec.singleton.seq-ctxt.cl-le-conv spec.singleton-le-ext-conv*)
qed
show *spec.seq-ctxt.cl T* (⨆ *X*) ≤ ⨆ (*spec.seq-ctxt.cl T* ‘ *X*) ⊔ *spec.seq-ctxt.cl T* ⊥
for *X* :: ('*a*, '*ls* × '*s*, '*v*) *spec set*
by (*auto simp*: *spec.seq-ctxt.cl-def*)
qed

setup ‹*Sign.mandatory-path idle.seq-ctxt*›

lemma *cl-le-conv*[*spec.idle-le*]:
shows *spec.idle* ≤ *spec.seq-ctxt.cl T P* ⟷ *spec.idle* ≤ *P* (**is** *?lhs* ⟷ *?rhs*)
proof(*rule iffI*[*OF - order.trans*[*OF - spec.seq-ctxt.expansive*]])
show *?lhs* ⟹ *?rhs*
by (*clarsimp simp*: *spec.idle-def spec.singleton.le-conv*)
(*metis trace.take.0 seq-ctxt.equivalent.partial-sel-equivE spec.singleton.takeI trace.t.sel*(*1*))
qed

setup ‹*Sign.parent-path*›

setup ‹*Sign.mandatory-path seq-ctxt.cl*›

lemma *bot*[*simp*]:
shows *spec.seq-ctxt.cl T* ⊥ = ⊥
by (*simp add*: *spec.seq-ctxt.cl-def spec.singleton.not-bot*)

lemma *mono*:
assumes *T'* ⟹ *T*
assumes *P* ≤ *P'*

shows *spec.seq-ctxt.cl T P* ≤ *spec.seq-ctxt.cl T′ P′*
unfolding *spec.seq-ctxt.cl-def*
by (*strengthen ord-to-strengthen*(*1*)[*OF* ‹*P* ≤ *P′*›])
(*blast intro*: *seq-ctxt.equivalent.mono*[*OF assms*(*1*)])

lemma *strengthen*[*strg*]:
assumes *st-ord* (¬ *F*) *T T′*
assumes *st-ord F P P′*
shows *st-ord F* (*spec.seq-ctxt.cl T P*) (*spec.seq-ctxt.cl T′ P′*)
using *assms* **by** (*cases F*; *simp add*: *spec.seq-ctxt.cl.mono le-bool-def*)

lemma *Sup*:
shows *spec.seq-ctxt.cl T* (⨆ *X*) = ⨆ (*spec.seq-ctxt.cl T* ‘ *X*)
by (*simp add*: *spec.seq-ctxt.cl-Sup*)

lemmas *sup* = *spec.seq-ctxt.cl.Sup*[**where** *X*={*P*, *Q*} **for** *P Q*, *simplified*]

lemma *singleton*:
shows *spec.seq-ctxt.cl T* ⟪*σ*⟫ = ⨆ (*spec.singleton* ‘ {*σ′*. *seq-ctxt.equivalent T σ σ′*}) (**is** *?lhs* = *?rhs*)
proof(*rule antisym*)
show *?lhs* ≤ *?rhs*
by (*clarsimp simp*: *spec.seq-ctxt.cl-def spec.singleton-le-conv*)
(*metis seq-ctxt.equivalent.downwards-existsE2 seq-ctxt.prj.natural trace.natural.mono*)
show *?rhs* ≤ *?lhs*
by (*fastforce simp*: *spec.seq-ctxt.cl-def*)
qed

lemma *idle*: — not *simp* friendly
shows *spec.seq-ctxt.cl T* (*spec.idle* :: (*′a*, *′ls* × *′s*, *′v*) *spec*)
= *spec.term.none* (*spec.rel* (*UNIV* × (*UNIV* ×$_R$ *Id*)) :: (*′a*, *′ls* × *′s*, *′w*) *spec*) (**is** *?lhs* = *?rhs*)
proof(*rule spec.singleton.antisym*)
have ∗: *s* = *s′*
if *snd* ‘ *set xs* ⊆ {(*ls*$_0$, *s*$_0$)}
and *trace.natural′ s*$_0$ (*map* (*map-prod id snd*) *ys*) = *trace.natural′ s*$_0$ (*map* (*map-prod id snd*) *xs*)
and (*a*, (*ls*, *s*), *ls′*, *s′*) ∈ *trace.steps′* (*ls*$_0$, *s*$_0$) *ys*
for *xs ys* :: (*′a* × (*′ls* × *′s*)) *list* **and** *ls*$_0$ *s*$_0$ *a ls s ls′* **and** *s′*
using *that*
proof(*induct ys rule*: *rev-induct*)
case *snoc* **from** *snoc.prems* **show** *?case*
by (*auto simp*: *trace.natural′.append trace.steps′.append split-pairs*
trace.final′.map[**where** *s*=(*ls*$_0$, *s*$_0$) **and** *sf*=*snd*, *simplified*]
trace.natural′.map-natural′[**where** *sf*=*snd* **and** *s*=(*ls*$_0$, *s*$_0$), *simplified*]
simp flip: *id-def trace.natural′.eq-Nil-conv*
split: *if-split-asm*
dest: *arg-cong*[**where** *f*=λ*xs*. *trace.natural′ s*$_0$ (*map* (*map-prod id snd*) *xs*)]
intro: *snoc.hyps*[*OF snoc.prems*(*1*)])
qed *simp*
show ⟪*σ*⟫ ≤ *?rhs* **if** ⟪*σ*⟫ ≤ *?lhs* **for** *σ*
using *that*
by (*cases σ*)
(*clarsimp simp*: *spec.singleton.le-conv* ∗ *trace.split-all seq-ctxt.prj-def trace.natural-def*)
have ∗: *s′* = *snd s*
if *trace.steps′ s xs* ⊆ *UNIV* × (*UNIV* ×$_R$ *Id*)
and (*a*, (*ls′*, *s′*)) ∈ *set xs*
for *a s ls′ s′* **and** *xs* :: (*′a* × (*′ls* × *′s*)) *list*
using *that* **by** (*induct xs arbitrary*: *s*) (*auto simp*: *trace.steps′.Cons-eq-if split*: *if-splits*)
show ⟪*σ*⟫ ≤ *?lhs* **if** ⟪*σ*⟫ ≤ *?rhs* **for** *σ*
using *that*

by (*cases trace.init* σ)
(*fastforce simp*: *spec.singleton.le-conv seq-ctxt.prj-def trace.natural-def map-prod.comp*
dest: $*$
intro: *exI*[**where** *x*=*trace.map id* (*map-prod* ⟨*fst* (*trace.init* σ)⟩ *id*) *id* σ])
qed

lemma *invmap-le*:
shows *spec.seq-ctxt.cl True* (*spec.invmap af id vf P*) ≤ *spec.invmap af id vf* (*spec.seq-ctxt.cl True P*)
by (*rule spec.singleton-le-extI*)
(*auto simp*: *spec.singleton.le-conv dest*: *seq-ctxt.equivalent.map-sf-eq-id*)

lemma *map-le*:
shows *spec.map af id vf* (*spec.seq-ctxt.cl True P*) ≤ *spec.seq-ctxt.cl True* (*spec.map af id vf P*)
by (*rule spec.singleton-le-extI*)
(*clarsimp simp*: *spec.singleton.le-conv*
dest!: *seq-ctxt.equivalent.map-sf-eq-id*[**where** *af*=*af* **and** *vf*=*vf*];
meson order.refl order.trans spec.singleton.seq-ctxt.cl-le-conv)

setup ‹*Sign.parent-path*›

setup ‹*Sign.mandatory-path term.none.seq-ctxt*›

lemma *cl*:
shows *spec.term.none* (*spec.seq-ctxt.cl T P*) = *spec.seq-ctxt.cl T* (*spec.term.none P*)
by (*rule spec.singleton.antisym*)
(*auto simp*: *spec.singleton.le-conv seq-ctxt.prj-def trace.split-Ex trace.natural-def*)

lemma *cl-True-False*:
shows *spec.seq-ctxt.cl True* (*spec.term.none f*) = *spec.seq-ctxt.cl False* (*spec.term.none f*)
by (*rule spec.singleton.antisym*)
(*auto simp*: *spec.singleton.le-conv seq-ctxt.prj-def trace.split-Ex trace.natural-def*)

setup ‹*Sign.parent-path*›

setup ‹*Sign.mandatory-path term.all.seq-ctxt*›

lemma *cl-le*:
shows *spec.seq-ctxt.cl T* (*spec.term.all P*) ≤ *spec.term.all* (*spec.seq-ctxt.cl T P*)
by (*metis spec.seq-ctxt.mono-cl spec.term.galois spec.term.lower-upper-contractive spec.term.none.seq-ctxt.cl*)

lemma *cl-False*:
shows *spec.seq-ctxt.cl False* (*spec.term.all P*) = *spec.term.all* (*spec.seq-ctxt.cl False P*)
by (*rule spec.singleton.antisym*)
(*auto simp*: *spec.singleton.le-conv seq-ctxt.prj-def trace.split-Ex trace.natural-def*)

setup ‹*Sign.parent-path*›

setup ‹*Sign.mandatory-path bind.seq-ctxt*›

lemma *cl-le*:
shows *spec.seq-ctxt.cl True f* ⤜ (λ*v*. *spec.seq-ctxt.cl T* (*g v*)) ≤ *spec.seq-ctxt.cl T* (*f* ⤜ *g*)
proof(*induct rule*: *spec.bind-le*)
case *incomplete* **show** *?case*
by (*simp add*: *spec.seq-ctxt.cl.mono spec.term.none.seq-ctxt.cl*)
next
case (*continue* σ_f σ_g *v*)
then obtain $\sigma_f{}'$ $\sigma_g{}'$
where $*$: ⟨$\sigma_f{}'$⟩ ≤ *f seq-ctxt.equivalent True* σ_f $\sigma_f{}'$

```
      ⟪σ_g'⟫ ≤ g v seq-ctxt.equivalent T σ_g σ_g'
   by (clarsimp simp: spec.singleton.seq-ctxt.cl-le-conv)
  let ?σ = trace.T (trace.init σ_f') (trace.rest σ_f' @ trace.rest σ_g') (trace.term σ_g')
  from continue(2,3) *
  have ⟪?σ⟫ ≤ f ⤜ g
   by (cases σ_f', cases σ_g')
     (fastforce intro: spec.bind.continueI[where v=v]  elim: seq-ctxt.equivalent.partial-sel-equivE)
  moreover
  from continue(2,3) *(2,4)
  have seq-ctxt.equivalent T (trace.T (trace.init σ_f) (trace.rest σ_f @ trace.rest σ_g) (trace.term σ_g)) ?σ
   by (cases σ_f, cases σ_g, cases σ_f', cases σ_g') (auto intro: seq-ctxt.equivalent.append)
  ultimately show ?case
   by (auto simp: spec.singleton.seq-ctxt.cl-le-conv)
qed

lemma clL-le:
  shows spec.seq-ctxt.cl True f ⤜ g ≤ spec.seq-ctxt.cl T (f ⤜ g)
by (strengthen ord-to-strengthen(2)[OF spec.bind.seq-ctxt.cl-le])
   (rule spec.bind.mono[OF order.refl spec.seq-ctxt.expansive])

lemma clR-le:
  shows f ⤜ (λv. spec.seq-ctxt.cl T (g v)) ≤ spec.seq-ctxt.cl T (f ⤜ g)
by (strengthen ord-to-strengthen(2)[OF spec.bind.seq-ctxt.cl-le])
   (rule spec.bind.mono[OF spec.seq-ctxt.expansive order.refl])

setup ‹Sign.parent-path›

setup ‹Sign.mandatory-path seq-ctxt›

lemma cl-local-le: — the RHS is the closure that arises from spec.local, ignoring the constraint
  shows spec.seq-ctxt.cl T P ≤ spec.map-invmap.cl - - - id snd id P
by (fastforce simp: spec.map-invmap.cl-def spec.seq-ctxt.cl-def seq-ctxt.prj-def spec.map.Sup
                    spec.map.singleton spec.singleton.map-le-conv spec.singleton-le-conv
         simp flip: spec.map-invmap.galois)

lemma cl-local:
  shows spec.local (spec.seq-ctxt.cl T (spec.local.interference ⊓ P))
      = spec.local P (is ?lhs = ?rhs)
proof(rule antisym)
  show ?lhs ≤ ?rhs
   by (simp add: spec.local-def spec.map-invmap.galois le-infI2 spec.seq-ctxt.cl-local-le
         flip: spec.map-invmap.cl-def)
  show ?rhs ≤ ?lhs
   unfolding spec.local-def by (strengthen ord-to-strengthen(2)[OF spec.seq-ctxt.expansive]) simp
qed

lemma cl-imp-local-le:
  assumes spec.local.interference ⊓ P
       ≤ spec.seq-ctxt.cl False (spec.local.interference ⊓ Q)
  shows spec.local P ≤ spec.local Q
by (subst (1 2) spec.seq-ctxt.cl-local[where T=False, symmetric])
   (use assms spec.seq-ctxt.cl[where T=False] in ‹auto intro: spec.local.mono›)

lemma cl-inf-pre:
  shows spec.pre P ⊓ spec.seq-ctxt.cl T c = spec.seq-ctxt.cl T (spec.pre P ⊓ c)
by (fastforce simp: spec.singleton.le-conv
          intro: spec.singleton.antisym
           elim: seq-ctxt.equivalent.partial-sel-equivE)
```

lemma *cl-pre-le-conv*:
shows *spec.seq-ctxt.cl T c* ≤ *spec.pre P* ⟷ *c* ≤ *spec.pre P* (**is** *?lhs* ⟷ *?rhs*)
proof(*rule iffI*)
from *spec.seq-ctxt.cl-inf-pre*[**where** *P*=*P* **and** *c*=*c*, *symmetric*]
show *?lhs* ⟹ *?rhs*
by (*auto intro*: *order.trans*[*OF spec.seq-ctxt.expansive*])
from *spec.seq-ctxt.cl-inf-pre*[**where** *P*=*P* **and** *c*=*c*, *symmetric*]
show *?rhs* ⟹ *?lhs*
by (*simp add*: *inf.absorb-iff2*)
qed

lemma *cl-inf-post*:
shows *spec.post Q* ⊓ *spec.seq-ctxt.cl True c* = *spec.seq-ctxt.cl True* (*spec.post Q* ⊓ *c*)
by (*fastforce simp*: *spec.singleton.le-conv*
intro: *spec.singleton.antisym*
elim: *seq-ctxt.equivalent.partial-sel-equivE*
split: *option.split*)

lemma *cl-post-le-conv*:
shows *spec.seq-ctxt.cl True c* ≤ *spec.post Q* ⟷ *c* ≤ *spec.post Q* (**is** *?lhs* ⟷ *?rhs*)
proof(*rule iffI*)
from *spec.seq-ctxt.cl-inf-post*[**where** *Q*=*Q* **and** *c*=*c*, *symmetric*]
show *?lhs* ⟹ *?rhs*
by (*auto intro*: *order.trans*[*OF spec.seq-ctxt.expansive*])
from *spec.seq-ctxt.cl-inf-post*[**where** *Q*=*Q* **and** *c*=*c*, *symmetric*]
show *?rhs* ⟹ *?lhs*
by (*simp add*: *inf.absorb-iff2*)
qed

setup ‹*Sign.parent-path*›

setup ‹*Sign.parent-path*›

15.2.1 Permuting local actions

We can reorder operations on the local state as these are not observable.
Firstly: an initial action F that does not change the observable state can be swapped with an arbitrary action G.

setup ‹*Sign.mandatory-path spec.seq-ctxt*›

lemma *cl-action-permuteL-le*:
fixes *F* :: (′*v* × ′*a* × (′*ls* × ′*s*) × (′*ls* × ′*s*)) *set*
fixes *G* :: ′*v* ⇒ (′*w* × ′*a* × (′*ls* × ′*s*) × (′*ls* × ′*s*)) *set*
fixes *G*′ :: (′*v*′ × ′*a* × (′*ls* × ′*s*) × (′*ls* × ′*s*)) *set*
fixes *F*′ :: ′*v*′ ⇒ (′*w* × ′*a* × (′*ls* × ′*s*) × (′*ls* × ′*s*)) *set*
— *F* does not change ′*s*, can be partial
assumes *F*: ⋀*v a s s*′. ⟦*P s*; (*v*, *a*, *s*, *s*′) ∈ *F*⟧ ⟹ *snd s*′ = *snd s*
— The final state and return value are independent of the order of actions. *F*′ does not change ′*s*, cannot be partial
assumes *FGG*′*F*′: ⋀*v w a a*′ *s s*′ *t*. ⟦*P s*; (*v*, *a*′, *s*, *t*) ∈ *F*; (*w*, *a*, *t*, *s*′) ∈ *G v*⟧
⟹ ∃ *v*′ *a*″ *a*‴ *s*″ *t*′. (*v*′, *a*″, *s*, *t*′) ∈ *G*′ ∧ (*w*, *a*‴, *t*′, *s*″) ∈ *F*′ *v*′
∧ *snd s*′ = *snd t*′ ∧ (*snd s* ≠ *snd t*′ ⟶ *a*″ = *a*) ∧ (*T* ⟶ *fst s*″ = *fst s*′) ∧ *snd s*″ = *snd t*′
shows (*spec.action F* ⤜ (λ*v*. *spec.action* (*G v*))) ⊓ *spec.pre P*
≤ *spec.seq-ctxt.cl T* (*spec.action G*′ ⤜ (λ*v*. *spec.action* (*F*′ *v*))) (**is** - ≤ *?rhs*)
unfolding *spec.bind.inf-pre*
proof(*rule order.trans*[*OF spec.bind.mono*[*OF spec.pre.inf-action-le*(*2*) *order.refl*]],
induct rule: *spec.bind-le*)

case *incomplete*
from *F* **have** *spec.term.none* (*spec.action* (*F* ∩ *UNIV* × *UNIV* × *Pre P*)) ≤ *spec.seq-ctxt.cl T spec.idle*
by (*fastforce intro*: *spec.term.none.mono*[*OF spec.action.rel-le*]
simp: *spec.seq-ctxt.cl.idle*[**where** *'w*=*'v*])
also have ... ≤ *?rhs*
by (*simp add*: *spec.idle-le spec.seq-ctxt.cl.mono*)
finally show *?case* **.**
next
case (*continue* σ_f σ_g *v*) **then show** *?case*
apply −
apply (*erule* (*1*) *spec.singleton.action-Some-leE*)
apply (*erule* (*1*) *spec.singleton.action-not-idle-leE*)
apply *clarsimp*
apply (*frule* (*1*) *F*)
apply (*frule* (*2*) *FGG'F'*)
apply (*clarsimp simp*: *spec.singleton.seq-ctxt.cl-le-conv*)
apply (*intro exI conjI*)
apply (*rule spec.bind.continueI*)
apply (*rule spec.action.stepI*, *assumption*, *blast*)
apply (*rule spec.action.stepI*[**where** *w*=*trace.term* σ_g], *simp*, *force*)
apply (*clarsimp simp*: *seq-ctxt.prj-def trace.stuttering.equiv.append-conv*
trace.stuttering.equiv.simps(*3*)[**where** *xs*=[*x*] **and** *ys*=[] **for** *x*, *simplified*])
apply (*rule exI*[**where** *x*=[]])
apply (*simp add*: *image-image trace.natural'.eq-Nil-conv*
trace-steps'-snd-le-const trace-natural'-took-step-shared-changes)
apply (*intro conjI impI*)
apply (*simp add*: *trace-steps'-snd-le-const*)
done
qed

Secondly: an initial action G that does change the observable state can be swapped with an arbitrary action action F that does not observably change the state.

lemma *cl-action-permuteR-le*:
fixes *G* :: (*'v* × *'a* × (*'ls* × *'s*) × (*'ls* × *'s*)) *set*
fixes *F* :: *'v* ⇒ (*'w* × *'a* × (*'ls* × *'s*) × (*'ls* × *'s*)) *set*
fixes *F'* :: (*'v'* × *'a* × (*'ls* × *'s*) × (*'ls* × *'s*)) *set*
fixes *G'* :: *'v'* ⇒ (*'w* × *'a* × (*'ls* × *'s*) × (*'ls* × *'s*)) *set*
— *F* does not stall if *G* makes an observable state change
assumes *G*: ⋀*v a s s'*. ⟦*P s*; (*v*, *a*, *s*, *s'*) ∈ *G*; *snd s'* ≠ *snd s*⟧
⟹ ∃ *v' w a'' t s''*. (*v'*, *a''*, *s*, *t*) ∈ *F'* ∧ (*w*, *a*, *t*, *s''*) ∈ *G' v'* ∧ *snd t* = *snd s* ∧ *snd s''* = *snd s'*
— The final state and return value are independent of the order of actions
assumes *GFF'G'*: ⋀*v w a a' s s' t*. ⟦*P s*; (*v*, *a*, *s*, *t*) ∈ *G*; (*w*, *a'*, *t*, *s'*) ∈ *F v*⟧
⟹ *snd s'* = *snd t* ∧ (∃ *v' a'' a''' s'' t'*. (*v'*, *a''*, *s*, *t'*) ∈ *F'* ∧ (*w*, *a'''*, *t'*, *s''*) ∈ *G' v'*
∧ *snd t'* = *snd s* ∧ (*T* ⟶ *fst s''* = *fst s'*) ∧ *snd s''* = *snd s'* ∧ (*snd s''* ≠ *snd t'* ⟶ *a'''* = *a*))
shows (*spec.action G* ⤜ (λ*v*. *spec.action* (*F v*))) ⊓ *spec.pre P*
≤ *spec.seq-ctxt.cl T* (*spec.action F'* ⤜ (λ*v*. *spec.action* (*G' v*)))
unfolding *spec.bind.inf-pre*
proof(*rule order.trans*[*OF spec.bind.mono*[*OF spec.pre.inf-action-le*(*2*) *order.refl*]],
induct rule: *spec.bind-le*)
case *incomplete* **show** *?case*
unfolding *spec.term.galois*
proof(*induct rule*: *spec.action-le*)
case *idle* **show** *?case*
by (*simp add*: *spec.idle-le*)
next
case (*step v a s s'*) **then show** *?case*
proof(*cases snd s'* = *snd s*)

```
    case True with step show ?thesis
      by (clarsimp simp: spec.singleton.le-conv intro!: exI[where x=None] exI[where x=trace.T s [] None])
        (simp add: seq-ctxt.prj-def trace.natural.simps order.trans[OF spec.idle.minimal-le] spec.idle-le)
  next
    case False with step show ?thesis
      by (fastforce simp: spec.singleton.le-conv seq-ctxt.prj-def trace.natural.simps
                    dest: G
                   intro: spec.bind.continueI
                    elim: spec.action.stepI)
  qed
 qed
next
 case (continue σ_f σ_g v) then show ?case
  apply −
  apply (erule (1) spec.singleton.action-Some-leE)
  apply (erule (1) spec.singleton.action-not-idle-leE)
  apply clarsimp
  apply (drule (2) GFF'G')
  apply (clarsimp simp: spec.singleton.seq-ctxt.cl-le-conv)
  apply (intro exI conjI)
   apply (rule spec.bind.continueI)
    apply (rule spec.action.stepI, assumption, blast)
   apply (rule spec.action.stepI[where w=trace.term σ_g], simp, force)
  apply (clarsimp simp: seq-ctxt.prj-def trace.stuttering.equiv.append-conv
                        trace.stuttering.equiv.simps(1)[where xs=[x] for x, simplified])
  apply (rule exI)
  apply (rule exI[where x=[]])
  apply (intro conjI)
   apply simp
  apply (fastforce simp: trace.natural'.eq-Nil-conv
                   dest: trace-natural'-took-step-shared-changes trace-natural'-took-step-shared-same)
  apply (auto simp: trace.natural'.eq-Nil-conv
        simp flip: trace.final'.map[where af=id and sf=snd]
           dest!: arg-cong[where f=snd] trace-natural'-took-step-shared-same)
  done
qed

lemma cl-action-bind-action-pre-post:
 fixes F' :: ('v × 'a × ('ls × 's) × ('ls × 's)) set
 fixes G' :: 'v ⇒ ('w × 'a × ('ls × 's) × ('ls × 's)) set
 fixes Q :: 'w ⇒ ('ls × 's) pred
 assumes ⋀v w a a' s s' s''. ⟦P s; (v, a, s, s') ∈ F; (w, a', s', s'') ∈ G v⟧ ⟹ Q w s''
 shows spec.pre P ⊓ spec.seq-ctxt.cl True (spec.action F ⤜ (λv. spec.action (G v))) ≤ spec.post Q
unfolding spec.seq-ctxt.cl-inf-pre spec.seq-ctxt.cl-post-le-conv spec.bind.inf-pre
proof(induct rule: spec.bind-le)
 case incomplete show ?case
  by (simp add: spec.term.none.post-le)
next
 case (continue σ_f σ_g v) with assms show ?case
  by (cases σ_f)
     (fastforce simp: spec.singleton.le-conv spec.singleton.action-le-conv trace.steps'.step-conv
               split: option.split-asm)
qed

lemma cl-rev-kleene-star-action-permute-le:
 fixes F G :: (unit × 'a × ('ls × 's) × ('ls × 's)) set
  — F does not stall if G changes the observable state
 assumes G: ⋀a s s'. ⟦((), a, s, s') ∈ G; snd s' ≠ snd s⟧
```

$\Longrightarrow \exists\, w\ a''\ t\ s''.\ ((),\ a'',\ s,\ t) \in F \wedge ((),\ a,\ t,\ s'') \in G \wedge snd\ t = snd\ s \wedge snd\ s'' = snd\ s'$

— The final state is independent of order of actions, F does not change $'s$, can be partial

assumes *GFFG*: $\bigwedge a\ a'\ s\ s'\ t.\ [\![((),\ a,\ s,\ t) \in G;\ ((),\ a',\ t,\ s') \in F]\!]$
$\Longrightarrow snd\ s' = snd\ t \wedge (\exists\, a''\ a'''\ t'.\ ((),\ a'',\ s,\ t') \in F \wedge ((),\ a''',\ t',\ s') \in G$
$\wedge\ snd\ t' = snd\ s \wedge (snd\ s' \neq snd\ t' \longrightarrow a''' = a))$

shows *spec.kleene.rev-star* (*spec.action G*) $\gg\!=$ (λ-::*unit*. *spec.action F*)
$\leq$ *spec.seq-ctxt.cl True* (*spec.action F* $\gg$ *spec.kleene.rev-star* (*spec.action G*)) (**is** *?lhs spec.kleene.rev-star* $\leq$ *?rhs*)

proof(*induct rule*: *spec.kleene.rev-star.fixp-induct*[**where** *P*=λR. *?lhs R* $\leq$ *?rhs*, *case-names adm bot step*])

case (*step S*)

from *assms* **have** *S* (*spec.action G*) $\gg$ *spec.action G* $\gg$ *spec.action F*
$\leq$ *S* (*spec.action G*) $\gg\!=$ (λx. *spec.seq-ctxt.cl True* (*spec.action F* $\gg\!=$ (λv. *spec.action G*)))

by (*simp add*: *spec.bind.bind*)
(*strengthen ord-to-strengthen*(*1*)[*OF spec.seq-ctxt.cl-action-permuteR-le*[**where** *P*=$\top$ **and** *T*=*True*, *simplified*]]; *force*)

also have ... $\leq$ *?rhs*

apply (*strengthen ord-to-strengthen*(*1*)[*OF spec.bind.seq-ctxt.clR-le*])
apply (*subst spec.bind.bind*[*symmetric*])
apply (*strengthen ord-to-strengthen*(*1*)[*OF step*])
apply (*strengthen ord-to-strengthen*(*1*)[*OF spec.bind.seq-ctxt.clL-le*[**where** *T*=*True*]])
apply (*strengthen ord-to-strengthen*(*2*)[*OF spec.kleene.fold-rev-starR*])
apply (*simp add*: *spec.bind.bind*)
done

moreover have *spec.return* () $\gg$ *spec.action F* $\leq$ *?rhs*
apply (*simp add*: *spec.bind.returnL spec.idle-le*)
apply (*rule order.trans*[*OF - spec.seq-ctxt.expansive*])
apply (*rule order.trans*[*OF - spec.bind.mono*[*OF order.refl spec.kleene.epsilon-rev-star-le*]])
apply *simp*
done

ultimately show *?case*
by (*simp add*: *spec.bind.supL*)

qed *simp-all*

lemma *cl-local-action-interference-permute-le*: — local actions permute with interference

fixes *F* :: (*unit* $\times$ $'a$ *agent* $\times$ ($'ls$ $\times$ $'s$) $\times$ ($'ls$ $\times$ $'s$)) *set*
fixes *r* :: $'s$ *rel*
— *F* does not block
assumes $\bigwedge s\ ls.\ \exists\, v\ a\ ls'.\ (v,\ a,\ (ls,\ s),\ (ls',\ s)) \in F$
— *F* is insensitive to and does not modify the shared state
assumes $\bigwedge v\ a\ s\ s'\ s''\ ls\ ls'.\ (v,\ a,\ (ls,\ s),\ (ls',\ s')) \in F$
$\Longrightarrow s' = s \wedge (v,\ a,\ (ls,\ s''),\ (ls',\ s'')) \in F$
shows *spec.rel* ($A \times (Id \times_R r)$) $\gg\!=$ (λ-::*unit*. *spec.action F*)
$\leq$ *spec.seq-ctxt.cl True* (*spec.action F* $\gg$ *spec.rel* ($A \times (Id \times_R r)$))
by (*simp add*: *spec.rel.monomorphic-conv spec.kleene.star-rev-star spec.rel.act-def*)
(*rule spec.seq-ctxt.cl-rev-kleene-star-action-permute-le*; *use assms* **in** *fastforce*)

lemma *cl-action-mumble-trailing-le*:
assumes $\bigwedge v\ a\ s\ s'.\ [\![P\ s;\ (v,\ a,\ s,\ s') \in F]\!]$
$\Longrightarrow \exists\, a'\ ls'.\ (v,\ a',\ s,\ (ls',\ snd\ s')) \in F'$
$\wedge\ (snd\ s' \neq snd\ s \longrightarrow a' = a) \wedge (T \longrightarrow ls' = fst\ s')$
shows *spec.action F* $\sqcap$ *spec.pre P* $\leq$ *spec.seq-ctxt.cl T* (*spec.action F'*)
proof(*rule order.trans*[*OF spec.pre.inf-action-le*(*2*)], *induct rule*: *spec.action-le*)
case *idle* **show** *?case*
by (*simp add*: *spec.idle-le*)
next
case (*step v a s s'*)
then obtain a' ls' **where** $(v,\ a',\ s,\ (ls',\ snd\ s')) \in F'$ **and** $snd\ s' \neq snd\ s \longrightarrow a' = a$ **and** $T \longrightarrow ls' = fst\ s'$
by (*blast dest*: *assms*)

then show *?case*
unfolding *spec.singleton.le-conv*
by (*fastforce intro*: *exI*[**where** *x*=*trace.T s* [(*a'*, (*ls'*, *snd s'*))] (*Some v*)] *spec.action.stepI*
simp: *seq-ctxt.prj-def trace.natural.simps*(*2*)[**where** *xs*=[]])
qed

lemma *cl-action-mumbleL-le*:
assumes $\bigwedge w\ a\ s\ s'.\ [\![P\ s;\ (w,\ a,\ s,\ s') \in G]\!]$
$\Longrightarrow \exists v\ a'\ a''\ t\ s''.\ (v,\ a',\ s,\ t) \in F' \wedge (w,\ a'',\ t,\ s'') \in G'\ v$
$\wedge\ snd\ t = snd\ s \wedge (T \longrightarrow fst\ s'' = fst\ s')$
$\wedge\ snd\ s'' = snd\ s' \wedge (snd\ s'' \neq snd\ t \longrightarrow a'' = a)$
shows *spec.action G* $\sqcap$ *spec.pre P* $\leq$ *spec.seq-ctxt.cl T* (*spec.action F'* $\gg\!=$ ($\lambda v.$ *spec.action* (*G' v*)))
using *assms* **by** (*fastforce intro*!: *spec.seq-ctxt.cl-action-permuteR-le*[**where** *F*=$\lambda v.$ ({*v*} $\times$ *UNIV* $\times$ *Id*), *simplified spec.return-def*[*symmetric*], *simplified*])

lemma *cl-action-mumbleR-le*:
assumes $\bigwedge v\ a\ s\ s'.\ [\![P\ s;\ (v,\ a,\ s,\ s') \in G]\!]$
$\Longrightarrow \exists w\ a'\ a''\ t.\ (w,\ a',\ s,\ t) \in G' \wedge (v,\ a'',\ t,\ s') \in F'\ w$
$\wedge\ snd\ t = snd\ s' \wedge (snd\ t \neq snd\ s \longrightarrow a' = a)$
shows *spec.action G* $\sqcap$ *spec.pre P* $\leq$ *spec.seq-ctxt.cl T* (*spec.action G'* $\gg\!=$ ($\lambda v.$ *spec.action* (*F' v*)))
using *assms* **by** (*fastforce intro*!: *spec.seq-ctxt.cl-action-permuteL-le*[**where** *F*=({()} $\times$ *UNIV* $\times$ *Id*), *simplified*, *simplified spec.idle-le spec.bind.returnL spec.return-def*[*symmetric*]])

lemma *cl-action-mumble-expandL-le*:
assumes $\bigwedge v\ a\ s\ s'.\ [\![P\ s;\ (v,\ a,\ s,\ s') \in F]\!] \Longrightarrow snd\ s' = snd\ s$
assumes $\bigwedge v\ w\ a\ a'\ s\ s'\ s''.\ [\![P\ s;\ (v,\ a,\ s,\ s') \in F;\ (w,\ a',\ s',\ s'') \in G\ v]\!]$
$\Longrightarrow \exists s'''.\ (w,\ a',\ s,\ s''') \in G' \wedge snd\ s''' = snd\ s'' \wedge (T \longrightarrow fst\ s''' = fst\ s'')$
shows (*spec.action F* $\gg\!=$ ($\lambda v.$ *spec.action* (*G v*))) $\sqcap$ *spec.pre P* $\leq$ *spec.seq-ctxt.cl T* (*spec.action G'*)
using *assms* **by** (*fastforce intro*!: *spec.seq-ctxt.cl-action-permuteL-le*[**where** *F'*=$\lambda v.$ ({*v*} $\times$ *UNIV* $\times$ *Id*), *simplified spec.return-def*[*symmetric*], *simplified*])

lemma *cl-action-mumble-expandR-le*:
assumes $\bigwedge v\ a\ s\ s'.\ [\![P\ s;\ (v,\ a,\ s,\ s') \in G;\ snd\ s' \neq snd\ s]\!] \Longrightarrow \exists v'\ s''.\ (v',\ a,\ s,\ s'') \in G' \wedge snd\ s'' = snd\ s'$
assumes $\bigwedge v\ w\ a\ a'\ s\ s'\ t.\ [\![P\ s;\ (v,\ a,\ s,\ t) \in G;\ (w,\ a',\ t,\ s') \in F\ v]\!]$
$\Longrightarrow snd\ s' = snd\ t \wedge (\exists a''\ s''.\ (w,\ a'',\ s,\ s'') \in G' \wedge snd\ s'' = snd\ s' \wedge (T \longrightarrow fst\ s'' = fst\ s') \wedge$
$(snd\ s'' \neq snd\ s \longrightarrow a'' = a))$
shows (*spec.action G* $\gg\!=$ ($\lambda v.$ *spec.action* (*F v*))) $\sqcap$ *spec.pre P* $\leq$ *spec.seq-ctxt.cl T* (*spec.action G'*)
using *assms* **by** (*fastforce intro*!: *spec.seq-ctxt.cl-action-permuteR-le*[**where** *F'*=({()} $\times$ *UNIV* $\times$ *Id*), *simplified*, *simplified spec.idle-le spec.bind.returnL spec.return-def*[*symmetric*]])

setup ‹*Sign.parent-path*›

setup ‹*Sign.mandatory-path spec.local*›

lemma *init-write-interference-permute-le*:
fixes *P* :: ('*a agent*, '*ls* $\times$ '*s*, '*v*) *spec*
shows *spec.local* (*spec.rel* ({*env*} $\times$ *UNIV*) $\gg\!=$ (λ-::*unit*. *spec.write* (*proc a*) (*map-prod* ⟨*ls*⟩ *id*) $\gg$ *P*))
$\leq$ *spec.local* (*spec.write* (*proc a*) (*map-prod* ⟨*ls*⟩ *id*) $\gg$ (*spec.rel* ({*env*} $\times$ *UNIV*) $\gg\!=$ (λ-::*unit*. *P*)))
apply (*rule spec.seq-ctxt.cl-imp-local-le*)
apply (*simp add*: *ac-simps spec.bind.inf-rel spec.action.inf-rel*
flip: *spec.rel.inf spec.bind.bind*)
apply (*rule order.trans*[*OF - spec.bind.seq-ctxt.cl-le*])
apply (*rule spec.bind.mono*[*OF - spec.seq-ctxt.expansive*])
apply (*rule spec.seq-ctxt.cl-local-action-interference-permute-le*)
apply *auto*
done

lemma *init-write-interference2-permute-le*:

fixes $P :: ('a\ agent, 'ls \times 's, 'v)\ spec$
shows *spec.local (spec.rel (A × (Id ×$_R$ r)) ⤜ (λ-::unit. spec.write (proc a) (map-prod ⟨ls⟩ id) ≫ P))*
≤ spec.local (spec.write (proc a) (map-prod ⟨ls⟩ id) ≫ (spec.rel (A × (Id ×$_R$ r)) ⤜ (λ-::unit. P)))
apply *(rule spec.seq-ctxt.cl-imp-local-le)*
apply *(simp add: ac-simps spec.bind.inf-rel spec.action.inf-rel*
flip: spec.rel.inf spec.bind.bind)
apply *(rule order.trans[OF - spec.bind.seq-ctxt.cl-le])*
apply *(rule spec.bind.mono[OF - spec.seq-ctxt.expansive])*
apply *(rule spec.seq-ctxt.cl-local-action-interference-permute-le)*
apply *auto*
done

lemma *trailing-local-act:*
fixes $F :: 'v \Rightarrow ('w \times 'a\ agent \times ('ls \times 's) \times ('ls \times 's))\ set$
shows *spec.local (P ⤜ (λv. spec.action (F v)))*
*= spec.local (P ⤜ (λv. spec.action {(w, a, (ls, s), (ls, s′)) |w a ls s ls′ s′. (w, a, (ls, s), (ls′, s′)) ∈ F v ∧ (a = env ⟶ ls′ = ls)})) (***is** *?lhs = ?rhs)*
proof(*rule antisym*)
show *?lhs ≤ ?rhs*
apply *(rule spec.seq-ctxt.cl-imp-local-le)*
apply *(simp add: spec.bind.inf-rel spec.action.inf-rel)*
apply *(rule order.trans[OF - spec.bind.seq-ctxt.clR-le])*
apply *(rule spec.bind.mono[OF order.refl])*
apply *(rule spec.seq-ctxt.cl-action-mumble-trailing-le[***where** *P=⊤, simplified])*
apply *(force simp: spec.local.qrm-def)*
done
show *?rhs ≤ ?lhs*
apply *(rule spec.seq-ctxt.cl-imp-local-le)*
apply *(simp add: spec.bind.inf-rel spec.action.inf-rel)*
apply *(rule order.trans[OF - spec.bind.seq-ctxt.clR-le])*
apply *(rule spec.bind.mono[OF order.refl])*
apply *(rule spec.seq-ctxt.cl-action-mumble-trailing-le[***where** *P=⊤, simplified])*
apply *(force simp: spec.local.qrm-def)*
done
qed

setup ‹*Sign.parent-path*›

15.3 *spec.localize*

We can transform a process into one with the same observable behavior that ignores a local state. For compositionality we allow the *env* steps to change the local state but not the *self* steps.

setup ‹*Sign.mandatory-path spec*›

definition *localize :: 'ls rel ⇒ ('a agent, 's, 'v) spec ⇒ ('a agent, 'ls × 's, 'v) spec* **where**
localize r P = spec.rel ({env} × (r ×$_R$ *UNIV) ∪ range proc × (Id ×*$_R$ *UNIV)) ⊓ spec.sinvmap snd P*

setup ‹*Sign.mandatory-path idle*›

lemma *localize-le:*
assumes *spec.idle ≤ P*
shows *spec.idle ≤ spec.localize r P*
by *(simp add: spec.localize-def spec.idle.rel-le spec.idle.invmap-le[OF assms] spec.idle.term.all-le-conv)*

setup ‹*Sign.parent-path*›

setup ‹*Sign.mandatory-path term*›

setup ‹*Sign.mandatory-path none*›

lemma *localize*:
  shows *spec.term.none* (*spec.localize r P*) = *spec.localize r* (*spec.term.none P*)
by (*simp add*: *spec.localize-def spec.term.none.inf spec.term.none.inf-none-rel*)
  (*simp add*: *spec.term.none.invmap*)

setup ‹*Sign.parent-path*›

setup ‹*Sign.mandatory-path all*›

lemma *localize*:
  shows *spec.term.all* (*spec.localize r P*) = *spec.localize r* (*spec.term.all P*)
by (*simp add*: *spec.localize-def spec.term.all.rel spec.term.all.inf spec.term.all.invmap*)

setup ‹*Sign.parent-path*›

setup ‹*Sign.mandatory-path closed*›

lemma *localize*:
  assumes *P* ∈ *spec.term.closed -*
  shows *spec.localize r P* ∈ *spec.term.closed -*
by (*rule spec.term.closed-clI*)
  (*simp add*: *spec.term.all.localize spec.term.all.monomorphic*
      flip: *spec.term.closed-conv*[*OF assms*, *simplified*])

setup ‹*Sign.parent-path*›

setup ‹*Sign.parent-path*›

setup ‹*Sign.mandatory-path localize*›

lemma *singleton*:
  fixes *σ* :: (*'a agent*, *'s*, *'v*) *trace.t*
  shows *spec.localize Id* ⟪*σ*⟫ = (⨆ *ls*::*'ls*. ⟪*trace.map id* (*Pair ls*) *id σ*⟫) (**is** *?lhs* = *?rhs*)
proof(*rule antisym*)
  have ∗: *map* (*map-prod id* (*map-prod* ⟨*ls*⟩ *id*)) *xs* = *xs*
    if *trace.steps'* (*ls*, *s*) *xs* ⊆ *UNIV* × (*Id* ×$_R$ *UNIV*)
   for *ls s* **and** *xs* :: (*'a agent* × (*'ls* × *'s*)) *list*
    using *that* **by** (*induct xs arbitrary*: *s*) (*auto simp*: *trace.steps'.Cons-eq-if split*: *if-split-asm*)
  have ∃ *x*. ⟪*σ''*⟫ ≤ ⟪*trace.map id* (*Pair x*) *id σ*⟫
    if ⟪*trace.map id snd id σ'*⟫ ≤ ⟪*σ*⟫
   and *σ''* ≤ *σ'*
   and *trace.steps'* (*trace.init σ''*) (*trace.rest σ''*) ⊆ *UNIV* × (*Id* ×$_R$ *UNIV*)
   for *σ' σ''* :: (*'a agent*, *'ls* × *'s*, *'v*) *trace.t*
    using *that*
    by − (*cases σ''*,
        drule spec.singleton.map-le[**where** *af*=*id* **and** *sf*=*Pair* (*fst* (*trace.init σ''*)) **and** *vf*=*id*],
          fastforce dest: ∗ *trace.map.mono*[**where** *af*=*id* **and** *sf*=λ*x*. (*fst* (*trace.init σ''*), *snd x*) **and** *vf*=*id*]
spec.singleton.mono
                intro: *exI*[**where** *x*=*fst* (*trace.init σ''*)]
                 simp: *map-prod-def split-def*
              simp flip: *id-def*)
  then show *?lhs* ≤ *?rhs*
    by (*simp add*: *spec.localize-def spec.invmap.singleton inf-Sup spec.singleton.inf-rel*
          flip: *Times-Un-distrib1*)
  show *?rhs* ≤ *?lhs*
    by (*auto simp*: *spec.localize-def spec.invmap.singleton spec.singleton.le-conv trace-steps'-map*

intro: *exI*[**where** *x*=*trace.map id* (*Pair a*) *id* σ **for** *a*])
qed

lemma *bot*:
  shows *spec.localize r* $\bot$ = $\bot$
by (*simp add*: *spec.localize-def spec.invmap.bot*)

lemma *top*:
  shows *spec.localize r* $\top$ = *spec.rel* ({*env*} × (*r* ×$_R$ *UNIV*) ∪ *range proc* × (*Id* ×$_R$ *UNIV*))
by (*simp add*: *spec.localize-def spec.invmap.top*)

lemma *Sup*:
  shows *spec.localize r* (⨆ *X*) = (⨆ *x*∈*X*. *spec.localize r x*)
by (*simp add*: *spec.localize-def spec.invmap.Sup inf-Sup image-image*)

lemmas *sup* = *spec.localize.Sup*[**where** *X*={*X*, *Y*} **for** *X Y*, *simplified*]

lemma *mono*:
  assumes *r* ⊆ *r'*
  assumes *P* ≤ *P'*
  shows *spec.localize r P* ≤ *spec.localize r' P'*
unfolding *spec.localize-def*
apply (*strengthen ord-to-strengthen*(*1*)[*OF* ‹*r* ⊆ *r'*›])
apply (*strengthen ord-to-strengthen*(*1*)[*OF* ‹*P* ≤ *P'*›])
apply *simp*
done

lemma *strengthen*[*strg*]:
  assumes *st-ord F r r'*
  assumes *st-ord F P P'*
  shows *st-ord F* (*spec.localize r P*) (*spec.localize r P'*)
using *assms* **by** (*cases F*; *simp add*: *spec.localize.mono*)

lemma *mono2mono*[*cont-intro*, *partial-function-mono*]:
  assumes *monotone orda* (≤) *r*
  assumes *monotone orda* (≤) *P*
  shows *monotone orda* (≤) (λ*x*. *spec.localize* (*r x*) (*P x*))
using *assms* **by** (*simp add*: *monotone-def spec.localize.mono*)

lemma *mcont2mcont*[*cont-intro*]:
  assumes *mcont luba orda Sup* (≤) *P*
  shows *mcont luba orda Sup* (≤) (λ*x*. *spec.localize r* (*P x*))
using *assms* **by** (*simp add*: *spec.localize-def*)

lemma *bind*:
  shows *spec.localize r* (*f* ⤜ *g*) = *spec.localize r f* ⤜ (λ*v*. *spec.localize r* (*g v*))
by (*simp add*: *spec.localize-def spec.invmap.bind spec.bind.inf-rel*)

lemma *action*:
  fixes *F* :: ('*v* × '*a agent* × '*s* × '*s*) *set*
  shows *spec.localize r* (*spec.action F*)
      = *spec.rel* ({*env*} × (*r* ×$_R$ *Id*))
       ⤜ (λ-::*unit*. *spec.action* ((*map-prod id* (*map-prod id* (*map-prod snd snd*)) −‘ *F*)
                                ∩ *UNIV* × ({*env*} × (*r* ×$_R$ *UNIV*) ∪ *range proc* × (*Id* ×$_R$ *UNIV*) ∪ *UNIV* × *Id*))
       ⤜ (λ*v*. *spec.rel* ({*env*} × (*r* ×$_R$ *Id*)) ⤜ (λ-::*unit*. *spec.return v*)))
by (*simp add*: *spec.localize-def spec.invmap.action spec.bind.inf-rel spec.action.inf-rel-reflcl spec.return.inf-rel*
          map-prod-snd-snd-vimage inf-sup-distrib Times-Int-Times relprod-inter
          spec.rel.reflcl

flip: *spec.rel.inf*)

lemma *return*:
shows (*spec.localize r* (*spec.return v*) :: (*'a agent*, *'ls* × *'s*, *'v*) *spec*)
= *spec.rel* ({*env*} × (*r* ×$_R$ *Id*)) ≫= (λ-::*unit*. *spec.return v*)
apply (*simp add*: *spec.return-def spec.localize.action*)
apply (*simp add*: *ac-simps map-prod-vimage-Times inf-sup-distrib1 Times-Int-Times*
map-prod-snd-snd-vimage relprod-inter relprod-inter-Id spec.idle-le
flip: *spec.return-def*)
apply (*simp add*: *sup.absorb1 Sigma-mono*
flip: *sup.assoc Times-Un-distrib2*)
apply (*subst spec.action.return-const*[**where** *W*={()}], *simp*, *simp*)
apply (*simp add*: *spec.bind.bind spec.bind.return*
flip: *spec.rel.act-def*[**where** *r*={*env*} × (*r* ×$_R$ *Id*), *simplified ac-simps*])
apply (*simp add*: *spec.rel.wind-bind*
flip: *spec.rel.unfoldL spec.bind.bind*)
done

lemma *rel*:
shows *spec.localize r* (*spec.rel s*)
= *spec.rel* (({*env*} × (*r* ×$_R$ *UNIV*) ∪ *range proc* × (*Id* ×$_R$ *UNIV*))
∩ *map-prod id* (*map-prod snd snd*) −‘ (*s* ∪ *UNIV* × *Id*))
by (*simp add*: *spec.localize-def spec.invmap.rel flip*: *spec.rel.inf*)

lemma *rel-le*:
shows *spec.localize Id P* ≤ *spec.rel* (*UNIV* × (*Id* ×$_R$ *UNIV*))
unfolding *spec.localize-def* **by** (*blast intro*: *le-infI1 spec.rel.mono*)

lemma *parallel*:
shows *spec.localize UNIV* (*P* ∥ *Q*) = *spec.localize UNIV P* ∥ *spec.localize UNIV Q*
by (*simp add*: *spec.localize-def spec.invmap.parallel spec.parallel.inf-rel*)

setup ‹*Sign.parent-path*›

setup ‹*Sign.mandatory-path action*›

lemma *localize-le*:
assumes *Id* ⊆ *r*
shows *spec.action* (*map-prod id* (*map-prod id* (*map-prod snd snd*)) −‘ *F* ∩ *UNIV* × *UNIV* × (*Id* ×$_R$ *UNIV*))
≤ *spec.localize r* (*spec.action F*)
unfolding *spec.localize.action*
by (*strengthen ord-to-strengthen*[*OF spec.return.rel-le*])
(*use assms* **in** ‹*force intro*!: *spec.action.mono*
simp: *spec.bind.return spec.bind.returnL spec.idle-le*›)

setup ‹*Sign.parent-path*›

setup ‹*Sign.mandatory-path interference.closed*›

lemma *localize*:
assumes *P* ∈ *spec.interference.closed* ({*env*} × *UNIV*)
shows *spec.localize UNIV P* ∈ *spec.interference.closed* ({*env*} × *UNIV*)
by (*force simp*: *spec.localize-def*
intro: *spec.interference.closed.rel subsetD*[*OF spec.interference.closed.antimono*, *rotated*]
spec.interference.closed.invmap[**where** *sf*=*snd* **and** *vf*=*id*, *OF assms*])

setup ‹*Sign.parent-path*›

setup ‹*Sign.mandatory-path local*›

lemma *localize*:
assumes $Id \subseteq r$
shows *spec.local* (*spec.localize r P*) = *P* (**is** *?lhs* = *?rhs*)
proof(*rule antisym*)
show *?lhs* ≤ *?rhs*
by (*simp add*: *ac-simps spec.local-def spec.local.qrm-def spec.localize-def spec.map-invmap.galois*)
from *assms* **show** *?rhs* ≤ *?lhs*
by (*simp add*: *spec.local-def spec.local.qrm-def spec.localize-def ac-simps*)
(*simp add*: *spec.map.rel spec.rel.UNIV relprod-inter inf.absorb1*
inf-sup-distrib Times-Int-Times map-prod-image-Times map-prod-image-relprod
flip: *Sigma-Un-distrib1 spec.map.inf-distr spec.rel.inf*)
qed

setup ‹*Sign.parent-path*›

setup ‹*Sign.parent-path*›

setup ‹*Sign.mandatory-path spec*›

setup ‹*Sign.mandatory-path bind*›

lemma *smap-sndL*:
assumes $UNIV \times (Id \times_R UNIV) \subseteq r$
shows *spec.smap snd f* ⤜ *g* = *spec.smap snd* (*f* ⤜ (λ*v*. *spec.rel r* ⊓ *spec.sinvmap snd* (*g v*))) (**is** *?lhs* = *?rhs*)
proof(*rule antisym*)
show *?lhs* ≤ *?rhs*
proof(*induct rule*: *spec.bind-le*)
case *incomplete* **show** *?case* **by** (*simp add*: *spec.map.mono spec.term.none.map*)
next
case (*continue* σ_f' σ_g *v*)
from ‹⟪σ_f'⟫ ≤ *spec.smap snd f*›
obtain σ_f
where ∗: ⟪σ_f⟫ ≤ *f* ⟪σ_f'⟫ ≤ ⟪*trace.map id snd id* σ_f⟫
by (*clarsimp simp*: *spec.singleton.le-conv*)
let *?σ* = *trace.T* (*trace.init* σ_f)
(*trace.rest* σ_f @ *map* (*map-prod id* (*Pair* (*fst* (*trace.final* σ_f)))) (*trace.rest* σ_g))
(*trace.term* σ_g)
from *continue*(*3*) ‹⟪σ_f'⟫ ≤ ⟪*trace.map id snd id* σ_f⟫›
have *trace.final* σ_f' = *snd* (*trace.final* σ_f)
by (*cases* σ_f')
(*clarsimp simp flip*: *trace.final'.map*[**where** *af*=*id* **and** *sf*=*snd*]
cong: *trace.final'.natural'-cong*)
with *assms continue*(*2,3,4*) ∗
have ⟪*?σ*⟫ ≤ *f* ⤜ (λ*v*. *spec.rel r* ⊓ *spec.sinvmap snd* (*g v*))
by (*cases* σ_f; *cases* σ_g)
(*force intro*!: *spec.bind.continueI*[**where** *v*=*v*]
simp: *spec.singleton.le-conv spec.singleton-le-conv trace-steps'-map*
trace.natural-def comp-def)
moreover
from *continue*(*3*) ‹⟪σ_f'⟫ ≤ ⟪*trace.map id snd id* σ_f⟫›
have ⟪*trace.init* σ_f', *trace.rest* σ_f' @ *trace.rest* σ_g, *trace.term* σ_g⟫ ≤ ⟪*trace.map id snd id ?σ*⟫
by (*clarsimp simp*: *comp-def spec.singleton-le-conv trace.natural-def trace.natural'.append*
cong: *trace.final'.natural'-cong*)
ultimately show *?case*
by (*force simp*: *spec.singleton.le-conv*)
qed

```
  show ?rhs ≤ ?lhs
    by (simp add: spec.bind.mono
                  spec.invmap.bind spec.map-invmap.galois spec.map-invmap.upper-lower-expansive)
qed

lemma smap-sndR:
  assumes UNIV × (Id ×_R UNIV) ⊆ r
  shows f ⤜ (λv. spec.smap snd (g v)) = spec.smap snd (spec.rel r ⊓ spec.sinvmap snd f ⤜ g) (is ?lhs = ?rhs)
proof(rule antisym)
  show ?lhs ≤ ?rhs
  proof(induct rule: spec.bind-le)
    case incomplete show ?case
    proof(rule spec.singleton-le-extI)
      show ⟪σ⟫ ≤ ?rhs if ⟪σ⟫ ≤ spec.term.none f for σ
        using assms that
        by (cases σ)
           (force simp: comp-def spec.singleton.le-conv trace-steps'-map
                intro!: exI[where x=trace.T (undefined, trace.init σ)
                                            (map (map-prod id (Pair undefined))
                                            (trace.rest σ)) None]
                        spec.bind.incompleteI)
    qed
  next
    case (continue σ_f σ_g' v)
    from ‹⟪σ_g'⟫ ≤ spec.smap snd (g v)›
    obtain σ_g
      where ⟪σ_g⟫ ≤ g v and ⟪σ_g'⟫ ≤ ⟪trace.map id snd id σ_g⟫
      by (clarsimp simp: spec.singleton.le-conv)
    let ?σ = trace.T (fst (trace.init σ_g), trace.init σ_f)
                     (map (map-prod id (Pair (fst (trace.init σ_g)))) (trace.rest σ_f) @ trace.rest σ_g)
                     (trace.term σ_g)
    from continue(2) ‹⟪σ_g'⟫ ≤ ⟪trace.map id snd id σ_g⟫›
    have snd (trace.init σ_g) = trace.final σ_f
      by (metis spec.singleton-le-conv trace.less-eqE trace.natural.sel(1) trace.t.map-sel(1))
    with assms continue(1,3) ‹⟪σ_g⟫ ≤ g v›
    have ⟪?σ⟫ ≤ spec.rel r ⊓ spec.sinvmap snd f ⤜ g
      by (cases σ_f, cases σ_g)
         (rule spec.bind.continueI[where v=v];
          force simp: spec.singleton.le-conv comp-def trace-steps'-map trace.final'.map)
    moreover
    from continue(2) ‹⟪σ_g'⟫ ≤ ⟪trace.map id snd id σ_g⟫›
    have ⟪trace.init σ_f, trace.rest σ_f @ trace.rest σ_g', trace.term σ_g'⟫ ≤ ⟪trace.map id snd id ?σ⟫
      by (cases σ_f)
         (clarsimp simp: comp-def spec.singleton-le-conv trace.natural-def trace.natural'.append;
              metis order-le-less same-prefix-prefix trace.less-eqE trace.less-eq-None(2) trace.t.sel(1) trace.t.sel(2)
trace.t.sel(3))
    ultimately show ?case
      by (force simp: spec.singleton.le-conv)
  qed
  show ?rhs ≤ ?lhs
    by (simp add: spec.bind.mono
                  spec.invmap.bind spec.map-invmap.galois spec.map-invmap.upper-lower-expansive)
qed

lemma localL:
  shows spec.local f ⤜ g = spec.local (f ⤜ (λv. spec.localize Id (g v)))
unfolding spec.local-def spec.localize-def spec.local.qrm-def
by (subst spec.bind.smap-sndL[where r={env} × (Id ×_R UNIV) ∪ range proc × (Id ×_R UNIV)];
```

fastforce simp: *ac-simps spec.bind.inf-rel inf-sup-distrib1 Times-Int-Times*
simp flip: *spec.rel.inf*)

lemma *localR*:
shows $f \gg\!= (\lambda v.\ spec.local\ (g\ v)) = spec.local\ (spec.localize\ Id\ f \gg\!= g)$
unfolding *spec.local-def spec.localize-def spec.local.qrm-def*
by (*subst spec.bind.smap-sndR*[**where** $r=\{env\} \times (Id \times_R UNIV) \cup range\ proc \times (Id \times_R UNIV)$];
fastforce simp: *ac-simps spec.bind.inf-rel inf-sup-distrib1 Times-Int-Times*
simp flip: *spec.rel.inf*)

setup ‹*Sign.parent-path*›

setup ‹*Sign.mandatory-path local*›

setup ‹*Sign.mandatory-path cam*›

lemma *cl-le*:
shows $spec.local\ (spec.cam.cl\ (\{env\} \times (s \times_R r))\ P) \leq spec.cam.cl\ (\{env\} \times r)\ (spec.local\ P)$
unfolding *spec.cam.cl-def spec.local.sup spec.term.all.local spec.term.none.local*[*symmetric*]
by (*fastforce intro*: *le-supI2 spec.term.none.mono spec.bind.mono spec.rel.mono*
simp flip: *spec.map-invmap.galois spec.rel.inf*
simp: *spec.local-def spec.map-invmap.galois spec.bind.inf-rel spec.invmap.bind spec.invmap.rel*)

lemma *cl*:
assumes $Id \subseteq r_l$
shows $spec.local\ (spec.cam.cl\ (\{env\} \times (r_l \times_R r))\ P)$
$= spec.cam.cl\ (\{env\} \times r)\ (spec.local\ P)$ (**is** *?lhs* = *?rhs*)
proof(*rule antisym*[*OF spec.local.cam.cl-le*])
have $spec.local\ (spec.term.all\ P) \gg spec.rel\ (\{env\} \times r)$
$\leq spec.local\ (spec.term.all\ P \gg spec.rel\ (\{env\} \times (r_l \times_R r)))$
proof(*induct rule*: *spec.bind-le*)
case *incomplete* **show** *?case*
by (*simp add*: *spec.term.none.local spec.local.mono order.trans*[*OF - spec.term.none.bindL-le*])
next
case (*continue* $\sigma_f\ \sigma_g\ v$)
from ‹$trace.term\ \sigma_f = Some\ v$› ‹$\langle\!|\sigma_f|\!\rangle \leq spec.local\ (spec.term.all\ P)$›
obtain *s xs w*
where *P*: $\langle\!|s, xs, w|\!\rangle \leq P$
$trace.steps'\ s\ xs \subseteq spec.local.qrm$
$\langle\!|\sigma_f|\!\rangle \leq \langle\!|snd\ s, map\ (map\text{-}prod\ id\ snd)\ xs, trace.term\ \sigma_f|\!\rangle$
by (*clarsimp simp*: *spec.singleton.local-le-conv spec.singleton.le-conv spec.singleton-le-conv*
trace.split-all trace.natural-def)
let ?σ = $trace.T\ s$
$(xs\ @\ map\ (map\text{-}prod\ id\ (Pair\ (fst\ (trace.final'\ s\ xs))))\ (trace.rest\ \sigma_g))$
$(trace.term\ \sigma_g)$
from *assms continue*(*2,3,4*) *P*(*1,3*)
have $\langle\!|?\sigma|\!\rangle \leq spec.term.all\ P \gg\!= (\lambda\text{-}::'g.\ spec.rel\ (\{env\} \times (r_l \times_R r)))$
by (*cases* σ_f)
(*fastforce intro*: *spec.bind.continueI*
simp: *spec.singleton.le-conv trace.steps'.map*(*1*)[**where** *af*=*id* **and** *sf*=*Pair* (*fst* (*trace.final'* *s* *xs*)) **and** *s*=*snd* (*trace.final'* *s* *xs*), *simplified*]
simp flip: *trace.final'.map*[**where** *af*=*id* **and** *sf*=*snd*]
cong: *trace.final'.natural'-cong*)
moreover
from *continue*(*2,3,4*) *P*(*2,3*)
have $trace.steps'\ (trace.init\ ?\sigma)\ (trace.rest\ ?\sigma) \subseteq spec.local.qrm$
by (*fastforce simp*: *spec.singleton.le-conv spec.singleton-le-conv trace.natural-def trace.natural'.append*
trace.steps'.append trace.steps'.map(*1*)[**where** *af*=*id* **and** *sf*=*Pair* (*fst* (*trace.final'* *s* *xs*))

and *s=snd (trace.final' s xs), simplified*]
simp flip: *trace.final'.map*[**where** *af=id* **and** *sf=snd*]
cong: *trace.final'.natural'-cong*)
moreover
from ‹*trace.term* σ_f = *Some v*› ‹$\langle\!|\sigma_f|\!\rangle \leq \langle\!|$*snd s, map* (*map-prod id snd*) *xs, trace.term* $\sigma_f|\!\rangle$›
have $\langle\!|$*trace.init* σ_f, *trace.rest* σ_f @ *trace.rest* σ_g, *trace.term* $\sigma_g|\!\rangle \leq \langle\!|$*trace.map id snd id* $?\sigma|\!\rangle$
by (*cases* σ_f)
(*simp add*: *spec.singleton-le-conv trace.natural'.append trace.natural-def comp-def*
cong: *trace.final'.natural'-cong*)
ultimately show *?case*
by (*force simp*: *spec.singleton.local-le-conv spec.singleton.le-conv*)
qed
then show *?rhs* $\leq$ *?lhs*
by (*auto simp*: *spec.cam.cl-def spec.local.sup spec.term.all.local*
simp flip: *spec.term.none.local*
intro: *le-supI2 spec.term.none.mono*)
qed

setup ‹*Sign.parent-path*›

setup ‹*Sign.mandatory-path cam.closed*›

lemma *local*:
assumes *Id* $\subseteq$ *s*
assumes *P* $\in$ *spec.cam.closed* ({*env*} $\times$ (*s* $\times_R$ *r*))
shows *spec.local P* $\in$ *spec.cam.closed* ({*env*} $\times$ *r*)
by (*metis spec.cam.closed spec.cam.closed-conv*[*OF assms*(*2*)] *spec.local.cam.cl*[*OF assms*(*1*)])

setup ‹*Sign.parent-path*›

setup ‹*Sign.mandatory-path interference*›

lemma *cl-le*:
shows *spec.local* (*spec.interference.cl* ({*env*} $\times$ (*s* $\times_R$ *r*)) *P*)
$\leq$ *spec.interference.cl* ({*env*} $\times$ *r*) (*spec.local P*)
by (*force intro*: *spec.interference.cl.mono*
simp: *spec.local-def spec.map-invmap.upper-lower-expansive spec.map-invmap.galois*
spec.invmap.interference.cl spec.interference.cl.inf-rel)

lemma *cl*:
assumes *Id* $\subseteq$ *s*
shows *spec.local* (*spec.interference.cl* ({*env*} $\times$ (*s* $\times_R$ *r*)) *P*)
= *spec.interference.cl* ({*env*} $\times$ *r*) (*spec.local P*)
apply (*rule antisym*[*OF spec.local.interference.cl-le*])
apply (*simp add*: *spec.interference.cl-def spec.bind.bind spec.bind.localL spec.bind.localR*
spec.invmap.bind spec.invmap.return spec.invmap.rel
flip: *spec.local.cam.cl*[*OF order.refl*])
apply (*simp add*: *spec.localize.bind spec.localize.rel spec.localize.return*)
apply (*simp add*: *spec.rel.wind-bind-trailing flip*: *spec.bind.bind*)
apply (*simp add*: *ac-simps inf-sup-distrib1 map-prod-vimage-Times Times-Int-Times*
map-prod-snd-snd-vimage relprod-inter spec.rel.reflcl spec.rel.Id UNIV-unit)
apply (*intro spec.local.mono spec.bind.mono spec.rel.mono spec.cam.cl.mono*)
using *assms* **apply** *force+*
done

setup ‹*Sign.parent-path*›

setup ‹*Sign.parent-path*›

setup ‹*Sign.mandatory-path interference.closed*›

lemma *local*:
  assumes $P \in$ *spec.interference.closed* $(\{env\} \times (Id \times_R r))$
  shows *spec.local* $P \in$ *spec.interference.closed* $(\{env\} \times r)$
by (*rule spec.interference.closed-clI*)
  (*simp flip*: *spec.interference.closed-conv*[*OF assms*] *spec.local.interference.cl*[*OF order.refl*])

lemma *local-UNIV*:
  assumes $P \in$ *spec.interference.closed* $(\{env\} \times UNIV)$
  shows *spec.local* $P \in$ *spec.interference.closed* $(\{env\} \times UNIV)$
proof −
  have ∗: $\{env\} \times (Id \times_R UNIV) \subseteq \{env\} \times UNIV$ **by** *blast*
  show *?thesis*
    by (*rule spec.interference.closed-clI*)
      (*simp flip*: *spec.interference.closed-conv*[*OF subsetD*[*OF spec.interference.closed.antimono*[*OF* ∗] *assms*]]
          spec.local.interference.cl[*OF order.refl*])
qed

setup ‹*Sign.parent-path*›

setup ‹*Sign.parent-path*›

15.4 *spec.local_init*

setup ‹*Sign.mandatory-path spec*›

definition *local-init* :: $'a \Rightarrow 'ls \Rightarrow ('a\ agent, 'ls \times 's, 'v)\ spec \Rightarrow ('a\ agent, 's, 'v)\ spec$ **where**
  local-init a ls P = *spec.local* (*spec.write* (*proc a*) (*map-prod* ⟨*ls*⟩ *id*) ≫ *P*)

setup ‹*Sign.mandatory-path singleton*›

lemma *local-init-le-conv*:
  shows $\langle\!|\sigma|\!\rangle \leq$ *spec.local-init a ls P*
      $\longleftrightarrow \langle\!|\sigma|\!\rangle \leq$ *spec.idle* $\vee$ $(\exists \sigma'.\ \langle\!|\sigma'|\!\rangle \leq P$
                  $\wedge$ *trace.steps* $\sigma' \subseteq$ *spec.local.qrm*
                  $\wedge \langle\!|\sigma|\!\rangle \leq \langle\!|$*trace.map id snd id* $\sigma'|\!\rangle$
                  $\wedge$ *fst* (*trace.init* σ') = *ls*) (**is** *?lhs* $\longleftrightarrow$ *?rhs*)
proof(*rule iffI*)
  assume *?lhs*
  then obtain σ'
      where $\langle\!|\sigma'|\!\rangle \leq$ *spec.write* (*proc a*) (*map-prod* ⟨*ls*⟩ *id*) ≫ *P*
        and *trace.steps'* (*trace.init* σ') (*trace.rest* σ') $\subseteq$ *spec.local.qrm*
        and $\langle\!|\sigma|\!\rangle \leq \langle\!|$*trace.map id snd id* $\sigma'|\!\rangle$
    by (*clarsimp simp*: *spec.local-init-def spec.singleton.local-le-conv*)
  then show *?rhs*
  proof(*induct rule*: *spec.singleton.bind-le*)
    case *incomplete* **then show** *?case*
      by (*cases* σ')
        (*rule disjI1*;
         fastforce simp: *spec.singleton.le-conv spec.singleton.action-le-conv trace.steps'.step-conv*
         elim!: *order.trans*)
  next
    case (*continue* $\sigma_f\ \sigma_g\ v_f$) **then show** *?case*
      by (*cases* σ_g)
        (*rule disjI2*; *erule* (*1*) *spec.singleton.action-Some-leE*;
         force simp: *exI*[**where** $x{=}\sigma_g$] *spec.singleton.le-conv image-image*

trace.steps′.append trace-steps′-snd-le-const)
qed
next
show *?rhs* ⟹ *?lhs*
by (*fastforce simp*: *spec.local-init-def spec.idle-le trace.split-all spec.singleton.local-le-conv*
intro!: *spec.bind.continueI*[**where** *xs*=[], *simplified*] *spec.action.stutterI*
elim!: *order.trans*)
qed

setup ‹*Sign.parent-path*›

setup ‹*Sign.mandatory-path idle*›

lemma *local-init-le*[*spec.idle-le*]:
shows *spec.idle* ≤ *spec.local-init a ls P*
by (*simp add*: *spec.local-init-def spec.idle-le*)

setup ‹*Sign.parent-path*›

setup ‹*Sign.mandatory-path local-init*›

lemma *Sup*:
shows *spec.local-init a ls* (⨆ *X*) = (⨆ *x*∈*X*. *spec.local-init a ls x*) ⊔ *spec.idle*
apply (*simp add*: *spec.local-init-def spec.local.Sup spec.local.sup image-image*
spec.bind.SUPR(*1*)[**where** *X*=*X* **and** *g*=*pred-K*, *simplified*]
spec.bind.botR spec.local.action
flip: *bot-fun-def spec.term.none.local*)
apply (*subst spec.return.cong*, *force*, *force*)
apply (*simp add*: *spec.term.none.return spec.term.none.Sup spec.term.none.sup spec.term.none.idle*
sup.absorb2)
done

lemma *Sup-not-empty*:
assumes *X* ≠ {}
shows *spec.local-init a ls* (⨆ *X*) = (⨆ *x*∈*X*. *spec.local-init a ls x*)
by (*subst spec.local-init.Sup*) (*meson assms spec.local-init.Sup sup.absorb1 SUPI spec.idle-le ex-in-conv*)

lemmas *sup* = *spec.local-init.Sup-not-empty*[**where** *X*={*X*, *Y*} **for** *X Y*, *simplified*]

lemma *bot*:
shows *spec.local-init a ls* ⊥ = *spec.idle*
using *spec.local-init.Sup*[**where** *X*={}] **by** *simp*

lemma *top*:
shows *spec.local-init a ls* ⊤ = (⊤ :: (*′a agent*, *′s*, *′v*) *spec*)
proof −
have ⟪*σ*⟫ ≤ *spec.local-init a ls* ⊤ **for** *σ* :: (*′a agent*, *′s*, *′v*) *trace.t*
by (*fastforce simp*: *spec.local-init-def spec.local.qrm-def spec.singleton.local-le-conv trace.steps′.map comp-def*
intro: *exI*[**where** *x*=*trace.T* (*ls*, *trace.init σ*) (*map* (*map-prod id* (*Pair ls*)) (*trace.rest σ*)) (*trace.term σ*)]
spec.bind.continueI[**where** *xs*=[], *simplified*] *spec.action.stutterI*)
then show *?thesis*
by (*fastforce intro*: *top-le spec.singleton-le-extI simp*: *spec.local-init-def*)
qed

lemma *monotone*:
shows *mono* (*spec.local-init a ls* :: (*′a agent*, *′ls* × *′s*, *′v*) *spec* ⇒ -)
proof(*rule monotoneI*)

show *spec.local-init a ls P* ≤ *spec.local-init a ls P'* **if** *P* ≤ *P'* **for** *P P'* :: (*'a agent*, *'ls* × *'s*, *'v*) *spec*
unfolding *spec.local-init-def* **by** (*strengthen ord-to-strengthen*(*1*)[*OF* ‹*P* ≤ *P'*›]) *simp*
qed

lemmas *strengthen*[*strg*] = *st-monotone*[*OF spec.local-init.monotone*]
lemmas *mono* = *monotoneD*[*OF spec.local-init.monotone*]

lemma *mono2mono*[*cont-intro*, *partial-function-mono*]:
assumes *monotone orda* (≤) *P*
shows *monotone orda* (≤) (λ*x*. *spec.local-init a ls* (*P x*))
by (*simp add*: *monotone2monotone*[*OF spec.local-init.monotone assms*])

lemma *mcont2mcont*[*cont-intro*]:
assumes *mcont luba orda Sup* (≤) *P*
shows *mcont luba orda Sup* (≤) (λ*x*. *spec.local-init a ls* (*P x*))
by (*simp add*: *spec.local-init-def assms*)

lemma *action*:
fixes *F* :: (*'v* × *'a agent* × (*'ls* × *'s*) × (*'ls* × *'s*)) *set*
shows *spec.local-init a ls* (*spec.action F*)
= *spec.action* {(*v*, *a*, *s*, *s'*) |*v a ls' s s'*. (*v*, *a*, (*ls*, *s*), (*ls'*, *s'*)) ∈ *F* ∧ (*a* = *env* ⟶ *ls'* = *ls*)} (**is** *?lhs* = *?rhs*)
proof(*rule antisym*)
show *?lhs* ≤ *?rhs*
apply (*subst* (*3*) *spec.local.localize*[**where** *r*=*UNIV*, *symmetric*], *simp*)
apply (*simp add*: *spec.local-init-def*)
apply (*rule spec.seq-ctxt.cl-imp-local-le*)
apply (*simp add*: *spec.localize.action spec.bind.inf-rel spec.action.inf-rel spec.return.inf-rel*)
apply (*strengthen ord-to-strengthen*(*2*)[*OF spec.return.rel-le*])
apply (*fastforce intro*: *spec.seq-ctxt.cl-action-mumble-expandL-le*[**where** *P*=⊤, *simplified*]
simp: *spec.local.qrm-def spec.bind.returnL spec.idle-le*)
done
show *?rhs* ≤ *?lhs*
apply (*subst* (*1*) *spec.local.localize*[**where** *r*=*UNIV*, *symmetric*], *simp*)
apply (*simp add*: *spec.local-init-def*)
apply (*rule spec.seq-ctxt.cl-imp-local-le*)
apply (*simp add*: *spec.localize.action spec.bind.inf-rel spec.action.inf-rel spec.return.inf-rel*
spec.rel.Id spec.bind.returnL spec.idle-le UNIV-unit
flip: *spec.rel.inf*)
apply (*fastforce intro*: *spec.seq-ctxt.cl-action-mumbleL-le*[**where** *P*=⊤, *simplified*]
simp: *spec.local.qrm-def*)
done
qed

lemma *return*:
shows *spec.local-init a ls* (*spec.return v*) = *spec.return v*
by (*auto simp*: *spec.local-init.action spec.return-def intro*: *arg-cong*[**where** *f*=*spec.action*])

lemma *localize-le*:
assumes *spec.idle* ≤ *P*
shows *spec.local-init a ls* (*spec.localize r P*) ≤ *P*
unfolding *spec.local-init-def spec.localize-def*
apply (*rule order.trans*[*OF spec.local.bind-le*])
apply (*simp add*: *spec.local.action*)
apply (*subst spec.return.cong*, *force*, *force*)
apply (*simp add*: *assms spec.bind.SupL spec.bind.supL*
spec.bind.returnL spec.idle.local-le spec.idle.invmap-le spec.idle.rel-le)
apply (*simp add*: *le-infI2 spec.local-def spec.map-invmap.galois*)
done

lemma *localize*:
  assumes *spec.idle* $\leq P$
  assumes $Id \subseteq r$
  shows *spec.local-init a ls* (*spec.localize r P*) = *P* (**is** *?lhs* = *?rhs*)
proof(*rule antisym*[*OF spec.local-init.localize-le*[*OF assms*(*1*)] *spec.singleton-le-extI*])
  show $\langle\!|\sigma|\!\rangle \leq$ *?lhs* **if** $\langle\!|\sigma|\!\rangle \leq$ *?rhs* **for** σ
    using *that*
    by (*cases* σ)
      (*fastforce simp*: *spec.local-init-def spec.localize-def spec.local.qrm-def comp-def*
                spec.singleton.le-conv spec.singleton.local-le-conv trace-steps′-map
            intro: *exI*[**where** *x*=*trace.map id* (*Pair ls*) *id* σ] *subsetD*[*OF* ‹$Id \subseteq r$›]
                spec.bind.continueI[**where** *xs*=[], *simplified*] *spec.action.stutterI*)
qed

lemma *inf-interference*:
  shows *spec.local-init a ls P* = *spec.local-init a ls* (*P* ⊓ *spec.local.interference*)
unfolding *spec.local-init-def*
by (*subst spec.local.inf-interference*)
  (*auto simp*: *ac-simps spec.bind.inf-rel spec.action.inf-rel*
      intro: *arg-cong*[**where** *f*=λ*x*. *spec.local* (*spec.action x* ⤜ (λ*x*. *P* ⊓ *spec.local.interference*))])

lemma *eq-local*:
  assumes *spec.idle* $\leq P$
  shows (⨆ *ls*. *spec.local-init a ls P*) = *spec.local P*
proof −
  have *spec.local* (*spec.action* {((), *proc a*, (*ls*, *s*), *ls′*, *s*) |*ls′ ls s*. *True*} ≫ *P*)
    = *spec.local P* (**is** *?lhs* = *?rhs*)
proof(*rule antisym*)
  have *?rhs* = *spec.return* () ≫ *?rhs*
    by (*simp add*: *spec.bind.returnL assms spec.idle.local-le*)
  also have ... = *spec.smap snd* (*spec.sinvmap snd* (*spec.return* ()) ≫ *spec.rel spec.local.qrm* ⊓ *P*)
    by (*simp add*: *spec.local-def spec.bind.smap-sndR*[**where** *r*=*UNIV*, *simplified spec.rel.UNIV*])
  also have *?lhs* ≤ ...
    by (*force intro*: *spec.map.mono spec.bind.mono le-infI2 spec.action.rel-le*
          simp: *spec.local-def spec.invmap.return spec.bind.bind spec.bind.return spec.bind.inf-rel*)
  finally show *?lhs* ≤ *?rhs* **.**
  show *?rhs* ≤ *?lhs*
    by (*force simp*: *assms*
        intro: *spec.local.mono order.trans*[*OF spec.bind.returnL-le*[**where** *g*=⟨*P*⟩ **and** *v*=()]]
            spec.return.action-le spec.bind.mono)
  qed
  then show *?thesis*
    by (*simp add*: *spec.local-init-def UNION-eq*
        flip: *spec.local.Sup*[**where** *X*=(λ*ls*. *spec.write* (*proc a*) (*map-prod* ⟨*ls*⟩ *id*) ≫ *P*) ‘ *UNIV*, *simplified*, *simplified image-image*]
            spec.bind.SUPL spec.action.SUP-not-empty)
qed

lemma *ag-le*:
  shows *spec.local-init a ls* (⦃*P*⦄, *Id* ×$_R$ *A* ⊢ *UNIV* ×$_R$ *G*, ⦃λ*v s*. *Q v* (*snd s*)⦄)
    ≤ ⦃λ*s*. *P* (*ls*, *s*)⦄, *A* ⊢ *G*, ⦃*Q*⦄
apply (*subst ag.reflcl-a*)
apply (*simp add*: *spec.local-init-def spec.local-def spec.local.qrm-def*
          spec.map-invmap.galois spec.invmap.ag map-prod-snd-snd-vimage)
apply (*subst inf.commute*)
apply (*subst heyting*[*symmetric*])
apply (*subst sup.commute*, *subst ag.assm-heyting*)

apply (*force intro*: *ag.spec.bind*[*rotated*] *ag.spec.action*[**where** *Q*=λ-. *FST* ((=) *ls*) ∧ *P*] *ag.mono*)
done

setup ‹*Sign.parent-path*›

setup ‹*Sign.mandatory-path bind*›

lemma *local-initL*:
shows *spec.local-init a ls f* ⤜ *g* = *spec.local-init a ls* (*f* ⤜ (λ*v*. *spec.localize Id* (*g v*)))
by (*simp add*: *spec.local-init-def spec.bind.localL spec.bind.bind*)

lemma *local-initR*:
shows *f* ⤜ (λ*v*. *spec.local-init a ls* (*g v*)) = *spec.local-init a ls* (*spec.localize Id f* ⤜ *g*)
oops

setup ‹*Sign.parent-path*›

setup ‹*Sign.mandatory-path sinvmap*›

lemma *local-init*:
fixes *P* :: ('*a agent*, '*ls* × '*t*, '*v*) *spec*
fixes *sf* :: '*s* ⇒ '*t*
shows *spec.sinvmap sf* (*spec.local-init a ls P*)
= *spec.local-init a ls* (*spec.rel* (*UNIV* × (*Id* ×$_R$ *map-prod sf sf* −‘ *Id*))
⤜ (λ-::*unit*. *spec.sinvmap* (*map-prod id sf*) *P*)) (**is** *?lhs* = *?rhs*)
proof(*rule antisym*)
let *?r* = *UNIV* × (*Id* ×$_R$ *map-prod sf sf* −‘ *Id*)
have *?lhs* = *spec.local* (*spec.rel ?r* ⤜
(λ-::*unit*. *spec.action* ({((), *proc a*, (*ls*', *s*), *ls*, *s*') |*ls*' *s s*'. *sf s* = *sf s*'}) ⤜
(λ-::*unit*. *spec.rel ?r* ⤜ (λ-::*unit*. *spec.sinvmap* (*map-prod id sf*) *P*))))
by (*simp add*: *spec.local-init-def spec.invmap.local spec.invmap.bind spec.invmap.action*
spec.bind.bind map-prod-conv map-prod-map-prod-vimage-Id)
also have ... ≤ *spec.local* (*spec.rel ?r* ⤜
(λ-::*unit*. *spec.write* (*proc a*) (*map-prod* ⟨*ls*⟩ *id*) ⤜
(λ-::*unit*. *spec.rel.act ?r* ⤜
(λ-::*unit*. *spec.rel ?r* ⤜ (λ-::*unit*. *spec.sinvmap* (*map-prod id sf*) *P*)))))
apply (*rule spec.seq-ctxt.cl-imp-local-le*)
apply (*simp add*: *ac-simps spec.bind.inf-rel spec.action.inf-rel spec.rel.act-def flip*: *spec.rel.inf*)
apply (*subst* (*4*) *spec.bind.bind*[*symmetric*])
apply ((*rule spec.seq-ctxt.cl-action-mumbleL-le*[**where** *P*=⊤, *simplified*]; *force*)
| *rule order.trans*[*OF spec.bind.mono spec.bind.seq-ctxt.cl-le*] *spec.seq-ctxt.expansive*)+
done
also have ... = *spec.local* (*spec.rel ?r* ⤜
(λ-::*unit*. *spec.write* (*proc a*) (*map-prod* ⟨*ls*⟩ *id*) ⤜
(λ-::*unit*. *spec.rel ?r* ⤜ (λ-::*unit*. *spec.sinvmap* (*map-prod id sf*) *P*))))
by (*simp add*: *spec.rel.wind-bind flip*: *spec.bind.bind spec.rel.unfoldL*)
also have ... ≤ *spec.local* (*spec.write* (*proc a*) (*map-prod* ⟨*ls*⟩ *id*) ⤜
(λ-::*unit*. *spec.rel ?r* ⤜
(λ-::*unit*. *spec.rel ?r* ⤜ (λ-::*unit*. *spec.sinvmap* (*map-prod id sf*) *P*))))
by (*rule spec.local.init-write-interference2-permute-le*)
also have ... = *spec.local* (*spec.write* (*proc a*) (*map-prod* ⟨*ls*⟩ *id*) ⤜
(λ-::*unit*. *spec.rel ?r* ⤜ (λ-::*unit*. *spec.sinvmap* (*map-prod id sf*) *P*)))
by (*simp add*: *spec.rel.wind-bind flip*: *spec.bind.bind*)
also have ... = *?rhs*
by (*simp add*: *spec.local-init-def*)
finally show *?lhs* ≤ *?rhs* .
show *?rhs* ≤ *?lhs*
by (*fastforce simp*: *spec.local-init-def spec.invmap.local spec.invmap.bind spec.invmap.action*

map-prod-conv spec.bind.bind map-prod-map-prod-vimage-Id
intro: *spec.local.mono order.trans*[*OF - spec.bind.relL-le*] *spec.bind.mono spec.action.mono*)
qed

setup ‹*Sign.parent-path*›

setup ‹*Sign.mandatory-path vmap*›

lemma *local-init*:
shows *spec.vmap vf* (*spec.local-init a ls P*) = *spec.local-init a ls* (*spec.vmap vf P*)
by (*simp add*: *spec.local-init-def spec.vmap.local spec.map.bind-inj-sf spec.map.id*)

setup ‹*Sign.parent-path*›

setup ‹*Sign.mandatory-path vinvmap*›

lemma *local-init*:
shows *spec.vinvmap vf* (*spec.local-init a ls P*) = *spec.local-init a ls* (*spec.vinvmap vf P*)
by (*simp add*: *spec.local-init-def spec.invmap.local spec.invmap.bind spec.invmap.action spec.rel.Id UNIV-unit*
spec.bind.returnL spec.idle-le)

setup ‹*Sign.parent-path*›

setup ‹*Sign.mandatory-path term.none*›

lemma *local-init*:
shows *spec.term.none* (*spec.local-init a ls P*) = *spec.local-init a ls* (*spec.term.none P*)
by (*simp add*: *spec.local-init-def spec.term.none.local spec.term.none.bind*)

setup ‹*Sign.parent-path*›

setup ‹*Sign.mandatory-path term.all*›

lemma *local-init*:
shows *spec.term.all* (*spec.local-init a ls P*)
= *spec.local-init a ls* (*spec.term.all P*) ⊔ ⨆ *range spec.return*
apply (*simp add*: *spec.local-init-def spec.term.all.local spec.term.all.bind spec.term.all.action*
spec.local.Sup spec.local.sup spec.local.action spec.local.return image-image ac-simps)
apply (*subst* (*2*) *spec.return.cong*, *force*, *force intro*!: *exI*[**where** *x*=*proc a*])
apply (*rule antisym*; *clarsimp simp*: *le-supI1 le-supI2 SUP-upper SUP-upper2 spec.idle-le*)
done

setup ‹*Sign.parent-path*›

setup ‹*Sign.mandatory-path interference.closed*›

lemma *local-init*:
assumes *P* ∈ *spec.interference.closed* ({*env*} × (*Id* ×$_R$ *r*))
shows *spec.local-init a ls P* ∈ *spec.interference.closed* ({*env*} × *r*)
by (*rule spec.interference.closed-clI*)
(*simp add*: *spec.local-init-def spec.bind.bind spec.interference.cl.action*
spec.interference.cl.bindR[*OF assms*] *spec.interference.closed.bind-relL*[*OF assms*]
order.trans[*OF spec.local.init-write-interference2-permute-le*[*simplified*]]
flip: *spec.local.interference.cl*[**where** *s*=*Id*])

setup ‹*Sign.parent-path*›

setup ‹*Sign.parent-path*›

15.5 Hoist to *('s, 'v) prog*

setup ‹*Sign.mandatory-path prog*›

lift-definition *local* :: ('*ls* × '*s*, '*v*) *prog* ⇒ ('*s*, '*v*) *prog* **is** *spec.local*
by (*blast intro*: *spec.interference.closed.local subsetD*[*OF spec.interference.closed.antimono*, *rotated*])

definition *local-init* :: '*ls* ⇒ ('*ls* × '*s*, '*v*) *prog* ⇒ ('*s*, '*v*) *prog* **where**
local-init ls P = *prog.local* (*prog.write* (*map-prod* ⟨*ls*⟩ *id*) ≫ *P*)
— equivalent to lifting *spec.local-init*; see *prog.p2s.local-init*

lift-definition *localize* :: ('*s*, '*v*) *prog* ⇒ ('*ls* × '*s*, '*v*) *prog* **is** *spec.localize UNIV*
by (*rule spec.interference.closed.localize*)

setup ‹*Sign.mandatory-path p2s*›

lemmas *local*[*prog.p2s.simps*] = *prog.local.rep-eq*

lemma *local-init*[*prog.p2s.simps*]:
shows *prog.p2s* (*prog.local-init ls P*) = *spec.local-init* () *ls* (*prog.p2s P*) (**is** *?lhs* = *?rhs*)
proof(*rule antisym*)
show *?lhs* ≤ *?rhs*
by (*simp add*: *prog.local-init-def spec.local-init-def*
prog.p2s.simps prog.p2s.action prog.p2s.interference-wind-bind
map-prod-image-Collect spec.interference.cl.action spec.bind.bind
order.trans[*OF spec.local.init-write-interference-permute-le*[*simplified*]])
show *?rhs* ≤ *?lhs*
by (*simp add*: *prog.local-init-def spec.local-init-def prog.p2s.simps prog.p2s.action*
map-prod-image-Collect
spec.local.mono[*OF spec.bind.mono*[*OF spec.interference.expansive order.refl*]])
qed

setup ‹*Sign.parent-path*›

setup ‹*Sign.mandatory-path local*›

lemma *Sup*:
shows *prog.local* (⨆ *X*) = (⨆ *x*∈*X*. *prog.local x*)
by *transfer*
(*simp add*: *spec.local.Sup spec.local.sup spec.interference.cl.bot spec.local.interference*
flip: *spec.term.none.local*)

lemmas *sup* = *prog.local.Sup*[**where** *X*={*X*, *Y*} **for** *X Y*, *simplified*]

lemma *bot*:
shows *prog.local* ⊥ = ⊥
using *prog.local.Sup*[**where** *X*={}] **by** *simp*

lemma *top*:
shows *prog.local* ⊤ = ⊤
by *transfer* (*simp add*: *spec.local.top*)

lemma *monotone*:
shows *mono prog.local*
by (*rule monotoneI*) (*transfer*; *erule monotoneD*[*OF spec.local.monotone*])

lemmas *strengthen*[*strg*] = *st-monotone*[*OF prog.local.monotone*]
lemmas *mono* = *monotoneD*[*OF prog.local.monotone*]

lemmas *mono2mono*[*cont-intro*, *partial-function-mono*]
  = *monotone2monotone*[*OF prog.local.monotone*, *simplified*, *of orda P* **for** *orda P*]

lemma *mcont2mcont*[*cont-intro*]:
  assumes *mcont luba orda Sup* (≤) *P*
  shows *mcont luba orda Sup* (≤) (λ*x*. *prog.local* (*P x*))
proof(*rule mcontI*)
  from *assms* **show** *monotone orda* (≤) (λ*x*. *prog.local* (*P x*))
    by (*blast intro*: *mcont-mono prog.local.mono2mono*)
  from *assms* **show** *cont luba orda Sup* (≤) (λ*x*. *prog.local* (*P x*))
    by (*fastforce intro*: *contI dest*: *mcont-cont contD simp*: *prog.local.Sup image-image*)
qed

lemma *bind-botR*:
  shows *prog.local* (*P* ⤜ ⊥) = *prog.local P* ⤜ ⊥
by (*simp add*: *prog.p2s.simps spec.interference.cl.bot bot-fun-def*
        spec.interference.closed.bind-relR spec.interference.closed.local-UNIV
      flip: *spec.bind.botR prog.p2s-inject*)
  (*simp add*: *spec.bind.localL spec.localize.bot*)

lemma *action*:
  shows *prog.local* (*prog.action F*) = *prog.action* (*map-prod id* (*map-prod snd snd*) ‘ *F*)
by *transfer*
  (*force simp*: *spec.local.interference.cl*[*OF subset-UNIV*, **where** *r*=*UNIV*, *simplified*] *spec.local.action*
      intro: *arg-cong*[**where** *f*=λ*F*. *spec.interference.cl* ({*env*} × *UNIV*) (*spec.action F*)])

lemma *return*:
  shows *prog.local* (*prog.return v*) = *prog.return v*
by (*simp add*: *prog.return-def prog.local.action map-prod-image-Times map-prod-snd-snd-image-Id*)

setup ‹*Sign.parent-path*›

setup ‹*Sign.mandatory-path local-init*›

lemma *transfer*[*transfer-rule*]:
  shows *rel-fun* (=) (*rel-fun cr-prog cr-prog*) (*spec.local-init* ()) *prog.local-init*
by (*simp add*: *cr-prog-def prog.p2s.local-init rel-fun-def*)

lemma *Sup*:
  shows *prog.local-init ls* (⨆ *X*) = (⨆ *x*∈*X*. *prog.local-init ls x*)
by (*simp add*: *prog.local-init-def prog.bind.SupR prog.local.Sup prog.local.sup image-image*
        prog.local.bind-botR prog.local.action)
  (*subst prog.return.cong*; *force simp*: *prog.bind.returnL*)

lemmas *sup* = *prog.local-init.Sup*[**where** *X*={*X*, *Y*} **for** *X Y*, *simplified*]

lemma *bot*[*simp*]:
  shows *prog.local-init ls* ⊥ = ⊥
using *prog.local-init.Sup*[**where** *ls*=*ls* **and** *X*={}] **by** *simp*

lemma *top*:
  shows *prog.local-init ls* ⊤ = ⊤
by (*simp add*: *prog.p2s.simps spec.local-init.top flip*: *prog.p2s-inject*)

lemma *monotone*:
  shows *mono* (*prog.local-init ls*)
unfolding *prog.local-init-def* **by** *simp*

lemmas *strengthen*[*strg*] = *st-monotone*[*OF prog.local-init.monotone*]
lemmas *mono* = *monotoneD*[*OF prog.local-init.monotone*]

lemma *mono2mono*[*cont-intro*, *partial-function-mono*]:
  assumes *monotone orda* (≤) *P*
  shows *monotone orda* (≤) (λ*x*. *prog.local-init ls* (*P x*))
by (*simp add*: *monotone2monotone*[*OF prog.local-init.monotone assms*])

lemma *mcont2mcont*[*cont-intro*]:
  assumes *mcont luba orda Sup* (≤) *P*
  shows *mcont luba orda Sup* (≤) (λ*x*. *prog.local-init ls* (*P x*))
proof(*rule mcontI*)
  from *assms* **show** *monotone orda* (≤) (λ*x*. *prog.local-init ls* (*P x*))
    by (*blast intro*: *mcont-mono prog.local-init.mono2mono*)
  from *assms* **show** *cont luba orda Sup* (≤) (λ*x*. *prog.local-init ls* (*P x*))
    by (*fastforce intro*: *contI dest*: *mcont-cont contD simp*: *prog.local-init.Sup image-image*)
qed

lemma *bind-botR*:
  shows *prog.local-init ls* (*P* ⤜ ⊥) = *prog.local-init ls P* ⤜ ⊥
by (*simp add*: *prog.local-init-def prog.local.bind-botR flip*: *prog.bind.bind*)

lemma *return*:
  shows *prog.local-init ls* (*prog.return v*) = *prog.return v* (**is** *?lhs* = *?rhs*)
proof −
  have *prog.p2s ?rhs* = *spec.local-init* () *ls* (*spec.localize UNIV* (*spec.rel* ({*env*} × *UNIV*)
                                    ⤜ (λ-::*unit*. *spec.return v*)))
    by (*simp add*: *prog.p2s.simps prog.p2s.return spec.interference.cl.return*
          spec.local-init.localize spec.idle-le)
  also have ... = *spec.local-init* () *ls* (*spec.rel* ({*env*} × *UNIV*) ⤜ (λ-::*unit*. *spec.return v*))
    by (*simp add*: *spec.localize.bind spec.localize.rel spec.localize.return*
          spec.bind.inf-rel spec.return.inf-rel
          map-prod-vimage-Times map-prod-snd-snd-vimage
          ac-simps Int-Un-distrib Int-Un-distrib2 Times-Int-Times relprod-inter spec.rel.reflcl
          spec.rel.wind-bind-trailing times-subset-iff Id-le-relprod-conv
      flip: *spec.rel.inf spec.bind.bind*)
  also have ... = *prog.p2s ?lhs*
    by (*simp add*: *prog.p2s.simps prog.p2s.return spec.interference.cl.return*)
  finally show *?thesis*
    by (*simp add*: *p2s-inject*)
qed

lemma *eq-local*:
  shows (⨆ *ls*. *prog.local-init ls P*) = *prog.local P*
by *transfer*
   (*simp add*: *spec.local-init.eq-local spec.idle-le sup.absorb1 spec.interference.least*
         spec.interference.closed.local-UNIV)

setup ‹*Sign.parent-path*›

lemma *localize-alt-def*:
  shows *prog.localize P* = *prog.rel* (*Id* ×$_R$ *UNIV*) ⊓ *prog.sinvmap snd P*
by *transfer* (*simp add*: *spec.localize-def ac-simps*)

setup ‹*Sign.mandatory-path localize*›

lemma *Sup*:
  shows *prog.localize* (⨆ *X*) = (⨆ *x*∈*X*. *prog.localize x*)

by (*simp add*: *prog.localize-alt-def prog.invmap.Sup inf-Sup inf-sup-distrib image-image prog.bind.inf-rel*
inv-image-alt-def map-prod-snd-snd-vimage relprod-inter
prog.rel.Id prog.rel.empty prog.bind.returnL prog.invmap.bot UNIV-unit
flip: *prog.rel.inf*)

lemmas *sup* = *prog.localize.Sup*[**where** *X*={*X*, *Y*} **for** *X Y*, *simplified*]

lemma *bot*:
shows *prog.localize* ⊥ = ⊥
using *prog.localize.Sup*[**where** *X*={}] **by** *simp*

lemma *top*:
shows *prog.localize* ⊤ = *prog.rel* (*Id* ×$_R$ *UNIV*)
by *transfer* (*simp add*: *spec.localize.top ac-simps*)

lemma *monotone*:
shows *mono prog.localize*
by (*rule monotoneI*) (*transfer*; *simp add*: *spec.interference.cl.mono spec.localize.mono*)

lemmas *strengthen*[*strg*] = *st-monotone*[*OF prog.localize.monotone*]
lemmas *mono* = *monotoneD*[*OF prog.localize.monotone*]
lemmas *mono2mono*[*cont-intro*, *partial-function-mono*]
= *monotone2monotone*[*OF prog.localize.monotone*, *simplified*, *of orda P* **for** *orda P*]

lemma *mcont2mcont*[*cont-intro*]:
assumes *mcont luba orda Sup* (≤) *P*
shows *mcont luba orda Sup* (≤) (λ*x*. *prog.localize* (*P x*))
proof(*rule mcontI*)
from *assms* **show** *monotone orda* (≤) (λ*x*. *prog.localize* (*P x*))
by (*blast intro*: *mcont-mono prog.localize.mono2mono*)
from *assms* **show** *cont luba orda Sup* (≤) (λ*x*. *prog.localize* (*P x*))
by (*fastforce intro*: *contI dest*: *mcont-cont contD simp*: *prog.localize.Sup image-image*)
qed

lemmas *p2s*[*prog.p2s.simps*] = *prog.localize.rep-eq*

lemma *bind*:
shows *prog.localize* (*f* ⤜ *g*) = *prog.localize f* ⤜ (λ*v*. *prog.localize* (*g v*))
by *transfer*
(*simp add*: *spec.localize.bind spec.interference.least spec.interference.closed.bind spec.bind.mono*)

lemma *parallel*:
shows *prog.localize* (*P* ∥ *Q*) = *prog.localize P* ∥ *prog.localize Q*
by *transfer* (*simp add*: *spec.localize.parallel*)

lemma *rel*:
fixes *r* :: *'s rel*
shows *prog.localize* (*prog.rel r*) = *prog.rel* (*Id* ×$_R$ *r*)
by (*subst* (*2*) *prog.rel.reflcl*[*symmetric*])
(*transfer*; *auto simp*: *spec.localize.rel intro*: *arg-cong*[**where** *f*=*spec.rel*])

lemma *action*:
shows *prog.localize* (*prog.action F*)
= *prog.action* (*map-prod id* (*map-prod snd snd*) −‘ *F* ∩ *UNIV* × (*Id* ×$_R$ *UNIV*))
by (*simp add*: *prog.localize-alt-def prog.invmap.action prog.bind.inf-rel prog.action.inf-rel*
map-prod-snd-snd-vimage relprod-inter prog.rel.Id UNIV-unit refl-relprod-conv
prog.bind.return prog.return.rel-le
inf.absorb2

flip: *prog.rel.inf*)

setup ‹*Sign.parent-path*›

setup ‹*Sign.mandatory-path local*›

lemma *localize*:
  fixes $P :: ('s, 'v)$ *prog*
  shows *prog.local* (*prog.localize* $P :: ('ls \times 's, 'v)$ *prog*) $= P$
by *transfer*
  (*simp add*: *spec.local.interference.cl*[**where** *r*=*UNIV* **and** *s*=*UNIV*, *simplified*] *spec.local.localize*
    flip: *spec.interference.closed-conv*)

setup ‹*Sign.parent-path*›

setup ‹*Sign.parent-path*›

15.6 Refinement rules

setup ‹*Sign.mandatory-path spec*›

We use *localizeA* to hoist assumes similarly to *spec.localize*.

definition *localizeA* :: (*sequential*, *'s*, *'v*) *spec* ⇒ (*sequential*, *'ls* × *'s*, *'v*) *spec* **where**
  localizeA P = *spec.local.interference* ⊓ *spec.sinvmap snd P*

setup ‹*Sign.mandatory-path localizeA*›

lemma *bot*:
  shows *spec.localizeA* ⊥ = ⊥
by (*simp add*: *spec.localizeA-def spec.invmap.bot*)

lemma *top*:
  shows *spec.localizeA* ⊤ = *spec.local.interference*
by (*simp add*: *spec.localizeA-def spec.invmap.top*)

lemma *ag-assm*:
  shows *spec.localizeA* (*ag.assm A*) = *ag.assm* (*Id* ×$_R$ *A*)
apply (*simp add*: *spec.localizeA-def spec.invmap.rel spec.local.qrm-def flip*: *spec.rel.inf*)
apply (*subst* (*1 2*) *spec.rel.reflcl*[**where** *A*=*UNIV*, *symmetric*])
apply (*auto intro*: *arg-cong*[**where** *f*=*spec.rel*])
done

setup ‹*Sign.parent-path*›

setup ‹*Sign.parent-path*›

setup ‹*Sign.mandatory-path refinement*›

setup ‹*Sign.mandatory-path spec*›

lemma *localI*: — Introduce local state
  fixes *A* :: (*sequential*, *'s*, *'v*) *spec*
  fixes *c* :: (*sequential*, *'ls* × *'s*, *'v*) *spec*
  fixes *c'* :: (*sequential*, *'s*, *'v*) *spec*
  fixes *P* :: *'s pred*
  fixes *Q* :: *'v* ⇒ *'s pred*
  assumes *c* ≤ ⦃λ*s*. *P* (*snd s*)⦄, *spec.localizeA A* ⊨ *spec.sinvmap snd c'*, ⦃λ*v s*. *Q v* (*snd s*)⦄
  shows *spec.local c* ≤ ⦃*P*⦄, *A* ⊨ *c'*, ⦃*Q*⦄

apply (*simp only*: *spec.local-def spec.map-invmap.galois spec.invmap.refinement id-apply*)
apply (*subst inf.commute*)
apply (*subst heyting*[*symmetric*])
apply (*strengthen ord-to-strengthen*[*OF assms*])
apply (*strengthen ord-to-strengthen*[*OF refinement.heyting-le*])
apply (*simp add*: *refinement.mono*[*OF order.refl - - order.refl*] *heyting spec.localizeA-def*)
done

setup ‹*Sign.mandatory-path seq-ctxt*›

lemma *local-seq-ctxt-cl*:
fixes *A* :: (*sequential*, *'s*, *'v*) *spec*
fixes *P* :: *'s pred*
fixes *Q* :: *'v* ⇒ *'s pred*
fixes *c* :: (*sequential*, *'ls* × *'s*, *'v*) *spec*
fixes *c'* :: (*sequential*, *'ls* × *'s*, *'v*) *spec*
assumes *spec.local.interference* ⊓ *c*
≤ ⦃λ*s*. *P* (*snd s*)⦄, *spec.localizeA A* ⊨ *spec.seq-ctxt.cl False* (*spec.local.interference* ⊓ *c'*), ⦃λ*v s*. *Q v* (*snd s*)⦄
shows *spec.local c* ≤ ⦃*P*⦄, *A* ⊨ *spec.local c'*, ⦃*Q*⦄
apply (*simp only*: *spec.local-def spec.map-invmap.galois spec.invmap.refinement id-apply*)
apply (*subst inf.left-idem*[*symmetric*]) — non-linear use of env constraint
apply (*strengthen ord-to-strengthen*(*1*)[*OF assms*])
apply (*subst inf.commute*)
apply (*subst heyting*[*symmetric*])
apply (*strengthen ord-to-strengthen*(*2*)[*OF refinement.heyting-le*])
apply (*subst inf.commute*, *fold spec.localizeA-def*)
apply (*rule refinement.mono*[*OF order.refl order.refl - order.refl*])
apply (*strengthen ord-to-strengthen*(*1*)[*OF spec.seq-ctxt.cl-local-le*])
apply (*simp add*: *heyting flip*: *spec.map-invmap.cl-def*)
done

lemma *cl-bind*:
fixes *f* :: (*'a agent*, *'ls* × *'s*, *'v*) *spec*
fixes *g* :: *'v* ⇒ (*'a agent*, *'ls* × *'s*, *'w*) *spec*
assumes *g*: ⋀*v*. *g v* ≤ ⦃*Q' v*⦄, *refinement.spec.bind.res* (*spec.pre P* ⊓ *spec.term.all A* ⊓ *spec.seq-ctxt.cl True f'*) *A v* ⊨ *spec.seq-ctxt.cl T* (*g' v*), ⦃*Q*⦄
assumes *f*: *f* ≤ ⦃*P*⦄, *spec.term.all A* ⊨ *spec.seq-ctxt.cl True f'*, ⦃*Q'*⦄
shows *f* ≫= *g* ≤ ⦃*P*⦄, *A* ⊨ *spec.seq-ctxt.cl T* (*f'* ≫= *g'*), ⦃*Q*⦄
by (*strengthen ord-to-strengthen*[*OF spec.bind.seq-ctxt.cl-le*]) (*rule refinement.spec.bind*[*OF assms*])

lemma *cl-action-permuteL*:
fixes *F* :: (*'v* × *'a* × (*'ls* × *'s*) × (*'ls* × *'s*)) *set*
fixes *G* :: *'v* ⇒ (*'w* × *'a* × (*'ls* × *'s*) × (*'ls* × *'s*)) *set*
fixes *G'* :: (*'v'* × *'a* × (*'ls* × *'s*) × (*'ls* × *'s*)) *set*
fixes *F'* :: *'v'* ⇒ (*'w* × *'a* × (*'ls* × *'s*) × (*'ls* × *'s*)) *set*
fixes *Q* :: *'w* ⇒ (*'ls* × *'s*) *pred*
assumes *F*: ⋀*v a s s'*. ⟦*P s*; (*v*, *a*, *s*, *s'*) ∈ *F*⟧ ⟹ *snd s'* = *snd s*
assumes *FGG'F'*: ⋀*v w a a' s s' t*. ⟦*P s*; (*v*, *a'*, *s*, *t*) ∈ *F*; (*w*, *a*, *t*, *s'*) ∈ *G v*⟧
⟹ ∃ *v' a'' a''' t'*. (*v'*, *a''*, *s*, *t'*) ∈ *G'* ∧ (*w*, *a'''*, *t'*, *s'*) ∈ *F' v'*
∧ *snd s'* = *snd t'* ∧ (*snd s* ≠ *snd t'* ⟶ *a''* = *a*)
assumes *Q*: ⋀*v w a a' s s' s''*. ⟦*P s*; (*v*, *a*, *s*, *s'*) ∈ *G'*; (*w*, *a'*, *s'*, *s''*) ∈ *F' v*⟧ ⟹ *Q w s''*
shows *spec.action F* ≫= (λ*v*. *spec.action* (*G v*)) ≤ ⦃*P*⦄, *A* ⊨ *spec.seq-ctxt.cl T* (*spec.action G'* ≫= (λ*v*. *spec.action* (*F' v*))), ⦃*Q*⦄
apply (*strengthen ord-to-strengthen*[*OF top-greatest*[**where** *a*=*T*]])
apply (*rule order.trans*[*OF spec.seq-ctxt.cl-action-permuteL-le*[**where** *T*=*True* **and** *F'*=*F'* **and** *G'*=*G'*, *simplified heyting*[*symmetric*]]])
apply (*erule* (*1*) *F*)

apply (*drule* (*2*) *FGG'F'*, *blast*)
apply (*simp only*: *refinement-def spec.pre.next-imp-eq-heyting spec.idle-le inf.bounded-iff*)
apply (*rule order.trans*[*OF - heyting.mono*[*OF order.refl spec.next-imp.mono*[*OF top-greatest order.refl*]]])
apply (*simp add*: *heyting heyting.detachment spec.seq-ctxt.cl-action-bind-action-pre-post*[*OF Q*])
done

lemma *cl-action-permuteR*:
fixes G :: $('v \times 'a \times ('ls \times 's) \times ('ls \times 's))$ *set*
fixes F :: $'v \Rightarrow ('w \times 'a \times ('ls \times 's) \times ('ls \times 's))$ *set*
fixes F' :: $('v' \times 'a \times ('ls \times 's) \times ('ls \times 's))$ *set*
fixes G' :: $'v' \Rightarrow ('w \times 'a \times ('ls \times 's) \times ('ls \times 's))$ *set*
assumes G: $\bigwedge v\ a\ s\ s'.\ [\![P\ s;\ (v,\ a,\ s,\ s') \in G;\ snd\ s' \neq snd\ s]\!]$
$\Longrightarrow \exists v'\ w\ a''\ t\ s''.\ (v',\ a'',\ s,\ t) \in F' \wedge (w,\ a,\ t,\ s'') \in G'\ v' \wedge snd\ t = snd\ s \wedge snd\ s'' = snd\ s'$
assumes $GFF'G'$: $\bigwedge v\ w\ a\ a'\ s\ s'\ t.\ [\![P\ s;\ (v,\ a,\ s,\ t) \in G;\ (w,\ a',\ t,\ s') \in F\ v]\!]$
$\Longrightarrow snd\ s' = snd\ t \wedge (\exists v'\ a''\ a'''\ t'.\ (v',\ a'',\ s,\ t') \in F' \wedge (w,\ a''',\ t',\ s') \in G'\ v'$
$\wedge\ snd\ t' = snd\ s \wedge (snd\ s' \neq snd\ t' \longrightarrow a''' = a))$
assumes Q: $\bigwedge v\ w\ a\ a'\ s\ s'\ s''.\ [\![P\ s;\ (v,\ a,\ s,\ s') \in F';\ (w,\ a',\ s',\ s'') \in G'\ v]\!] \Longrightarrow Q\ w\ s''$
shows *spec.action* $G \gg\!= (\lambda v.\ spec.action\ (F\ v)) \leq \{\!|P|\!\}, A \Vdash spec.seq\text{-}ctxt.cl\ T\ (spec.action\ F' \gg\!= (\lambda v.\ spec.action\ (G'\ v))), \{\!|Q|\!\}$
apply (*strengthen ord-to-strengthen*[*OF top-greatest*[**where** *a=T*]])
apply (*rule order.trans*[*OF spec.seq-ctxt.cl-action-permuteR-le*[**where** *T=True*, *simplified heyting*[*symmetric*]]])
apply (*erule* (*2*) *G*)
apply (*drule* (*2*) *GFF'G'*, *blast*)
apply (*simp only*: *refinement-def spec.pre.next-imp-eq-heyting spec.idle-le inf.bounded-iff*)
apply (*rule order.trans*[*OF - heyting.mono*[*OF order.refl spec.next-imp.mono*[*OF top-greatest order.refl*]]])
apply (*simp add*: *heyting heyting.detachment spec.seq-ctxt.cl-action-bind-action-pre-post*[*OF Q*])
done

setup ‹*Sign.parent-path*›

setup ‹*Sign.parent-path*›

setup ‹*Sign.mandatory-path prog*›

lemma *localI*: — Introduce local state
fixes A :: $(sequential,\ 's,\ 'v)\ spec$
fixes c :: $('ls \times 's,\ 'v)\ prog$
fixes c' :: $(sequential,\ 's,\ 'v)\ spec$
fixes P :: $'s\ pred$
fixes Q :: $'v \Rightarrow 's\ pred$
assumes *prog.p2s* $c \leq \{\!|\lambda s.\ P\ (snd\ s)|\!\}$, *spec.localizeA* $A \Vdash spec.sinvmap\ snd\ c', \{\!|\lambda v\ s.\ Q\ v\ (snd\ s)|\!\}$
shows *prog.p2s* $(prog.local\ c) \leq \{\!|P|\!\}, A \Vdash c', \{\!|Q|\!\}$
using *assms* **by** *transfer* (*erule refinement.spec.localI*)

setup ‹*Sign.parent-path*›

setup ‹*Sign.parent-path*›

15.6.1 Data refinement

In this setting a (concrete) specification c is a *data refinement* of (abstract) specification c' if:

- the observable state changes coincide
- concrete local states are mapped to abstract local states by *sf* which then coincide

Observations:

- pre/post are in terms of the concrete local states

  - *sf* can be used to lift these to the abstract local states

- we do not require c or c' to disallow the environment from changing the local state
- essentially a Skolemization of Lamport's existentials (Lamport 1994, §8)

References:

- de Roever and Engelhardt (1998, Chapter 14 "Refinement Methods due to Abadi and Lamport and to Lynch")
  - in general c will need to be augmented with auxiliary variables

setup ‹*Sign.mandatory-path refinement*›

setup ‹*Sign.mandatory-path spec*›

lemma *data*:
  fixes *A* :: (*sequential*, *'s*, *'v*) *spec*
  fixes *c* :: (*sequential*, *'cls* × *'s*, *'v*) *spec*
  fixes *c'* :: (*sequential*, *'als* × *'s*, *'v*) *spec*
  fixes *sf* :: *'cls* ⇒ *'als*
  assumes *c* ≤ {⌊λ*s*. *P* (*snd s*)⌋}, *spec.localizeA A* ⊨ *spec.sinvmap* (*map-prod sf id*) *c'*, {⌊λ*v s*. *Q v* (*snd s*)⌋}
  shows *spec.local c* ≤ {⌊*P*⌋}, *A* ⊨ *spec.local c'*, {⌊*Q*⌋}
proof −
  have ∗: *spec.smap snd* (*spec.local.interference* ⊓ *spec.sinvmap* (*map-prod sf id*) *c'*)
      ≤ *spec.smap snd* (*spec.local.interference* ⊓ *c'*) (**is** *?lhs* ≤ *?rhs*)
  proof(*rule spec.singleton-le-extI*)
    show ⟨*σ*⟩ ≤ *?rhs* **if** ⟨*σ*⟩ ≤ *?lhs* **for** *σ*
      using *that*
      by (*clarsimp simp*: *spec.singleton.le-conv*)
        (*fastforce simp*: *trace.steps'.map spec.local.qrm-def*
            simp flip: *id-def*
              intro!: *exI*[**where** *x*=*trace.map id* (*map-prod sf id*) *id σ'* **for** *σ'*])
  qed
  show *?thesis*
    apply (*simp only*: *spec.local-def spec.map-invmap.galois spec.invmap.refinement id-apply*)
    apply (*subst inf.left-idem*[*symmetric*]) — non-linear use of env constraint
    apply (*strengthen ord-to-strengthen*(*1*)[*OF assms*])
    apply (*strengthen ord-to-strengthen*(*1*)[*OF refinement.inf-le*])
    apply (*subst inf.commute*)
    apply (*subst heyting*[*symmetric*])
    apply (*strengthen ord-to-strengthen*(*2*)[*OF refinement.heyting-le*])
    apply (*subst inf.commute*, *fold spec.localizeA-def*)
    apply (*rule refinement.mono*[*OF order.refl - - order.refl*])
     apply (*simp add*: *spec.localizeA-def*; *fail*)
    apply (*simp add*: *heyting inf.absorb1* ∗ *flip*: *spec.map-invmap.galois*)
    done
qed

setup ‹*Sign.parent-path*›

setup ‹*Sign.mandatory-path prog*›

lemma *data*:
  fixes *A* :: (*sequential*, *'s*, *'v*) *spec*
  fixes *c* :: (*'cls* × *'s*, *'v*) *prog*
  fixes *c'* :: (*'als* × *'s*, *'v*) *prog*

fixes *sf* :: *'cls* ⇒ *'als*
assumes *prog.p2s c* ≤ ⦃λ*s*. *P* (*snd s*)⦄, *spec.localizeA A* ⊩ *spec.sinvmap* (*map-prod sf id*) (*prog.p2s c'*), ⦃λ*v s*. *Q v* (*snd s*)⦄
shows *prog.p2s* (*prog.local c*) ≤ ⦃*P*⦄, *A* ⊩ *prog.p2s* (*prog.local c'*), ⦃*Q*⦄
using *assms* **by** *transfer* (*erule refinement.spec.data*)

setup ‹*Sign.parent-path*›

setup ‹*Sign.parent-path*›

15.7 Assume/guarantee

setup ‹*Sign.mandatory-path ag*›

setup ‹*Sign.mandatory-path spec*›

lemma *local*:
fixes *A G* :: *'s rel*
fixes *P* :: *'s pred*
fixes *Q* :: *'v* ⇒ *'s pred*
fixes *c* :: (*sequential*, *'ls* × *'s*, *'v*) *spec*
assumes *c* ≤ ⦃λ*s*. *P* (*snd s*)⦄, *Id* ×$_R$ *A* ⊢ *UNIV* ×$_R$ *G*, ⦃λ*v s*. *Q v* (*snd s*)⦄
shows *spec.local c* ≤ ⦃*P*⦄, *A* ⊢ *G*, ⦃*Q*⦄
unfolding *spec.local-def*
apply (*subst spec.map-invmap.galois*)
apply (*strengthen ord-to-strengthen*(*1*)[*OF assms*])
apply (*subst* (*1*) *ag.reflcl-ag*)
apply (*simp only*: *spec.invmap.ag inv-image-alt-def map-prod-snd-snd-vimage*)
apply (*subst inf.commute*)
apply (*subst heyting*[*symmetric*])
apply (*subst spec.local.qrm-def*)
apply (*subst sequential.range-proc-self*)
apply (*subst Un-commute*, *subst ag.assm-heyting*)
apply (*auto intro*: *ag.mono*)
done

lemma *localize-lift*:
fixes *A G* :: *'s rel*
fixes *P* :: *'s* ⇒ *bool*
fixes *Q* :: *'v* ⇒ *'s* ⇒ *bool*
fixes *c* :: (*sequential*, *'s*, *'v*) *spec*
notes *inf.bounded-iff*[*simp del*]
assumes *c*: *c* ≤ ⦃*P*⦄, *A* ⊢ *G*, ⦃*Q*⦄
shows *spec.localize UNIV c* ≤ ⦃λ*s*. *P* (*snd s*)⦄, *UNIV* ×$_R$ *A* ⊢ *Id* ×$_R$ *G*, ⦃λ*v s*::*'ls* × *'s*. *Q v* (*snd s*)⦄
proof(*rule ag.name-pre-state*)
fix *s* :: *'ls* × *'s* **assume** *P* (*snd s*)
show *spec.localize UNIV c* ≤ ⦃(=) *s*⦄, *UNIV* ×$_R$ *A* ⊢ *Id* ×$_R$ *G*, ⦃λ*v s*. *Q v* (*snd s*)⦄
apply (*strengthen ord-to-strengthen*[*OF c*])
apply (*simp add*: *spec.localize-def spec.invmap.ag inv-image-snd*)
apply (*simp add*: *ac-simps ag-def heyting*)

— discharge pre
apply (*subst* (*2*) *inf.commute*)
apply (*subst* (*2*) *inf.commute*)
apply (*subst inf.assoc*)
apply (*subst inf.assoc*[*symmetric*])
apply (*subst heyting.curry-conv*)
apply (*subst heyting.discharge*)

apply (*simp add*: ‹*P* (*snd s*)› *predicate1I spec.pre.mono*; *fail*)
apply (*simp add*: *ac-simps*)

— discarge assume
apply (*subst inf.assoc*[*symmetric*])
apply (*subst inf.assoc*[*symmetric*])
apply (*subst heyting.discharge*)
apply (*force intro*: *le-infI2 spec.rel.mono*)
apply (*simp add*: *ac-simps*)

— establish post
apply (*subst inf.bounded-iff*, *rule conjI*)
apply (*simp add*: *le-infI2*; *fail*)

— establish guarantee
apply (*force simp*: *inf.bounded-iff*
simp flip: *inf.assoc spec.rel.inf*
intro: *le-infI2 spec.rel.mono-reflcl*)
done
qed

setup ‹*Sign.parent-path*›

setup ‹*Sign.mandatory-path prog*›

lemma *local*:
fixes *A G* :: *'s rel*
fixes *P* :: *'s pred*
fixes *Q* :: *'v* ⇒ *'s pred*
fixes *c* :: (*'ls* × *'s*, *'v*) *prog*
assumes *prog.p2s c* ≤ ⦃λ*s*. *P* (*snd s*)⦄, *Id* ×$_R$ *A* ⊢ *UNIV* ×$_R$ *G*, ⦃λ*v s*. *Q v* (*snd s*)⦄
shows *prog.p2s* (*prog.local c*) ≤ ⦃*P*⦄, *A* ⊢ *G*, ⦃*Q*⦄
using *assms* **by** *transfer* (*rule ag.spec.local*)

lemma *localize-lift*:
fixes *A G* :: *'s rel*
fixes *P* :: *'s* ⇒ *bool*
fixes *Q* :: *'v* ⇒ *'s* ⇒ *bool*
fixes *c* :: (*'s*, *'v*) *prog*
assumes *prog.p2s c* ≤ ⦃*P*⦄, *A* ⊢ *G*, ⦃*Q*⦄
shows *prog.p2s* (*prog.localize c*) ≤ ⦃λ*s*. *P* (*snd s*)⦄, *UNIV* ×$_R$ *A* ⊢ *Id* ×$_R$ *G*, ⦃λ*v s*. *Q v* (*snd s*)⦄
using *assms* **by** *transfer* (*rule ag.spec.localize-lift*)

setup ‹*Sign.parent-path*›

setup ‹*Sign.parent-path*›

15.8 Specification inhabitation

setup ‹*Sign.mandatory-path inhabits.spec*›

lemma *localize*:
assumes *P* −*s*, *xs*→ *P'*
assumes *Id* ⊆ *r*
shows *spec.localize r P* −(*ls*, *s*), *map* (*map-prod id* (*Pair ls*)) *xs*→ *spec.localize r P'*
by (*auto intro*!: *inhabits.inf inhabits.spec.rel.rel*
simp: *spec.localize-def assms*(*1*) *trace-steps'-map subsetD*[*OF assms*(*2*)] *inhabits.spec.invmap comp-def*)

lemma *local*:
assumes $P\ -(ls,\ s),\ xs\rightarrow$ *spec.return v*
assumes *trace.steps'* $(ls,\ s)$ *xs* $\subseteq$ *spec.local.qrm*
shows *spec.local P* $-s,$ *map* (*map-prod id snd*) *xs*$\rightarrow$ *spec.return v*
unfolding *spec.local-def*
by (*rule inhabits.spec.map*[**where** *af*=*id* **and** *sf*=*snd* **and** *vf*=*id* **and** *s*=$(ls,\ s)$, *simplified*])
(*rule inhabits.inf*[*OF inhabits.spec.rel.rel-term*[*OF assms*(*2*), **where** *v*=*v*] *assms*(*1*), *simplified*])

lemma *local-init*:
assumes $P\ -(ls,\ s),\ xs\rightarrow\ P'$
assumes *trace.steps'* $(ls,\ s)$ *xs* $\subseteq$ *spec.local.qrm*
shows *spec.local-init a ls P* $-s,$ *map* (*map-prod id snd*) *xs*$\rightarrow$ *spec.local-init a* (*fst* (*trace.final'* $(ls,\ s)$ *xs*)) P'
proof −
have $\langle\!|s,$ *map* (*map-prod id snd*) *xs*, *Some* $()|\!\rangle \gg$ *spec.local-init a* (*fst* (*trace.final'* $(ls,\ s)$ *xs*)) P'
$\leq$ *spec.local-init a ls* $(\langle\!|(ls,\ s),\ xs,$ *Some* $()|\!\rangle \gg\!=\ (\lambda\text{-}.\ P'))$
proof(*induct rule*: *spec.bind-le*)
case *incomplete* **from** *assms*(*2*) **show** *?case*
by (*fastforce simp*: *spec.term.none.singleton spec.singleton.local-init-le-conv*
intro: *spec.bind.incompleteI*)
next
case (*continue* $\sigma_f\ \sigma_g\ v$)
consider (*idle*) $\langle\!|\sigma_g|\!\rangle \leq$ *spec.idle*
| (*steps*) σ' **where** $\langle\!|\sigma'|\!\rangle \leq P'$
and *trace.steps'* (*trace.init* σ') (*trace.rest* σ') $\subseteq$ *spec.local.qrm*
and $\langle\!|\sigma_g|\!\rangle \leq \langle\!|$*trace.map id snd id* $\sigma'|\!\rangle$
and *fst* (*trace.init* σ') = *fst* (*trace.final'* $(ls,\ s)$ *xs*)
using *disjE*[*OF iffD1*[*OF spec.singleton.local-init-le-conv continue*(*4*)]] **by** *metis*
then show *?case*
proof *cases*
case *idle* **with** ‹$\natural\sigma_g \neq$ *trace.T* (*trace.init* σ_g) [] *None*› **show** *?thesis*
by (*simp add*: *spec.singleton.le-conv spec.singleton.local-init-le-conv*
trace.natural-def trace.natural'.eq-Nil-conv)
next
case (*steps* σ')
let *?*σ' = *trace.T* $(ls,\ s)$ (*xs* @ *trace.rest* σ') (*trace.term* σ')
from *continue*(*1,2,3*) *steps*(*3*)
have $*$: *snd* (*trace.final'* $(ls,\ s)$ *xs*) = *snd* (*trace.init* σ')
by (*cases* σ'; *cases* σ_f)
(*clarsimp*; *metis snd-conv spec.singleton-le-conv trace.natural.sel*(*1*)
trace.final'.map trace.final'.natural' trace.less-eqE trace.t.sel(*1*))
with *steps*(*4*) **have** $*$: *trace.final'* $(ls,\ s)$ *xs* = *trace.init* σ'
by (*simp add*: *prod.expand*)
from *steps*(*1*) $*$
have $\langle\!|$*?*$\sigma'|\!\rangle \leq \langle\!|(ls,\ s),\ xs,$ *Some* $()|\!\rangle \gg P'$
by (*simp add*: *spec.bind.continueI*[*OF order.refl*])
moreover
from *assms*(*2*) *steps*(*2*) $*$
have *trace.steps ?*$\sigma' \subseteq$ *spec.local.qrm*
by (*simp add*: *trace.steps'.append*)
moreover
from *continue*(*1−3*) *steps* $*$
have $\langle\!|$*trace.init* σ_f, *trace.rest* σ_f @ *trace.rest* σ_g, *trace.term* $\sigma_g|\!\rangle \leq \langle\!|$*trace.map id snd id ?*$\sigma'|\!\rangle$
by (*auto simp*: *trace.less-eq-None spec.singleton-le-conv trace.natural-def trace.natural'.append*
cong: *trace.final'.natural'-cong*
elim: *trace.less-eqE*)
ultimately show *?thesis*
by (*simp add*: *spec.singleton.local-init-le-conv exI*[**where** *x*=*?*σ'])
qed

qed
then show *?thesis*
unfolding *inhabits-def*
by (*rule order.trans*[*OF - spec.local-init.mono*[*OF assms*(*1*)[*unfolded inhabits-def*]]])
qed

setup ‹*Sign.parent-path*›

setup ‹*Sign.mandatory-path prog.inhabits*›

lemma *localize*:
assumes *prog.p2s P* −*s*, *xs*→ *prog.p2s P′*
shows *prog.p2s* (*prog.localize P*) −(*ls*, *s*), *map* (*map-prod id* (*Pair ls*)) *xs*→ *prog.p2s* (*prog.localize P′*)
using *assms* **by** *transfer* (*rule inhabits.spec.localize*; *blast*)

lemma *local*:
assumes *prog.p2s P* −(*ls*, *s*), *xs*→ *spec.return v*
assumes *trace.steps′* (*ls*, *s*) *xs* ⊆ *spec.local.qrm*
shows *prog.p2s* (*prog.local P*) −*s*, *map* (*map-prod id snd*) *xs*→ *spec.return v*
using *assms* **by** *transfer* (*rule inhabits.spec.local*)

lemma *local-init*:
assumes *prog.p2s P* −(*ls*, *s*), *xs*→ *prog.p2s P′*
assumes *trace.steps′* (*ls*, *s*) *xs* ⊆ *spec.local.qrm*
shows *prog.p2s* (*prog.local-init ls P*) −*s*, *map* (*map-prod id snd*) *xs*→ *prog.p2s* (*prog.local-init* (*fst* (*trace.final′* (*ls*, *s*) *xs*)) *P′*)
using *assms* **by** *transfer* (*rule inhabits.spec.local-init*)

setup ‹*Sign.parent-path*›

setup ‹*Sign.parent-path*›

16 A Temporal Logic of Safety (TLS)

We model systems with finite and infinite sequences of states, closed under stuttering following Lamport (1994). This theory relates the safety logic of §8 to the powerset (quotiented by stuttering) representing properties of these sequences (see §16.6). Most of this story is standard but the addition of finite sequences does have some impact.

References:

- historical motivations for future-time linear temporal logic (LTL): Manna and Pnueli (1991); Owicki and Lamport (1982).
- a discussion on the merits of proving liveness: https://cs.nyu.edu/acsys/beyond-safety/liveness.htm

Observations:

- Lamport (and Abadi et al) treat infinite stuttering as termination
  - Lamport (2000, p189): "we can represent a terminating execution of any system by an infinite behavior that ends with a sequence of nothing but stuttering steps. We have no need of finite behaviors (finite sequences of states), so we consider only infinite ones."
  - this conflates divergence with termination
  - we separate those concepts here so we can support sequential composition
- the traditional account of liveness properties breaks down (see §24)

16.1 Stuttering

An infinitary version of *trace.natural'*.
Observations:

- we need to normalize the agent labels for sequences that infinitely stutter

Source materials:

- $ISABELLE_HOME/src/HOL/Corec_Examples/LFilter.thy.
- $AFP/Coinductive/Coinductive_List.thy
- $AFP/Coinductive/TLList.thy
- $AFP/TLA/Sequence.thy.

definition *trailing* :: *'c* ⇒ (*'a*, *'b*) *tllist* ⇒ (*'c*, *'b*) *tllist* **where**
trailing s xs = (*if tfinite xs then TNil* (*terminal xs*) *else trepeat s*)

corecursive *collapse* :: *'s* ⇒ (*'a* × *'s*, *'v*) *tllist* ⇒ (*'a* × *'s*, *'v*) *tllist* **where**
collapse s xs = (*if snd* ‘ *tset xs* ⊆ {*s*} *then trailing* (*undefined*, *s*) *xs*
else if snd (*thd xs*) = *s then collapse s* (*ttl xs*)
else TCons (*thd xs*) (*collapse* (*snd* (*thd xs*)) (*ttl xs*)))
proof −
have (*LEAST i. s* ≠ *snd* (*tnth* (*ttl xs*) *i*)) < (*LEAST i. s* ≠ *snd* (*tnth xs i*))
if ∗: ¬ *snd* ‘ *tset xs* ⊆ {*s*}
and ∗∗: *snd* (*thd xs*) = *s*
for *s* **and** *xs* :: (*'a* × *'s*, *'v*) *tllist*
proof −
from ∗ **obtain** *a s'* **where** (*a*, *s'*) ∈ *tset xs* **and** *s* ≠ *s'* **by** *fastforce*
then obtain *i* **where** *snd* (*tnth xs i*) ≠ *s*
by (*atomize-elim*, *induct rule*: *tset-induct*) (*auto intro*: *exI*[*of - 0*] *exI*[*of - Suc i* **for** *i*])
with ∗ ∗∗ **have** (*LEAST i. s* ≠ *snd* (*tnth xs i*)) = *Suc* (*LEAST i. s* ≠ *snd* (*tnth xs* (*Suc i*)))
by (*cases xs*) (*simp-all add*: *Least-Suc*[**where** *n*=*i*])
with ∗ **show** (*LEAST i. s* ≠ *snd* (*tnth* (*ttl xs*) *i*)) < (*LEAST i. s* ≠ *snd* (*tnth xs i*))
by (*cases xs*) *simp-all*
qed
then show *?thesis*
by (*relation measure* (λ(*s*, *xs*). *LEAST i. s* ≠ *snd* (*tnth xs i*)); *simp*)
qed

setup ‹*Sign.mandatory-path tmap*›

lemma *trailing*:
shows *tmap sf vf* (*trailing s xs*) = *trailing* (*sf s*) (*tmap sf vf xs*)
by (*simp add*: *trailing-def tmap-trepeat*)

setup ‹*Sign.parent-path*›

setup ‹*Sign.mandatory-path tlength*›

lemma *trailing*:
shows *tlength* (*trailing s xs*) ≤ *tlength xs*
by (*fastforce simp*: *trailing-def dest*: *not-lfinite-llength*)

setup ‹*Sign.parent-path*›

setup ‹*Sign.mandatory-path trailing*›

lemma *simps*[*simp*]:
  shows *TNil*: *trailing s* (*TNil b*) = *TNil b*
    and *TCons*: *trailing s* (*TCons x xs*) = *trailing s xs*
    and *ttl*: *ttl* (*trailing s xs*) = *trailing s xs*
    and *idempotent*: *trailing s* (*trailing s xs*) = *trailing s xs*
    and *tset-finite*: *tset* (*trailing s xs*) = (*if tfinite xs then* {} *else* {*s*})
    and *trepeat*: *trailing s* (*trepeat s*) = *trepeat s*
by (*simp-all add*: *trailing-def*)

lemma *eq-TNil-conv*:
  shows *trailing s xs* = *TNil b* ⟷ *tfinite xs* ∧ *terminal xs* = *b*
    and *TNil b* = *trailing s xs* ⟷ *tfinite xs* ∧ *terminal xs* = *b*
    and *is-TNil* (*trailing s xs*) ⟷ *tfinite xs*
by (*auto simp*: *trailing-def dest*: *is-TNil-tfinite*)

lemma *eq-TCons-conv*:
  shows *trailing s xs* = *TCons y ys* ⟷ ¬*tfinite xs* ∧ *TCons y ys* = *trepeat s*
    and *TCons y ys* = *trailing s xs* ⟷ ¬*tfinite xs* ∧ *TCons y ys* = *trepeat s*
by (*auto simp*: *trailing-def*)

lemma *tmap*:
  shows *trailing s* (*tmap sf vf xs*) = *tmap id vf* (*trailing s xs*)
by (*simp add*: *trailing-def tmap-trepeat*)

setup ‹*Sign.parent-path*›

setup ‹*Sign.mandatory-path collapse*›

lemma *unique*:
  assumes ⋀*s xs*. *f s xs* = (*if snd* ‘ *tset xs* ⊆ {*s*} *then trailing* (*undefined*, *s*) *xs*
                else if snd (*thd xs*) = *s then f s* (*ttl xs*)
                else TCons (*thd xs*) (*f* (*snd* (*thd xs*)) (*ttl xs*)))
  shows *f* = *collapse*
proof(*intro ext*)
  show *f s xs* = *collapse s xs* **for** *s xs*
  proof(*coinduction arbitrary*: *s xs*)
    case (*Eq-tllist s xs*) **show** *?case*
      apply (*induct arg*≡(*s*, *xs*) *arbitrary*: *s xs rule*: *collapse.inner-induct*)
      apply (*subst* (*1 2 3*) *assms*)
      apply (*subst* (*1 2 3*) *collapse.code*)
      apply *simp*
      apply (*subst* (*1 2 3*) *assms*)
      apply (*subst* (*1 2 3*) *collapse.code*)
      apply *simp*
      apply (*metis assms collapse.code*)
      done
  qed
qed

lemma *collapse*:
  shows *collapse s* (*collapse s xs*) = *collapse s xs*
proof −
  have (λ*s xs*. *collapse s* (*collapse s xs*)) = *collapse*
    apply (*rule collapse.unique*)
    apply (*subst* (*1 2 3*) *collapse.code*)
    apply *auto*
    done

then show *?thesis*
by (*fastforce simp*: *fun-eq-iff*)
qed

lemma *simps*[*simp*]:
shows *TNil*: *collapse s* (*TNil b*) = *TNil b*
and *TCons*: *collapse s* (*TCons x xs*) = (*if snd x* = *s then collapse s xs else TCons x* (*collapse* (*snd x*) *xs*))
and *trailing*: *collapse s* (*trailing* (*undefined*, *s*) *xs*) = *trailing* (*undefined*, *s*) *xs*
by (*simp-all add*: *collapse.code trailing-def*)

lemma *tshift-stuttering*:
assumes *snd* ‘ *set xs* ⊆ {*s*}
shows *collapse s* (*tshift xs ys*) = *collapse s ys*
using *assms* **by** (*induct xs*) *simp-all*

lemma *infinite-trailing*:
assumes ¬*tfinite xs*
assumes *snd* ‘ *tset xs* ⊆ {*s′*}
shows *collapse s xs* = (*if s* = *s′ then trepeat* (*undefined*, *s′*) *else TCons* (*thd xs*) (*trepeat* (*undefined*, *s′*)))
using *assms* **by** (*cases xs*) (*simp-all add*: *assms collapse.code trailing-def*)

lemma *eq-TNil-conv*:
shows *collapse s xs* = *TNil b* ⟷ *tfinite xs* ∧ *snd* ‘ *tset xs* ⊆ {*s*} ∧ *terminal xs* = *b* (**is** *?lhs* ⟷ *?rhs*)
and *TNil b* = *collapse s xs* ⟷ *tfinite xs* ∧ *snd* ‘ *tset xs* ⊆ {*s*} ∧ *terminal xs* = *b* (**is** *?thesis1*)
proof −
show *?lhs* ⟷ *?rhs*
proof(*rule iffI*)
show *?lhs* ⟹ *?rhs*
proof(*induct arg*≡(*s*, *xs*) *arbitrary*: *s xs rule*: *collapse.inner-induct*[*case-names step*])
case (*step s xs*) **then show** *?case*
by (*cases xs*; *clarsimp split*: *if-splits*)
(*subst* (*asm*) *collapse.code*; *clarsimp simp*: *trailing.eq-TNil-conv split*: *if-splits*)
qed
show *?rhs* ⟹ *?lhs*
by (*simp add*: *conj-explode*) (*induct arbitrary*: *s rule*: *tfinite-induct*; *simp*)
qed
then show *?thesis1*
by (*rule eq-commute-conv*)
qed

lemma *is-TNil-conv*:
shows *is-TNil* (*collapse s xs*) ⟷ *tfinite xs* ∧ *snd* ‘ *tset xs* ⊆ {*s*} (**is** *?thesis2*)
by (*simp add*: *is-TNil-def collapse.eq-TNil-conv*)

lemma *eq-TConsE*:
assumes *collapse s xs* = *TCons y ys*
obtains
(*trailing-stuttering*) ¬ *tfinite xs*
and *snd* ‘ *tset xs* = {*s*}
and *TCons y ys* = *trepeat* (*undefined*, *s*)
| (*step*) *us ys′* **where** *xs* = *tshift us* (*TCons y ys′*)
and *snd* ‘ *set us* ⊆ {*s*}
and *snd y* ≠ *s*
and *collapse* (*snd y*) *ys′* = *ys*
apply *atomize-elim*
using *assms*
proof(*induct arg*≡(*s*, *xs*) *arbitrary*: *s xs rule*: *collapse.inner-induct*[*case-names step*])
case (*step s xs*) **show** *?case*

```
  proof(cases xs)
    case (TNil v) with step.prems show ?thesis by simp
  next
    case (TCons x xs') show ?thesis
    proof(cases snd ‘ tset xs' ⊆ {snd x})
      case True with TCons trans[OF collapse.code[symmetric] step.prems] show ?thesis
        by (force simp: trailing.eq-TCons-conv tshift-eq-TCons-conv split: if-split-asm)
    next
      case False with TCons trans[OF collapse.code[symmetric] step.prems] step.hyps[OF refl]
      show ?thesis
        by (cases x, cases y)
          (simp add: trailing.eq-TCons-conv tshift-eq-TCons-conv trepeat-eq-TCons-conv
                     eq-snd-iff exI[where x=[]]
              split: if-split-asm; safe; force dest!: spec[where x=(fst x, s) # us for us])
    qed
  qed
qed

lemma eq-TCons-conv:
  shows collapse s xs = TCons y ys
    ⟷ (¬tfinite xs ∧ snd ‘ tset xs = {s} ∧ TCons y ys = trepeat (undefined, s))
      ∨ (∃xs' ys'. xs = tshift xs' (TCons y ys') ∧ snd ‘ set xs' ⊆ {s} ∧ snd y ≠ s ∧ collapse (snd y) ys' = ys) (is
?lhs ⟷ ?rhs)
    and TCons y ys = collapse s xs
    ⟷ (¬tfinite xs ∧ snd ‘ tset xs = {s} ∧ TCons y ys = trepeat (undefined, s))
      ∨ (∃xs' ys'. xs = tshift xs' (TCons y ys') ∧ snd ‘ set xs' ⊆ {s} ∧ snd y ≠ s ∧ collapse (snd y) ys' = ys) (is
?thesis1)
proof −
  show ?lhs ⟷ ?rhs
    by (auto elim: collapse.eq-TConsE simp: collapse.tshift-stuttering collapse.infinite-trailing)
  then show ?thesis1
    by (rule eq-commute-conv)
qed

lemma tfinite:
  shows tfinite (collapse s xs) ⟷ tfinite xs (is ?lhs ⟷ ?rhs)
proof(rule iffI)
  show ?lhs if ?rhs
    using that by (induct arbitrary: s rule: tfinite-induct) simp-all
  show ?rhs if ?lhs
    using that by (induct collapse s xs arbitrary: s xs rule: tfinite-induct)
                  (auto simp: collapse.eq-TNil-conv collapse.eq-TCons-conv trepeat-eq-TCons-conv)
qed

lemma tfinite-conv:
  assumes collapse s xs = collapse s' xs'
  shows tfinite xs ⟷ tfinite xs'
by (metis assms collapse.tfinite)

lemma terminal:
  shows terminal (collapse s xs) = terminal xs
proof(cases tfinite xs)
  case True
  then obtain i where tlength xs ≤ enat i
    using llength-eq-infty-conv-lfinite by fastforce
  then show ?thesis
  proof(induct i arbitrary: s xs)
    case (Suc i s xs) then show ?case
```

by (*cases xs*) (*simp-all flip*: *eSuc-enat*)
qed (*clarsimp simp*: *enat-0 tlength-0-conv*)
qed (*simp add*: *collapse.tfinite terminal-tinfinite*)

lemma *tlength*:
shows *tlength* (*collapse s xs*) ≤ *tlength xs*
proof(*cases tfinite xs*)
case *True* **then show** *?thesis*
by (*induct arbitrary*: *s rule*: *tfinite-induct*) (*auto intro*: *order.trans*[*OF - ile-eSuc*])
next
case *False* **then show** *?thesis*
by (*fastforce dest*: *not-lfinite-llength*)
qed

lemma *tset-memberD*:
assumes (*a*, *s'*) ∈ *tset* (*collapse s xs*)
shows *s'* ∈ *snd* ‘ *tset xs*
using *assms*
by (*induct collapse s xs arbitrary*: *s xs rule*: *tset-induct*)
(*auto simp*: *collapse.eq-TCons-conv trepeat-eq-TCons-conv tset-tshift image-Un*)

lemma *tset-memberD2*:
assumes (*a*, *s'*) ∈ *tset xs*
shows *s* = *s'* ∨ *s'* ∈ *snd* ‘ *tset* (*collapse s xs*)
using *assms* **by** (*induct xs arbitrary*: *a s rule*: *tset-induct*; *simp*; *fast*)

lemma *tshift*:
shows *collapse s* (*tshift xs ys*) = *tshift* (*trace.natural' s xs*) (*collapse* (*trace.final' s xs*) *ys*)
by (*induct xs arbitrary*: *s*) *simp-all*

lemma *trepeat*:
shows *collapse s* (*trepeat* (*a*, *s*)) = *trepeat* (*undefined*, *s*)
by (*subst collapse.code*) (*simp add*: *trailing-def*)

lemma *eq-trepeat-conv*:
shows *trepeat* (*undefined*, *s*) = *collapse s xs* ⟷ ¬*tfinite xs* ∧ *snd* ‘ *tset xs* = {*s*} (**is** *?thesis1*)
and *collapse s xs* = *trepeat* (*undefined*, *s*) ⟷ ¬*tfinite xs* ∧ *snd* ‘ *tset xs* = {*s*} (**is** *?thesis2*)
proof −
show *?thesis1*
by (*rule iffI*,
(*subst* (*asm*) *trepeat-unfold*, *simp add*: *collapse.eq-TCons-conv*),
simp add: *collapse.infinite-trailing*)
then show *?thesis2*
by (*rule eq-commute-conv*)
qed

lemma *treplicate*:
shows *collapse s* (*treplicate i* (*a*, *s*) *v*) = *TNil v*
by (*subst collapse.code*) (*simp add*: *trailing.eq-TNil-conv split*: *nat.split*)

lemma *eq-tshift-conv*:
shows *collapse s xs* = *tshift ys zs*
⟷ (∃ *xs' xs'' ys'*. *tshift xs' xs''* = *xs* ∧ *trace.natural' s xs'* @ *ys'* = *ys*
∧ ((¬*tfinite xs''* ∧ *snd* ‘ *tset xs''* = {*trace.final' s xs'*} ∧ *tshift ys' zs* = *trepeat* (*undefined*, *trace.final' s xs'*))
∨ (*ys'* = [] ∧ *collapse* (*trace.final' s xs'*) *xs''* = *zs*))) (**is** *?lhs* ⟷ *?rhs*)
and *tshift ys zs* = *collapse s xs*
⟷ (∃ *xs' xs'' ys'*. *tshift xs' xs''* = *xs* ∧ *trace.natural' s xs'* @ *ys'* = *ys*

```
        ∧ ((¬tfinite xs'' ∧ snd ‘ tset xs'' = {trace.final' s xs'} ∧ tshift ys' zs = trepeat (undefined, trace.final' s
xs'))
         ∨ (ys' = [] ∧ collapse (trace.final' s xs') xs'' = zs))) (is ?thesis1)
proof −
  show ?lhs ⟷ ?rhs
  proof(rule iffI)
    show ?lhs ⟹ ?rhs
    proof(induct ys arbitrary: s xs)
      case Nil then show ?case
        by (simp add: exI[where x=[]])
    next
      case (Cons y ys s xs)
      from Cons.prems[simplified] show ?case
      proof(cases rule: collapse.eq-TConsE)
        case trailing-stuttering then show ?thesis
          by (simp add: exI[where x=[]])
      next
        case (step xs' ys')
        from step(1−3) Cons.hyps[OF step(4)] show ?thesis
          by (fastforce simp: trace.natural'.append tshift-append
                    simp flip: trace.natural'.eq-Nil-conv
                        intro: exI[where x=xs' @ y # ys'' for ys''])
      qed
    qed
    show ?rhs ⟹ ?lhs
      by (auto simp: collapse.tshift tshift-append collapse.infinite-trailing)
  qed
  then show ?thesis1
    by (rule eq-commute-conv)
qed

lemma eq-collapse-ttake-dropn-conv:
  shows collapse s xs = collapse s ys
    ⟷ (∃j. trace.natural' s (fst (ttake i xs)) = trace.natural' s (fst (ttake j ys))
          ∧ snd (ttake i xs) = snd (ttake j ys)
          ∧ collapse (trace.final' s (fst (ttake i xs))) (tdropn i xs)
          = collapse (trace.final' s (fst (ttake i xs))) (tdropn j ys)) (is ?lhs ⟷ (∃j. ?rhs i j s xs ys))
proof(rule iffI)
  show ?lhs ⟹ (∃j. ?rhs i j s xs ys)
  proof(induct i arbitrary: s xs ys)
    case (Suc i s xs ys) show ?case
    proof(cases xs)
      case (TNil b) with Suc.prems show ?thesis
        by (fastforce intro: exI[where x=case tlength ys of ∞ ⇒ undefined | enat j ⇒ Suc j]
                    simp: collapse.eq-TNil-conv trace.natural'.eq-Nil-conv
                          ttake-eq-Some-conv tfinite-tlength-conv tdropn-tlength
                    dest: in-set-ttakeD)
    next
      case (TCons x xs') show ?thesis
      proof(cases snd x = s)
        case True with Suc TCons show ?thesis by simp
      next
        case False
        note Suc.prems TCons False
        moreover from calculation
        obtain us ys'
          where ys = tshift us (TCons x ys')
            and snd ‘ set us ⊆ {s}
```

```
        and collapse (snd x) ys' = collapse (snd x) xs'
      by (auto simp: collapse.eq-TCons-conv trepeat-eq-TCons-conv)
    moreover from calculation Suc.hyps[of snd x xs' ys']
    obtain j where ?rhs i j (snd x) xs' ys'
      by presburger
    ultimately show ?thesis
      by (auto simp: ttake-tshift trace.natural'.append tdropn-tshift
          simp flip: trace.natural'.eq-Nil-conv
              intro: exI[where x=Suc (length us) + j])
  qed
 qed
 qed (simp add: exI[where x=0])
 show ∃j. ?rhs i j s xs ys ⟹ ?lhs
  by (metis collapse.tshift trace.final'.natural' tshift-fst-ttake-tdropn-id)
qed

lemmas eq-collapse-ttake-dropnE = exE[OF iffD1[OF collapse.eq-collapse-ttake-dropn-conv]]

lemma tshift-tdropn:
 assumes trace.natural' s (fst (ttake i xs)) = trace.natural' s ys
 shows collapse s (tshift ys (tdropn i xs)) = collapse s xs
by (metis assms collapse.tshift trace.final'.natural' tshift-fst-ttake-tdropn-id)

lemma map-collapse:
 shows collapse (sf s) (tmap (map-prod af sf) vf (collapse s xs))
     = collapse (sf s) (tmap (map-prod af sf) vf xs) (is ?lhs s xs = ?rhs s xs)
proof(coinduction arbitrary: s xs)
 case (Eq-tllist s xs) show ?case
 proof(intro conjI; (intro impI)?)
  have ∗: sf s' = sf s
   if tfinite xs and sf ‘ snd ‘ tset (collapse s xs) ⊆ {sf s} and (a, s') ∈ tset xs
  for a s s'
   using that by (induct arbitrary: s rule: tfinite-induct; clarsimp split: if-split-asm; metis)
  show is-TNil (?lhs s xs) ⟷ is-TNil (?rhs s xs)
   by (rule iffI,
       fastforce dest!: ∗ simp: collapse.is-TNil-conv collapse.tfinite tllist.set-map snd-image-map-prod,
       fastforce dest!: collapse.tset-memberD simp: collapse.is-TNil-conv collapse.tfinite tllist.set-map)
  show terminal (?lhs s xs) = terminal (?rhs s xs)
   if is-TNil (?lhs s xs) and is-TNil (?rhs s xs)
   using that by (simp add: collapse.is-TNil-conv collapse.terminal)
  assume ¬is-TNil (?lhs s xs) and ¬is-TNil (?rhs s xs)
  then obtain y ys z zs where l: ?lhs s xs = TCons y ys and r: ?rhs s xs = TCons z zs
   by (simp add: tllist.disc-eq-case(2) split: tllist.split-asm)
  from l show thd (?lhs s xs) = thd (?rhs s xs)
         ∧ (∃s' xs'. ttl (?lhs s xs) = ?lhs s' xs' ∧ ttl (?rhs s xs) = ?rhs s' xs')
  proof(cases rule: collapse.eq-TConsE)
   case trailing-stuttering
   note left = this
   from r show ?thesis
   proof(cases rule: collapse.eq-TConsE)
    case trailing-stuttering
    from left(3) trailing-stuttering(3) show ?thesis
     by (fold l r) (simp; metis)
   next
    case (step us zs')
    from left(2) step(1,3) have False
     by (clarsimp simp: tset-tshift tset-tmap tmap-eq-tshift-conv TCons-eq-tmap-conv collapse.tshift
                 split: if-split-asm)
```

```
        (use step(2) in ‹fastforce simp flip: trace.final'.map[where af=af]›)
    then show ?thesis ..
  qed
 next
  case (step us ys')
  note left = this
  from r show ?thesis
  proof(cases rule: collapse.eq-TConsE)
   case trailing-stuttering
   have False
    if sf s' ≠ sf s
    and (λx. sf (snd x)) ‘ tset xs = {sf s}
    and (λx. sf (snd x)) ‘ set us ⊆ {sf s}
    and collapse s xs = tshift us (TCons (a, s') vs)
    for a s' us vs
    using that
    by (force simp: tset-tshift
          dest!: arg-cong[where f=λxs. s' ∈ snd ‘ tset xs] collapse.tset-memberD
          intro: imageI[where f=λx. sf (snd x)])
   with l left(3) trailing-stuttering(2) have False
    by (fastforce simp: tset-tmap tmap-eq-tshift-conv TCons-eq-tmap-conv collapse.eq-TCons-conv
                  trepeat-eq-TCons-conv snd-image-map-prod image-image)
   then show ?thesis ..
  next
   case (step vs zs')
   from left step show ?thesis
    unfolding l r
    apply (clarsimp simp: tmap-eq-tshift-conv collapse.tshift TCons-eq-tmap-conv
                    tmap-tshift trace.natural'.map-natural'[where af=af and sf=sf and s=s]
                    iffD2[OF trace.natural'.eq-Nil-conv(1)]
                dest!: arg-cong[where f=λxs. collapse (sf s) (tmap (map-prod af sf) vf xs)]
                split: if-split-asm)
     apply (use step(2) in ‹fastforce simp flip: trace.final'.map[where af=af]›)
    apply (metis list.set-map trace.final'.idle trace.final'.map trace.final'.natural')
    apply metis
    done
  qed
 qed
qed

setup ‹Sign.parent-path›

setup ‹Sign.mandatory-path behavior›

definition natural :: ('a, 's, 'v) behavior.t ⇒ ('a, 's, 'v) behavior.t (‹♮T›) where
 ♮Tω = behavior.B (behavior.init ω) (collapse (behavior.init ω) (behavior.rest ω))

setup ‹Sign.mandatory-path sset›

lemma collapse[simp]:
 shows behavior.sset (behavior.B s (collapse s xs)) = behavior.sset (behavior.B s xs)
by (auto simp: behavior.sset.simps collapse.tset-memberD dest: collapse.tset-memberD2[where s=s])

lemma natural[simp]:
 shows behavior.sset (♮Tω) = behavior.sset ω
by (simp add: behavior.natural-def)
```

lemma *continue*:
  shows *behavior.sset* (σ @$-_B$ *xs*) = *trace.sset* σ $\cup$ (*case trace.term* σ *of None* $\Rightarrow$ *snd* ‘ *tset xs* | *Some* - $\Rightarrow$ {})
by (*cases* σ)
  (*simp add*: *behavior.sset.simps behavior.continue-def tshift2-def tset-tshift image-Un trace.sset.simps*
    split: *option.split*)

setup ‹*Sign.parent-path*›

setup ‹*Sign.mandatory-path natural*›

lemma *sel*[*simp*]:
  shows *behavior.init* ($\natural_T\omega$) = *behavior.init* ω
    and *behavior.rest* ($\natural_T\omega$) = *collapse* (*behavior.init* ω) (*behavior.rest* ω)
by (*simp-all add*: *behavior.natural-def*)

lemma *TNil*:
  shows $\natural_T$(*behavior.B s* (*TNil v*)) = *behavior.B s* (*TNil v*)
by (*simp add*: *behavior.natural-def*)

lemma *tfinite*:
  shows *tfinite* (*behavior.rest* ($\natural_T$ ω)) $\longleftrightarrow$ *tfinite* (*behavior.rest* ω)
by (*simp add*: *behavior.natural-def collapse.tfinite*)

lemma *continue*:
  shows $\natural_T$(σ @$-_B$ *xs*) = $\natural\sigma$ @$-_B$ (*collapse* (*trace.final* σ) *xs*)
by (*simp add*: *behavior.t.expand tshift2-def collapse.tshift split*: *option.split*)

lemma *tshift*:
  shows $\natural_T$(*behavior.B s* (*tshift as xs*)) = *behavior.B s* (*collapse s* (*tshift as xs*))
by (*simp add*: *behavior.natural-def*)

lemma *trepeat*:
  shows $\natural_T$(*behavior.B s* (*trepeat* (*a*, *s*))) = *behavior.B s* (*trepeat* (*undefined*, *s*))
by (*simp add*: *behavior.natural-def collapse.trepeat*)

lemma *treplicate*:
  shows $\natural_T$(*behavior.B s* (*treplicate i* (*a*, *s*) *v*)) = *behavior.B s* (*TNil v*)
by (*simp add*: *behavior.natural-def collapse.treplicate*)

lemma *map-natural*:
  shows $\natural_T$(*behavior.map af sf vf* ($\natural_T\omega$)) = $\natural_T$(*behavior.map af sf vf* ω)
by (*simp add*: *behavior.natural-def collapse.map-collapse*)

lemma *idle*:
  assumes *behavior.sset* ω $\subseteq$ {*behavior.init* ω}
  shows $\natural_T\omega$ = *behavior.B* (*behavior.init* ω) (*trailing* (*undefined*, *behavior.init* ω) (*behavior.rest* ω))
using *assms* **by** (*cases* ω) (*simp add*: *behavior.natural-def behavior.sset.simps collapse.code*)

setup ‹*Sign.parent-path*›

interpretation *stuttering*: *galois.image-vimage-idempotent* $\natural_T$
by *standard* (*simp add*: *behavior.natural-def collapse.collapse*)

setup ‹*Sign.mandatory-path stuttering*›

setup ‹*Sign.mandatory-path equiv*›

abbreviation *syn* :: (*'a*, *'s*, *'v*) *behavior.t* $\Rightarrow$ (*'a*, *'s*, *'v*) *behavior.t* $\Rightarrow$ *bool* (**infix** ‹$\simeq_T$› *50*) **where**

$\omega_1 \simeq_T \omega_2 \equiv$ *behavior.stuttering.equivalent* $\omega_1\ \omega_2$

lemma *map*:
assumes $\omega_1 \simeq_T \omega_2$
shows *behavior.map af sf vf* $\omega_1 \simeq_T$ *behavior.map af sf vf* ω_2
by (*metis assms behavior.natural.map-natural*)

lemma *takeE*:
assumes $\omega_1 \simeq_T \omega_2$
obtains *j* **where** *behavior.take i* $\omega_1 \simeq_S$ *behavior.take j* ω_2
using *assms*
by (*fastforce simp*: *behavior.natural-def trace.natural-def*
elim: *collapse.eq-collapse-ttake-dropnE*[**where** *s*=*behavior.init* ω_2 **and** *i*=*i* **and** *xs*=*behavior.rest* ω_1
and *ys*=*behavior.rest* ω_2])

lemma *idle-dropn*:
assumes *behavior.dropn i* ω = *Some* ω'
assumes *behavior.sset* $\omega \subseteq \{$*behavior.init* $\omega\}$
shows $\omega \simeq_T \omega'$
proof −
from *behavior.sset.dropn-le*[*OF assms*(*1*)] *assms*(*2*)
have *behavior.sset* $\omega' \subseteq \{$*behavior.init* $\omega'\}$ **and** *behavior.init* ω' = *behavior.init* ω
using *behavior.t.set-sel*(*2*) *subset-singletonD* **by** *fastforce*+
from *assms*(*1*) *behavior.natural.idle*[*OF assms*(*2*)] *behavior.natural.idle*[*OF this*(*1*)] *this*(*2*)
show *?thesis*
by (*simp add*: *trailing-def*)
(*metis behavior.dropn.tfiniteD behavior.dropn.eq-Some-tdropnD terminal-tdropn*)
qed

setup ‹*Sign.parent-path*›

setup ‹*Sign.parent-path*›

setup ‹*Sign.parent-path*›

setup ‹*Sign.mandatory-path trace.stuttering.equiv.behavior*›

lemma *takeE*:
fixes σ :: ('*a*, '*s*, '*v*) *trace.t*
assumes *behavior.take i* $\omega \simeq_S \sigma$
obtains ω' *j* **where** $\omega \simeq_T \omega'$ **and** σ = *behavior.take j* ω'
proof *atomize-elim*
have $\exists$ *ys j. collapse s xs = collapse s ys* $\wedge$ *trace.T s xs'* (*snd* (*ttake i xs*)) = *behavior.take j* (*behavior.B s ys*)
if *trace.natural' s* (*fst* (*ttake i xs*)) = *trace.natural' s xs'*
for *s xs'* **and** *xs* :: ('*a* × '*s*, '*v*) *tllist*
using *that*
by (*cases snd* (*ttake i xs*))
(*fastforce simp*: *behavior.take.tshift ttake-eq-Some-conv tdropn-tlength*
trace.take.all trace.take.all-iff
intro: *exI*[**where** *x*=*tshift xs'* (*tdropn i xs*)]
exI[**where** *x*=*length xs'*] *exI*[**where** *x*=*Suc* (*length xs'*)]
dest: *collapse.tshift-tdropn*)+
with *assms* **show** $\exists \omega'$ *j.* $\omega \simeq_T \omega' \wedge \sigma$ = *behavior.take j* ω'
by (*cases* σ)
(*clarsimp simp*: *behavior.natural-def trace.natural-def behavior.split-Ex*)
qed

lemmas *rev-takeE* = *trace.stuttering.equiv.behavior.takeE*[*OF sym*]

setup ‹*Sign.parent-path*›

setup ‹*Sign.mandatory-path trace.natural.behavior*›

lemma *takeE*:
  fixes ω :: ('*a*, '*s*, '*v*) *behavior.t*
  obtains *j* **where** ♮(*behavior.take i* ω) = *behavior.take j* (♮$_T\omega$)
proof *atomize-elim*
  have ∃*j*. *trace.natural'* *s* (*fst* (*ttake i xs*)) = *fst* (*ttake j* (*collapse s xs*))
        ∧ *snd* (*ttake i xs*) = *snd* (*ttake j* (*collapse s xs*))
   for *s* **and** *xs* :: ('*a* × '*s*, '*v*) *tllist*
  proof(*induct i arbitrary*: *s xs*)
    case *0* **show** *?case* **by** (*fastforce simp*: *ttake-eq-Nil-conv*)
  next
    case (*Suc i s xs*) **show** *?case*
    proof(*cases xs*)
      case (*TCons x' xs'*) **with** *Suc*[**where** *s*=*snd x'* **and** *xs*=*xs'*] **show** *?thesis*
        by (*fastforce intro*: *exI*[**where** *x*=*Suc j* **for** *j*])
    qed (*simp add*: *exI*[**where** *x*=*1*])
  qed
  then show ∃*j*. ♮ (*behavior.take i* ω) = *behavior.take j* (♮$_T$ ω)
    by (*simp add*: *behavior.take-def trace.natural-def split-def*)
qed

setup ‹*Sign.parent-path*›

16.2 The *('a, 's, 'v) tls* lattice

This is our version of Lamport's TLA lattice which we treat in a "semantic" way similarly to Abadi and Merz (1996).

Observations:

- there is a somewhat natural partial ordering on the *tls* lattice induced by the connection with the *spec* lattice (see §16.6 and §24) which we do not use

typedef ('*a*, '*s*, '*v*) *tls* = *behavior.stuttering.closed* :: ('*a*, '*s*, '*v*) *behavior.t set set*
morphisms *unTLS TLS*
by *blast*

setup-lifting *type-definition-tls*

instantiation *tls* :: (*type*, *type*, *type*) *complete-boolean-algebra*
begin

lift-definition *bot-tls* :: ('*a*, '*s*, '*v*) *tls* **is** *empty* **..**
lift-definition *top-tls* :: ('*a*, '*s*, '*v*) *tls* **is** *UNIV* **..**
lift-definition *sup-tls* :: ('*a*, '*s*, '*v*) *tls* ⇒ ('*a*, '*s*, '*v*) *tls* ⇒ ('*a*, '*s*, '*v*) *tls* **is** *sup* **..**
lift-definition *inf-tls* :: ('*a*, '*s*, '*v*) *tls* ⇒ ('*a*, '*s*, '*v*) *tls* ⇒ ('*a*, '*s*, '*v*) *tls* **is** *inf* **..**
lift-definition *less-eq-tls* :: ('*a*, '*s*, '*v*) *tls* ⇒ ('*a*, '*s*, '*v*) *tls* ⇒ *bool* **is** *less-eq* **.**
lift-definition *less-tls* :: ('*a*, '*s*, '*v*) *tls* ⇒ ('*a*, '*s*, '*v*) *tls* ⇒ *bool* **is** *less* **.**
lift-definition *Inf-tls* :: ('*a*, '*s*, '*v*) *tls set* ⇒ ('*a*, '*s*, '*v*) *tls* **is** *Inf* **..**
lift-definition *Sup-tls* :: ('*a*, '*s*, '*v*) *tls set* ⇒ ('*a*, '*s*, '*v*) *tls* **is** λ*X*. *Sup X* ⊔ *behavior.stuttering.cl* {} **..**
lift-definition *minus-tls* :: ('*a*, '*s*, '*v*) *tls* ⇒ ('*a*, '*s*, '*v*) *tls* ⇒ ('*a*, '*s*, '*v*) *tls* **is** *minus* **..**
lift-definition *uminus-tls* :: ('*a*, '*s*, '*v*) *tls* ⇒ ('*a*, '*s*, '*v*) *tls* **is** *uminus* **..**

instance
by (*standard*; *transfer*;

auto simp: *behavior.stuttering.cl-bot*
behavior.stuttering.closed-strict-complete-distrib-lattice-axiomI[*OF behavior.stuttering.cl-bot*])

end

declare
SUPE[**where** *'a*=(*'a*, *'s*, *'v*) *tls*, *intro*!]
SupE[**where** *'a*=(*'a*, *'s*, *'v*) *tls*, *intro*!]
Sup-le-iff[**where** *'a*=(*'a*, *'s*, *'v*) *tls*, *simp*]
SupI[**where** *'a*=(*'a*, *'s*, *'v*) *tls*, *intro*]
SUPI[**where** *'a*=(*'a*, *'s*, *'v*) *tls*, *intro*]
rev-SUPI[**where** *'a*=(*'a*, *'s*, *'v*) *tls*, *intro?*]
INFE[**where** *'a*=(*'a*, *'s*, *'v*) *tls*, *intro*]

setup ‹*Sign.mandatory-path tls*›

lemma *boolean-implication-transfer*[*transfer-rule*]:
shows *rel-fun* (*pcr-tls* (=) (=) (=)) (*rel-fun* (*pcr-tls* (=) (=) (=)) (*pcr-tls* (=) (=) (=))) ($\longrightarrow_B$) ($\longrightarrow_B$)
unfolding *boolean-implication-def* **by** *transfer-prover*

lemma *bot-not-top*:
shows $\bot \neq (\top :: ('a, 's, 'v)\ tls)$
by *transfer simp*

setup ‹*Sign.parent-path*›

16.3 Irreducible elements

setup ‹*Sign.mandatory-path raw*›

definition *singleton* :: (*'a*, *'s*, *'v*) *behavior.t* ⇒ (*'a*, *'s*, *'v*) *behavior.t set* **where**
singleton ω = *behavior.stuttering.cl* {ω}

lemma *singleton-le-conv*:
shows *raw.singleton* $\sigma_1 \leq$ *raw.singleton* $\sigma_2 \longleftrightarrow \natural_T\sigma_1 = \natural_T\sigma_2$
by (*rule iffI*; *fastforce simp*: *raw.singleton-def simp flip*: *behavior.stuttering.cl-axiom*
dest: *behavior.stuttering.clE behavior.stuttering.equiv-cl-singleton*)

setup ‹*Sign.parent-path*›

setup ‹*Sign.mandatory-path tls*›

lift-definition *singleton* :: (*'a*, *'s*, *'v*) *behavior.t* ⇒ (*'a*, *'s*, *'v*) *tls* (‹⦇-⦈$_T$› [*0*]) **is** *raw.singleton*
by (*simp add*: *raw.singleton-def*)

abbreviation *singleton-behavior-syn* :: *'s* ⇒ (*'a* × *'s*, *'v*) *tllist* ⇒ (*'a*, *'s*, *'v*) *tls* (‹⦇-, -⦈$_T$› [*0*, *0*]) **where**
⦇*s*, *xs*⦈$_T$ ≡ ⦇*behavior.B s xs*⦈$_T$

setup ‹*Sign.mandatory-path singleton*›

lemma *Sup-prime*:
shows *Sup-prime* ⦇ω⦈$_T$
by (*clarsimp simp*: *Sup-prime-on-def*)
(*transfer*; *auto simp*: *raw.singleton-def behavior.stuttering.cl-bot*
elim!: *Sup-prime-onE*[*OF behavior.stuttering.Sup-prime-on-singleton*])

lemma *nchotomy*:
shows ∃ *X*∈*behavior.stuttering.closed*. *x* = ⨆(*tls.singleton* ‘ *X*)

by *transfer*
(*use behavior.stuttering.closed-conv* **in** ‹*auto simp add*: *raw.singleton-def*
simp flip: *behavior.stuttering.distributive*›)

lemmas *exhaust* = *bexE*[*OF tls.singleton.nchotomy*]

lemma *collapse*[*simp*]:
shows $\bigsqcup$(*tls.singleton* ‘ {ω. $\langle\!|\omega|\!\rangle_T \leq P$}) = P
by (*rule tls.singleton.exhaust*[*of P*]) (*simp add*: *antisym SUP-le-iff SUP-upper*)

lemmas *not-bot* = *Sup-prime-not-bot*[*OF tls.singleton.Sup-prime*] — Non-triviality

setup ‹*Sign.parent-path*›

lemma *singleton-le-ext-conv*:
shows $P \leq Q \longleftrightarrow (\forall \omega.\ \langle\!|\omega|\!\rangle_T \leq P \longrightarrow \langle\!|\omega|\!\rangle_T \leq Q)$ (**is** *?lhs* $\longleftrightarrow$ *?rhs*)
proof(*rule iffI*)
show *?rhs* $\Longrightarrow$ *?lhs*
by (*rule tls.singleton.exhaust*[**where** *x*=*P*]; *rule tls.singleton.exhaust*[**where** *x*=*Q*]; *blast*)
qed *fastforce*

lemmas *singleton-le-conv* = *raw.singleton-le-conv*[*transferred*]
lemmas *singleton-le-extI* = *iffD2*[*OF tls.singleton-le-ext-conv*, *rule-format*]

lemma *singleton-eq-conv*[*simp*]:
shows $\langle\!|\omega|\!\rangle_T = \langle\!|\omega'|\!\rangle_T \longleftrightarrow \omega \simeq_T \omega'$
using *tls.singleton-le-conv* **by** (*force intro*: *antisym*)

lemma *singleton-cong*:
assumes $\omega \simeq_T \omega'$
shows $\langle\!|\omega|\!\rangle_T = \langle\!|\omega'|\!\rangle_T$
using *assms* **by** *simp*

setup ‹*Sign.mandatory-path singleton*›

named-theorems *le-conv* ‹ *simplification rules for* ‹$\langle\!|\sigma|\!\rangle_T \leq$ *const* ...› ›

lemma *boolean-implication-le-conv*[*tls.singleton.le-conv*]:
shows $\langle\!|\sigma|\!\rangle_T \leq P \longrightarrow_B Q \longleftrightarrow (\langle\!|\sigma|\!\rangle_T \leq P \longrightarrow \langle\!|\sigma|\!\rangle_T \leq Q)$
by *transfer*
(*auto simp*: *raw.singleton-def boolean-implication.set-alt-def*
elim!: *behavior.stuttering.clE behavior.stuttering.closed-in*[*OF - sym*])

lemmas *antisym* = *antisym*[*OF tls.singleton-le-extI tls.singleton-le-extI*]

lemmas *top* = *tls.singleton.collapse*[*of* $\top$, *simplified*, *symmetric*]

lemma *simps*[*simp*]:
shows $\langle\!|\natural_T\omega|\!\rangle_T = \langle\!|\omega|\!\rangle_T$
and $\langle\!|s, xs|\!\rangle_T \leq \langle\!|s, collapse\ s\ xs|\!\rangle_T$
and *snd* ‘ *set ys* $\subseteq \{s\} \Longrightarrow \langle\!|s, tshift\ ys\ xs|\!\rangle_T = \langle\!|s, xs|\!\rangle_T$
and $\langle\!|s, TCons\ (a, s)\ xs|\!\rangle_T = \langle\!|s, xs|\!\rangle_T$
by (*simp-all add*: *antisym tls.singleton-le-conv behavior.natural-def*
behavior.stuttering.f-idempotent collapse.collapse collapse.tshift-stuttering)

lemmas *Sup-irreducible* = *iffD1*[*OF heyting.Sup-prime-Sup-irreducible-iff tls.singleton.Sup-prime*]
lemmas *sup-irreducible* = *Sup-irreducible-on-imp-sup-irreducible-on*[*OF tls.singleton.Sup-irreducible*, *simplified*]
lemmas *Sup-leE*[*elim*] = *Sup-prime-onE*[*OF tls.singleton.Sup-prime*, *simplified*]

lemmas *sup-le-conv*[*simp*] = *sup-irreducible-le-conv*[*OF tls.singleton.sup-irreducible*]
lemmas *Sup-le-conv*[*simp*] = *Sup-prime-on-conv*[*OF tls.singleton.Sup-prime*, *simplified*]
lemmas *compact-point* = *Sup-prime-is-compact*[*OF tls.singleton.Sup-prime*]
lemmas *compact*[*cont-intro*] = *compact-points-are-ccpo-compact*[*OF tls.singleton.compact-point*]

setup ‹*Sign.parent-path*›

setup ‹*Sign.parent-path*›

16.4 The idle process

The idle process contains no transitions and does not terminate.

setup ‹*Sign.mandatory-path raw*›

definition *idle* :: ('*a*, '*s*, '*v*) *behavior.t set* **where**
  idle = (⋃ *s*. *raw.singleton* (*behavior.B s* (*trepeat* (*undefined*, *s*))))

lemma *idle-alt-def*:
  shows *raw.idle* = {ω. ¬*tfinite* (*behavior.rest* ω) ∧ *behavior.sset* ω ⊆ {*behavior.init* ω}} (**is** *?lhs* = *?rhs*)
proof(*rule antisym*[*OF - subsetI*])
  show *?lhs* ⊆ *?rhs*
    by (*force simp*: *raw.idle-def raw.singleton-def behavior.split-all behavior.natural-def*
                behavior.sset.simps collapse.trepeat collapse.eq-trepeat-conv
              elim: *behavior.stuttering.clE*
              dest: *collapse.tfinite-conv*)
  show ω ∈ *?lhs* **if** ω ∈ *?rhs* **for** ω
    using *that*
    by (*cases* ω)
      (*auto simp*: *raw.idle-def raw.singleton-def behavior.natural-def behavior.sset.simps*
              behavior.stuttering.idemI collapse.infinite-trailing
            elim: *behavior.stuttering.clE*
           intro: *exI*[**where** *x*=*behavior.init* ω])
qed

setup ‹*Sign.mandatory-path idle*›

lemma *not-tfinite*:
  assumes ω ∈ *raw.idle*
  shows ¬*tfinite* (*behavior.rest* ω)
using *assms* **by** (*simp add*: *raw.idle-alt-def*)

setup ‹*Sign.parent-path*›

setup ‹*Sign.parent-path*›

setup ‹*Sign.mandatory-path behavior.stuttering.closed*›

lemma *idle*[*iff*]:
  shows *raw.idle* ∈ *behavior.stuttering.closed*
by (*simp add*: *raw.idle-def raw.singleton-def*
          behavior.stuttering.closed-UNION[*simplified behavior.stuttering.cl-bot*, *simplified*])

setup ‹*Sign.parent-path*›

setup ‹*Sign.mandatory-path tls*›

lift-definition *idle* :: ('*a*, '*s*, '*v*) *tls* **is** *raw.idle* **..**

lemma *idle-alt-def*:
shows *tls.idle* = (⨆ *s*. ⟪*s*, *trepeat* (*undefined*, *s*)⟫$_T$)
by *transfer* (*simp add*: *raw.idle-def behavior.stuttering.cl-bot*)

setup ‹*Sign.mandatory-path singleton*›

lemma *idle-le-conv*[*tls.singleton.le-conv*]:
shows ⟪*ω*⟫$_T$ ≤ *tls.idle* ⟷ ¬*tfinite* (*behavior.rest ω*) ∧ *behavior.sset ω* ⊆ {*behavior.init ω*}
by *transfer* (*simp add*: *raw.singleton-def behavior.stuttering.least-conv*; *simp add*: *raw.idle-alt-def*)

setup ‹*Sign.parent-path*›

setup ‹*Sign.mandatory-path idle*›

lemma *minimal-le*:
shows ⟪*s*, *trepeat* (*undefined*, *s*)⟫$_T$ ≤ *tls.idle*
by (*simp add*: *tls.singleton.idle-le-conv behavior.sset.simps*)

setup ‹*Sign.parent-path*›

setup ‹*Sign.parent-path*›

16.5 Temporal Logic for *('a, 's, 'v) tls*

The following is a straightforward shallow embedding of the now-traditional anchored semantics of LTL Manna and Pnueli (1988).
References:

- $AFP/TLA/Liveness.thy
- $ISABELLE_HOME/src/HOL/TLA/TLA.thy
- https://en.wikipedia.org/wiki/Linear_temporal_logic
- Kröger and Merz (2008)
- Warford, Vega, and Staley (2020)

Observations:

- as we lack next/X/◎ (due to stuttering closure) we do not have induction for $\mathcal{U}$ (until)
- Lamport (1994) omitted the LTL "until" operator from TLA as he considered it too hard to use
- As De Giacomo and Vardi (2013) observe, things get non-standard on finite traces
  - see §24 for an example
  - Maier (2004) provides an alternative account

setup ‹*Sign.mandatory-path raw*›

definition *state-prop* :: *'s pred* ⇒ (*'a*, *'s*, *'v*) *behavior.t set* **where**
state-prop P = {*ω*. *P* (*behavior.init ω*)}

definition
until :: (*'a*, *'s*, *'v*) *behavior.t set* ⇒ (*'a*, *'s*, *'v*) *behavior.t set* ⇒ (*'a*, *'s*, *'v*) *behavior.t set*
where
until P Q = {*ω* . ∃ *i*. ∃*ω'*∈*Q*. *behavior.dropn i ω* = *Some ω'* ∧ (∀ *j*<*i*. *the* (*behavior.dropn j ω*) ∈ *P*)}

definition

eventually :: ('*a*, '*s*, '*v*) *behavior.t set* ⇒ ('*a*, '*s*, '*v*) *behavior.t set*
where
eventually P = *raw.until UNIV P*

definition
always :: ('*a*, '*s*, '*v*) *behavior.t set* ⇒ ('*a*, '*s*, '*v*) *behavior.t set*
where
always P = −*raw.eventually* (−*P*)

abbreviation (*input*) *unless P Q* ≡ *raw.until P Q* ∪ *raw.always P*

definition *terminated* :: ('*a*, '*s*, '*v*) *behavior.t set* **where**
terminated = {ω. *tfinite* (*behavior.rest* ω) ∧ *behavior.sset* ω ⊆ {*behavior.init* ω}}

lemma *untilI*:
assumes *behavior.dropn i* ω = *Some* ω′
assumes ω′ ∈ *Q*
assumes ⋀*j*. *j* < *i* ⟹ *the* (*behavior.dropn j* ω) ∈ *P*
shows ω ∈ *raw.until P Q*
using *assms* **unfolding** *raw.until-def* **by** *blast*

lemma *eventually-alt-def*:
shows *raw.eventually P* = {ω . ∃ω′∈*P*. ∃ *i*. *behavior.dropn i* ω = *Some* ω′}
by (*auto simp*: *raw.eventually-def raw.until-def*)

lemma *always-alt-def*:
shows *raw.always P* = {ω . ∀ *i* ω′. *behavior.dropn i* ω = *Some* ω′ ⟶ ω′ ∈ *P*}
by (*auto simp*: *raw.always-def raw.eventually-alt-def*)

lemma *alwaysI*:
assumes ⋀*i* ω′. *behavior.dropn i* ω = *Some* ω′ ⟹ ω′ ∈ *P*
shows ω ∈ *raw.always P*
by (*simp add*: *raw.always-alt-def assms*)

lemma *alwaysD*:
assumes ω ∈ *raw.always P*
assumes *behavior.dropn i* ω = *Some* ω′
shows ω′ ∈ *P*
using *assms* **by** (*simp add*: *raw.always-alt-def*)

setup ‹*Sign.mandatory-path state-prop*›

lemma *monotone*:
shows *mono raw.state-prop*
by (*fastforce intro*: *monoI simp*: *raw.state-prop-def*)

lemma *simps*:
shows
raw.state-prop ⟨*False*⟩ = {}
raw.state-prop ⊥ = {}
raw.state-prop ⟨*True*⟩ = *UNIV*
raw.state-prop ⊤ = *UNIV*
− *raw.state-prop P* = *raw.state-prop* (− *P*)
raw.state-prop P ∪ *raw.state-prop Q* = *raw.state-prop* (*P* ⊔ *Q*)
raw.state-prop P ∩ *raw.state-prop Q* = *raw.state-prop* (*P* ⊓ *Q*)
(*raw.state-prop P* ⟶$_B$ *raw.state-prop Q*) = *raw.state-prop* (*P* ⟶$_B$ *Q*)
by (*auto simp*: *raw.state-prop-def boolean-implication.set-alt-def*)

lemma *Inf*:
  shows *raw.state-prop* $(\bigsqcap X) = \bigcap (raw.state\text{-}prop\ `\ X)$
by (*fastforce simp*: *raw.state-prop-def*)

lemma *Sup*:
  shows *raw.state-prop* $(\bigsqcup X) = \bigcup (raw.state\text{-}prop\ `\ X)$
by (*fastforce simp*: *raw.state-prop-def*)

setup ‹*Sign.parent-path*›

setup ‹*Sign.mandatory-path terminated*›

lemma *inf-always-le*:
  fixes $P :: ('a, 's, 'v)$ *behavior.t set*
  assumes $P \in$ *behavior.stuttering.closed*
  shows *raw.terminated* $\cap$ $P \subseteq$ *raw.always* P
by (*rule subsetI*[*OF raw.alwaysI*])
   (*auto simp*: *raw.terminated-def*
        elim: *behavior.stuttering.closed-in*[*OF - - assms*] *behavior.stuttering.equiv.idle-dropn*)

setup ‹*Sign.parent-path*›

setup ‹*Sign.mandatory-path until*›

lemma *base*:
  shows $\omega \in Q \Longrightarrow \omega \in$ *raw.until* P Q
    and $Q \subseteq$ *raw.until* P Q
by (*force simp*: *raw.until-def*)+

lemma *step*:
  assumes $\omega \in P$
  assumes *behavior.tl* $\omega =$ *Some* ω'
  assumes $\omega' \in$ *raw.until* P Q
  shows $\omega \in$ *raw.until* P Q
proof −
  from ‹$\omega' \in$ *raw.until* P Q›
  obtain i ω''
    where $\omega'' \in Q$ **and** $\forall j{<}i.$ *the* (*behavior.dropn* j ω') $\in P$ **and** *behavior.dropn* i $\omega' =$ *Some* ω''
    by (*clarsimp simp*: *raw.until-def*)
  with *assms*(*1*,*2*) **show** *?thesis*
    by (*clarsimp simp*: *raw.until-def behavior.dropn.Suc less-Suc-eq-0-disj*
          intro!: *exI*[**where** x=*Suc* i])
qed

lemmas *intro*[*intro*] =
  raw.until.base
  raw.until.step

lemma *induct*[*case-names base step*, *consumes 1*, *induct set*: *raw.until*]:
  assumes $\omega \in$ *raw.until* P Q
  assumes *base*: $\bigwedge\omega.\ \omega \in Q \Longrightarrow R\ \omega$
  assumes *step*: $\bigwedge\omega\ \omega'.\ [\![\omega \in P;$ *behavior.tl* $\omega =$ *Some* $\omega';\ \omega' \in$ *raw.until* P $Q;\ R\ \omega']\!] \Longrightarrow R\ \omega$
  shows $R\ \omega$
proof −
  from ‹$\omega \in$ *raw.until* P Q› **obtain** ω' i
    where *behavior.dropn* i $\omega =$ *Some* ω' **and** $\omega' \in Q$ **and** $\forall j{<}i.$ *the* (*behavior.dropn* j ω) $\in P$
    unfolding *raw.until-def* **by** *blast*
  then show *?thesis*

```
  proof(induct i arbitrary: ω)
    case 0 then show ?case
      by (force intro: base)
  next
    case Suc from Suc.prems show ?case
      by (fastforce intro: step Suc.hyps dest: spec[where x=Suc j for j]
                    simp: behavior.dropn.Suc raw.until-def
                   split: Option.bind-split-asm)
  qed
qed

lemma mono:
  assumes P ⊆ P'
  assumes Q ⊆ Q'
  shows raw.until P Q ⊆ raw.until P' Q'
unfolding raw.until-def using assms by blast

lemma botL:
  shows raw.until {} Q = Q
by (force simp: raw.until-def)

lemma botR:
  shows raw.until P {} = {}
by (force simp: raw.until-def)

lemma untilR:
  shows raw.until P (raw.until P Q) = raw.until P Q (is ?lhs = ?rhs)
proof(rule antisym[OF subsetI])
  show ω ∈ ?rhs if ω ∈ ?lhs for ω using that by induct blast+
  show ?rhs ⊆ ?lhs by blast
qed

lemma InfL-not-empty:
  assumes X ≠ {}
  shows raw.until (⋂ X) Q = (⋂ x∈X. raw.until x Q) (is ?lhs = ?rhs)
proof(rule antisym[OF - subsetI])
  show ?lhs ⊆ ?rhs
    by (simp add: INT-greatest Inter-lower raw.until.mono)
  show ω ∈ ?lhs if ω ∈ ?rhs for ω
  proof −
    from ‹X ≠ {}› obtain P where P ∈ X by blast
    with that obtain i ω'
      where ∗: behavior.dropn i ω = Some ω' ω' ∈ Q ∀ j<i. the (behavior.dropn j ω) ∈ P
      unfolding raw.until-def by blast
    from this(1,2) obtain k ω''
      where ∗∗: k ≤ i behavior.dropn k ω = Some ω'' ω'' ∈ Q ∀ j<k. the (behavior.dropn j ω) ∉ Q
      using ex-has-least-nat[where k=i and P=λk. k ≤ i ∧ (∀ ω''. behavior.dropn k ω = Some ω'' ⟶ ω'' ∈ Q)
and m=id]
      by clarsimp (metis (no-types, lifting) behavior.dropn.shorterD leD nle-le option.sel order.trans)
    from that ∗ ∗∗ show ?thesis
      by (clarsimp simp: raw.until-def intro!: exI[where x=k])
         (metis order.strict-trans1 linorder-not-le option.sel)
  qed
qed

lemma SupR:
  shows raw.until P (⋃ X) = ⋃ (raw.until P ‘ X)
unfolding raw.until-def by blast
```

lemma *weakenL*:
shows *raw.until UNIV P* = *raw.until* (− *P*) *P* (**is** *?lhs* = *?rhs*)
proof(*rule antisym*[*OF subsetI*])
show ω ∈ *?rhs* **if** ω ∈ *?lhs* **for** ω **using** *that* **by** *induct blast*+
show *?rhs* ⊆ *?lhs* **by** (*simp add*: *raw.until.mono*)
qed

lemma *implication-ordering-le*: — Warford et al. (2020, (16))
shows *raw.until P Q* ∩ *raw.until* (−*Q*) *R* ⊆ *raw.until P R*
by (*clarsimp simp*: *raw.until-def*) (*metis order.trans linorder-not-le option.sel*)

lemma *infR-ordering-le*: — Warford et al. (2020, (18))
shows *raw.until P* (*Q* ∩ *R*) ⊆ *raw.until* (*raw.until P Q*) *R* (**is** *?lhs*⊆ *?rhs*)
proof(*rule subsetI*)
show ω ∈ *?rhs* **if** ω ∈ *?lhs* **for** ω
using *that*
proof *induct*
case (*step* ω ω′) **then show** *?case*
by − (*rule raw.until.step*, *rule raw.until.step*;
blast intro: *subsetD*[*OF raw.until.mono*, *rotated* −*1*])
qed *blast*
qed

lemma *untilL*:
shows *raw.until* (*raw.until P Q*) *Q* ⊆ *raw.until P Q* (**is** *?lhs*⊆ *?rhs*)
proof(*rule subsetI*)
show ω ∈ *?rhs* **if** ω ∈ *?lhs* **for** ω
using *that* **by** *induct auto*
qed

lemma *alwaysR-le*:
shows *raw.until P* (*raw.always Q*) ⊆ *raw.always* (*raw.until P Q*) (**is** *?lhs* ⊆ *?rhs*)
proof(*rule subsetI*)
show ω ∈ *?rhs* **if** ω ∈ *?lhs* **for** ω
using *that*
proof *induct*
case (*base* ω) **then show** *?case* **by** (*auto simp*: *raw.always-alt-def*)
next
case (*step* ω ω′) **show** *?case*
proof(*rule raw.alwaysI*)
fix *i* ω″ **assume** *behavior.dropn i* ω = *Some* ω″
with *step behavior.dropn.0* **show** ω″ ∈ *raw.until P Q*
by (*cases i*; *clarsimp simp*: *raw.always-alt-def behavior.dropn.Suc*; *blast*)
qed
qed
qed

setup ‹*Sign.parent-path*›

setup ‹*Sign.mandatory-path unless*›

lemma *neg*:
shows − (*raw.until P Q* ∪ *raw.always P*) = *raw.until* (− *Q*) (− *P* ∩ − *Q*) (**is** *?lhs* = *?rhs*)
proof(*rule antisym*[*OF subsetI*], (*unfold Compl-Un Int-iff conj-explode Compl-iff*)[*1*])
fix ω
assume ∗: ω ∉ *raw.until P Q*
assume ω ∉ *raw.always P*

```
  then obtain k ω'
    where behavior.dropn k ω = Some ω'
      and ω' ∉ P
    by (clarsimp simp: raw.always-alt-def)
  with ex-has-least-nat[where k=k and P=λi. ∃ω'. behavior.dropn i ω = Some ω' ∧ ω' ∉ P and m=id]
  obtain k ω'
    where behavior.dropn k ω = Some ω'
      and ω' ∉ P
      and ∀j<k. the (behavior.dropn j ω) ∈ P
    by clarsimp (metis behavior.dropn.shorterD less-le-not-le option.distinct(1) option.exhaust-sel)
  with * behavior.dropn.shorterD show ω ∈ ?rhs
    by (fastforce simp: raw.until-def intro: exI[where x=k])
next
  show ?rhs ⊆ ?lhs
    by (clarsimp simp: raw.always-alt-def raw.until-def subset-iff; metis nat-neq-iff option.sel)
qed

setup ‹Sign.parent-path›

setup ‹Sign.mandatory-path eventually›

lemma terminated:
  shows raw.eventually raw.terminated = {ω. tfinite (behavior.rest ω)} (is ?lhs = ?rhs)
proof(rule antisym[OF - subsetI])
  show ?lhs ⊆ ?rhs
    by (clarsimp simp: raw.eventually-alt-def raw.terminated-def behavior.dropn.tfiniteD)
  show ω ∈ ?lhs if ω ∈ ?rhs for ω
  proof −
    note ‹ω ∈ ?rhs›
    moreover from calculation
    obtain i where tlength (behavior.rest ω) = enat i
      by (clarsimp simp: tfinite-tlength-conv)
    moreover from calculation
    obtain ω' where behavior.dropn i ω = Some ω'
      using behavior.dropn.eq-Some-tlength-conv by fastforce
    moreover from calculation
    have behavior.sset ω' ⊆ {behavior.init ω'}
      by (cases ω')
         (clarsimp dest!: behavior.dropn.eq-Some-tdropnD simp: tdropn-tlength behavior.sset.simps)
    ultimately show ω ∈ ?lhs
      by (auto simp: raw.eventually-alt-def raw.terminated-def dest: behavior.dropn.tfiniteD)
  qed
qed

setup ‹Sign.parent-path›

setup ‹Sign.parent-path›

setup ‹Sign.mandatory-path behavior.stuttering.closed.raw›

lemma state-prop[intro]:
  shows raw.state-prop P ∈ behavior.stuttering.closed
by (fastforce simp: raw.state-prop-def behavior.natural-def elim: behavior.stuttering.clE)

lemma terminated[intro]:
  shows raw.terminated ∈ behavior.stuttering.closed
by (rule behavior.stuttering.closedI)
   (clarsimp simp: raw.terminated-def elim!: behavior.stuttering.clE;
```

metis behavior.natural.sel(1) behavior.natural.tfinite behavior.sset.natural)

lemma *until*[*intro*]:
assumes $P \in$ *behavior.stuttering.closed*
assumes $Q \in$ *behavior.stuttering.closed*
shows *raw.until P Q* $\in$ *behavior.stuttering.closed*
proof −
have $\omega_2 \in$ *raw.until P Q* **if** $\omega_1 \in$ *raw.until P Q* **and** $\omega_1 \simeq_T \omega_2$ **for** $\omega_1\ \omega_2$
using *that*
proof(*induct arbitrary*: ω_2 *rule*: *raw.until.induct*)
case (*base* $\omega_1\ \omega_2$) **with** *assms*(*2*) **show** *?case*
by (*blast intro*: *behavior.stuttering.closed-in*)
next
case (*step* $\omega_1\ \omega'\ \omega_2$)
show *?case*
proof(*cases* $\omega' \simeq_T \omega_1$)
case *True* **with** ‹$\omega_1 \simeq_T \omega_2$› *step.hyps*(*4*) **show** *?thesis*
by *simp*
next
case *False*
from *assms*(*1*) ‹$\omega_1 \in P$› ‹$\omega_1 \simeq_T \omega_2$› **have** $\omega_2 \in P$
by (*blast intro*: *behavior.stuttering.closed-in*)
from *False* ‹$\omega_1 \simeq_T \omega_2$› ‹*behavior.tl* ω_1 = *Some* ω'›
obtain *a* $s_0\ s_1\ xs_1\ xs'\ ys'$
where ω_1: ω_1 = *behavior.B* s_0 (*TCons* (*a*, s_1) xs_1)
and ω_2: ω_2 = *behavior.B* s_0 (*tshift* xs' (*TCons* (*a*, s_1) ys'))
and ∗: *collapse* s_0 (*TCons* (*a*, s_1) xs_1) = *collapse* s_0 (*tshift* xs' (*TCons* (*a*, s_1) ys'))
$s_0 \neq s_1$
and ∗∗: *collapse* $s_1\ ys'$ = *collapse* $s_1\ xs_1$
and xs': *snd* ‘ *set* $xs' \subseteq \{s_0\}$
by (*cases* ω_1; *cases* ω_2; *cases behavior.rest* ω_1; *simp*)
(*fastforce simp*: *behavior.natural-def collapse.eq-TCons-conv trepeat-eq-TCons-conv*
split: *if-splits*)
from ω_2 ‹$\omega_2 \in P$› xs' **show** *?thesis*
proof(*induct* xs' *arbitrary*: ω_2)
case *Nil* **with** ω_1 ∗∗ *step.hyps*(*2*,*4*) **show** *?case*
by (*auto simp*: *behavior.natural-def*)
next
case (*Cons* $x'\ xs'$)
with *behavior.stuttering.closed-in*[*OF - -* ‹$P \in$ *behavior.stuttering.closed*›] ω_1 ∗∗ *step*(*3*)
show *?case*
by (*auto simp*: *behavior.natural-def behavior.split-all*)
qed
qed
qed
then show *?thesis*
by (*fastforce elim*: *behavior.stuttering.clE*)
qed

lemma *eventually*[*intro*]:
assumes $P \in$ *behavior.stuttering.closed*
shows *raw.eventually P* $\in$ *behavior.stuttering.closed*
using *assms* **by** (*auto simp*: *raw.eventually-def*)

lemma *always*[*intro*]:
assumes $P \in$ *behavior.stuttering.closed*
shows *raw.always P* $\in$ *behavior.stuttering.closed*
using *assms* **by** (*auto simp*: *raw.always-def*)

setup ‹*Sign.parent-path*›

setup ‹*Sign.mandatory-path tls*›

definition *valid* :: ('*a*, '*s*, '*v*) *tls* ⇒ *bool* **where**
valid P ⟷ *P* = ⊤

lift-definition *state-prop* :: '*s pred* ⇒ ('*a*, '*s*, '*v*) *tls* **is** *raw.state-prop* ..
lift-definition *terminated* :: ('*a*, '*s*, '*v*) *tls* **is** *raw.terminated* ..
lift-definition *until* :: ('*a*, '*s*, '*v*) *tls* ⇒ ('*a*, '*s*, '*v*) *tls* ⇒ ('*a*, '*s*, '*v*) *tls* **is** *raw.until* ..

definition *eventually* :: ('*a*, '*s*, '*v*) *tls* ⇒ ('*a*, '*s*, '*v*) *tls* **where**
eventually P = *tls.until* ⊤ *P*

definition *always* :: ('*a*, '*s*, '*v*) *tls* ⇒ ('*a*, '*s*, '*v*) *tls* **where**
always P = −*tls.eventually* (−*P*)

definition *release* :: ('*a*, '*s*, '*v*) *tls* ⇒ ('*a*, '*s*, '*v*) *tls* ⇒ ('*a*, '*s*, '*v*) *tls* **where**
release P Q = −(*tls.until* (−*P*) (−*Q*))

definition *unless* :: ('*a*, '*s*, '*v*) *tls* ⇒ ('*a*, '*s*, '*v*) *tls* ⇒ ('*a*, '*s*, '*v*) *tls* **where**
unless P Q = *tls.until P Q* ⊔ *tls.always P*

abbreviation (*input*) *always-imp-syn* :: ('*a*, '*s*, '*v*) *tls* ⇒ ('*a*, '*s*, '*v*) *tls* ⇒ ('*a*, '*s*, '*v*) *tls* **where**
always-imp-syn P Q ≡ *tls.always* (*P* ⟶$_B$ *Q*)

abbreviation (*input*) *leads-to* :: ('*a*, '*s*, '*v*) *tls* ⇒ ('*a*, '*s*, '*v*) *tls* ⇒ ('*a*, '*s*, '*v*) *tls* **where**
leads-to P Q ≡ *tls.always-imp-syn P* (*tls.eventually Q*)

open-bundle *syntax*
begin
notation *tls.valid* (‹⊨ -› [*30*] *30*)
notation *tls.state-prop* (‹⟪-⟫› [*0*])
notation *tls.until* (**infix** ‹*U*› *85*)
notation *tls.eventually* (‹◇-› [*87*] *87*)
notation *tls.always* (‹□-› [*87*] *87*)
notation *tls.release* (**infixr** ‹*R*› *85*)
notation *tls.unless* (**infixr** ‹*W*› *85*)
notation *tls.always-imp-syn* (**infixr** ‹⟶$_□$› *75*)
notation *tls.leads-to* (**infixr** ‹⇝› *75*)
end

bundle *no-syntax*
begin
no-notation *tls.valid* (‹⊨ -› [*30*] *30*)
no-notation *tls.state-prop* (‹⟪-⟫› [*0*])
no-notation *tls.until* (**infixr** ‹*U*› *85*)
no-notation *tls.eventually* (‹◇-› [*87*] *87*)
no-notation *tls.always* (‹□-› [*87*] *87*)
no-notation *tls.release* (**infixr** ‹*R*› *85*)
no-notation *tls.unless* (**infixr** ‹*W*› *85*)
no-notation *tls.always-imp-syn* (**infixr** ‹⟶$_□$› *75*)
no-notation *tls.leads-to* (**infixr** ‹⇝› *75*)
end

lemma *validI*:
assumes ⊤ ≤ *P*

shows $\models P$
by (*simp add*: *assms tls.valid-def top.extremum-uniqueI*)

setup ‹*Sign.mandatory-path valid*›

lemma *trans*[*trans*]:
assumes $\models P$
assumes $P \leq Q$
shows $\models Q$
using *assms* **by** (*simp add*: *tls.valid-def top.extremum-unique*)

lemma *mp*:
assumes $\models P \longrightarrow_B Q$
assumes $\models P$
shows $\models Q$
using *assms* **by** (*simp add*: *tls.valid-def*)

lemmas *rev-mp* = *tls.valid.mp*[*rotated*]

setup ‹*Sign.parent-path*›

setup ‹*Sign.mandatory-path singleton*›

lemma *uminus-le-conv*[*tls.singleton.le-conv*]:
shows $\langle\!|\omega|\!\rangle_T \leq -P \longleftrightarrow \neg\langle\!|\omega|\!\rangle_T \leq P$
by *transfer*
(*simp add*: *raw.singleton-def behavior.stuttering.closed-uminus behavior.stuttering.least-conv*)

lemma *state-prop-le-conv*[*tls.singleton.le-conv*]:
shows $\langle\!|\omega|\!\rangle_T \leq$ *tls.state-prop* $P \longleftrightarrow P$ (*behavior.init* ω)
by *transfer*
(*simp add*: *raw.singleton-def behavior.stuttering.least-conv*[*OF behavior.stuttering.closed.raw.state-prop*];
simp add: *raw.state-prop-def*)

lemma *terminated-le-conv*[*tls.singleton.le-conv*]:
shows $\langle\!|\omega|\!\rangle_T \leq$ *tls.terminated* $\longleftrightarrow$ *tfinite* (*behavior.rest* ω) $\wedge$ *behavior.sset* $\omega \subseteq \{$*behavior.init* $\omega\}$
by *transfer*
(*simp add*: *raw.singleton-def behavior.stuttering.least-conv*[*OF behavior.stuttering.closed.raw.terminated*];
simp add: *raw.terminated-def*)

lemma *until-le-conv*[*tls.singleton.le-conv*]:
fixes P :: ('*a*, '*s*, '*v*) *tls*
shows $\langle\!|\omega|\!\rangle_T \leq P\ \mathcal{U}\ Q \longleftrightarrow (\exists\, i\ \omega'.$ *behavior.dropn* $i\ \omega =$ *Some* ω'
$\wedge\ \langle\!|\omega'|\!\rangle_T \leq Q$
$\wedge\ (\forall\, j{<}i.\ \langle\!|$*the* (*behavior.dropn* $j\ \omega)|\!\rangle_T \leq P))$ (**is** *?lhs* $\longleftrightarrow$ *?rhs*)
proof(*rule iffI*)
show *?lhs* $\Longrightarrow$ *?rhs*
proof *transfer*
fix ω **and** $P\ Q$ :: ('*a*, '*s*, '*v*) *behavior.t set*
assume $*$: $P \in$ *behavior.stuttering.closed* $Q \in$ *behavior.stuttering.closed*
and *raw.singleton* $\omega \subseteq$ *raw.until* $P\ Q$
then have $\exists\, i.\ \exists\, \omega' {\in} Q.$ *behavior.dropn* $i\ \omega =$ *Some* $\omega' \wedge (\forall\, j{<}i.$ *the* (*behavior.dropn* $j\ \omega) \in P)$
by (*auto simp*: *raw.singleton-def raw.until-def*)
with $*$ **show** $\exists\, i\ \omega'.$ *behavior.dropn* $i\ \omega =$ *Some* ω'
$\wedge$ *raw.singleton* $\omega' \subseteq Q \wedge (\forall\, j{<}i.$ *raw.singleton* (*the* (*behavior.dropn* $j\ \omega)) \subseteq P)$
by (*auto simp*: *raw.singleton-def behavior.stuttering.least-conv*)
qed
show *?rhs* $\Longrightarrow$ *?lhs*

by *transfer*
(*unfold raw.singleton-def*;
rule behavior.stuttering.least[*OF - behavior.stuttering.closed.raw.until*];
auto 10 0 intro: *iffD2*[*OF eqset-imp-iff*[*OF raw.until-def*]])
qed

lemma *eventually-le-conv*[*tls.singleton.le-conv*]:
shows $\langle\!|\omega|\!\rangle_T \leq \Diamond P \longleftrightarrow (\exists\, i\ \omega'.\ behavior.dropn\ i\ \omega = Some\ \omega' \wedge \langle\!|\omega'|\!\rangle_T \leq P)$
by (*simp add*: *tls.eventually-def tls.singleton.le-conv*)

lemma *always-le-conv*[*tls.singleton.le-conv*]:
shows $\langle\!|\omega|\!\rangle_T \leq tls.always\ P \longleftrightarrow (\forall\, i\ \omega'.\ behavior.dropn\ i\ \omega = Some\ \omega' \longrightarrow \langle\!|\omega'|\!\rangle_T \leq P)$
by (*simp add*: *tls.always-def tls.singleton.le-conv*)

setup ‹*Sign.parent-path*›

interpretation *until*: *closure-complete-lattice-distributive-class tls.until P* **for** *P*
proof *standard*
show $(x \leq tls.until\ P\ y) = (tls.until\ P\ x \leq tls.until\ P\ y)$ **for** *x y*
by *transfer*
(*intro iffD2*[*OF order-class.order.closure-axioms-alt-def*[*unfolded closure-axioms-def*], *rule-format*]
conjI allI raw.until.base monoI raw.until.mono order.refl raw.until.untilR, *assumption*)
show $tls.until\ P\ (\bigsqcup X) \leq \bigsqcup (tls.until\ P\ `\ X) \sqcup tls.until\ P\ \bot$ **for** *X*
by *transfer* (*simp add*: *raw.until.SupR behavior.stuttering.cl-bot*)
qed

setup ‹*Sign.mandatory-path until*›

lemmas *botL* = *raw.until.botL*[*transferred*]
lemmas *botR* = *raw.until.botR*[*transferred*]
lemmas *topR* = *tls.until.cl-top*
lemmas *expansiveR* = *tls.until.expansive*[*of P Q* **for** *P Q*]

lemmas *weakenL* = *raw.until.weakenL*[*transferred*]

lemmas *mono* = *raw.until.mono*[*transferred*]

lemma *strengthen*[*strg*]:
assumes *st-ord F P P'*
assumes *st-ord F Q Q'*
shows *st-ord F* $(P\ \mathcal{U}\ Q)\ (P'\ \mathcal{U}\ Q')$
using *assms* **by** (*cases F*) (*auto simp*: *tls.until.mono*)

lemma *SupL-le*:
shows $(\bigsqcup x{\in}X.\ x\ \mathcal{U}\ R) \leq (\bigsqcup X)\ \mathcal{U}\ R$
by (*simp add*: *SupI tls.until.mono*)

lemma *supL-le*:
shows $P\ \mathcal{U}\ R \sqcup Q\ \mathcal{U}\ R \leq (P \sqcup Q)\ \mathcal{U}\ R$
by (*simp add*: *tls.until.mono*)

lemma *SupR*:
shows $P\ \mathcal{U}\ (\bigsqcup X) = \bigsqcup ((\mathcal{U})\ P\ `\ X)$
by (*simp add*: *tls.until.cl-Sup tls.until.botR*)

lemmas *supR* = *tls.until.cl-sup*

lemmas *InfL-not-empty* = *raw.until.InfL-not-empty*[*transferred*]

lemmas *infL* = *tls.until.InfL-not-empty*[**where** *X*={*P*, *Q*} **for** *P Q*, *simplified*, *of P Q R* **for** *P Q R*]
lemmas *InfR-le* = *tls.until.cl-Inf-le*
lemmas *infR-le* = *tls.until.cl-inf-le*[*of P Q R* **for** *P Q R*]

lemma *implication-ordering-le*: — Warford et al. (2020, (16))
  shows $P\ \mathcal{U}\ Q \sqcap (-Q)\ \mathcal{U}\ R \leq P\ \mathcal{U}\ R$
by (*transfer* (*rule raw.until.implication-ordering-le*))

lemma *supL-ordering-le*: — Warford et al. (2020, (17))
  shows $P\ \mathcal{U}\ (Q\ \mathcal{U}\ R) \leq (P \sqcup Q)\ \mathcal{U}\ R$ (**is** *?lhs* ≤ *?rhs*)
proof −
  have *?rhs* = $(P \sqcup Q)\ \mathcal{U}\ ((P \sqcup Q)\ \mathcal{U}\ R)$ **by** (*rule tls.until.idempotent*(*1*)[*symmetric*])
  also have *?lhs* ≤ ... **by** (*blast intro*: *tls.until.mono le-supI1 le-supI2*)
  finally show *?thesis* **.**
qed

lemma *infR-ordering-le*: — Warford et al. (2020, (18))
  shows $P\ \mathcal{U}\ (Q \sqcap R) \leq (P\ \mathcal{U}\ Q)\ \mathcal{U}\ R$
by (*transfer* (*rule raw.until.infR-ordering-le*))

lemma *boolean-implication-distrib-le*: — Warford et al. (2020, (19))
  shows $(P \longrightarrow_B Q)\ \mathcal{U}\ R \leq (P\ \mathcal{U}\ R) \longrightarrow_B (Q\ \mathcal{U}\ R)$
by (*metis galois.conj-imp.galois order.refl tls.until.infL tls.until.mono*)

lemma *excluded-middleR*: — Warford et al. (2020, (23))
  shows $\models P\ \mathcal{U}\ Q \sqcup P\ \mathcal{U}\ (-Q)$
by (*simp add*: *tls.validI tls.until.cl-top flip*: *tls.until.cl-sup*)

lemmas *untilR* = *tls.until.idempotent*(*1*)[*of P Q* **for** *P Q*]

lemma *untilL*:
  shows $(P\ \mathcal{U}\ Q)\ \mathcal{U}\ Q = P\ \mathcal{U}\ Q$ (**is** *?lhs* = *?rhs*)
proof(*rule antisym*)
  show *?lhs* ≤ *?rhs*
    by *transfer* (*rule raw.until.untilL*)
  show *?rhs* ≤ *?lhs*
    using *tls.until.infR-ordering-le*[**where** *P*=*P* **and** *Q*=*Q* **and** *R*=*Q*] **by** *simp*
qed

lemma *absorb*:
  shows $P\ \mathcal{U}\ P = P$
by (*metis tls.until.botL tls.until.untilL*)

lemma *absorb-supL*: — Warford et al. (2020, (23))
  shows $P \sqcup P\ \mathcal{U}\ Q = P \sqcup Q$
by (*metis inf-commute inf-sup-absorb le-iff-sup*
      tls.until.absorb tls.until.cl-sup tls.until.expansive tls.until.infL)

lemma *absorb-supR*: — Warford et al. (2020, (23))
  shows $Q \sqcup P\ \mathcal{U}\ Q = P\ \mathcal{U}\ Q$
by (*simp add*: *sup.absorb2 tls.until.expansive*)

lemma *eventually-le*:
  shows $P\ \mathcal{U}\ Q \leq \Diamond Q$
by (*simp add*: *tls.eventually-def tls.until.mono*)

lemma *absorb-eventually*:
  shows *inf-eventually-absorbR*: $P\ \mathcal{U}\ Q \sqcap \Diamond Q = P\ \mathcal{U}\ Q$ — Warford et al. (2020, (39))

and *sup-eventually-absorbR*: $P\ \mathcal{U}\ Q \sqcup \Diamond Q = \Diamond Q$ — Warford et al. (2020, (40))
and *eventually-absorbR*: $P\ \mathcal{U}\ \Diamond Q = \Diamond Q$ — Warford et al. (2020, (41))
by (*simp-all add*: *tls.eventually-def sup.absorb2 tls.until.mono*
order.eq-iff order.trans[*OF tls.until.supL-ordering-le*] *tls.until.expansiveR*
flip: *tls.until.infL*)

lemma *sup-le*: — Warford et al. (2020, (28))
shows $P\ \mathcal{U}\ Q \leq P \sqcup Q$
by (*simp add*: *ac-simps sup.absorb-iff1 tls.until.absorb-supL tls.until.absorb-supR*)

lemma *ordering*: — Warford et al. (2020, (251))
shows $(-P)\ \mathcal{U}\ Q \sqcup (-Q)\ \mathcal{U}\ P = \Diamond(P \sqcup Q)$ (**is** *?lhs* = *?rhs*)
proof −
have *?lhs* = $\top\ \mathcal{U}\ P \sqcap (-\ Q)\ \mathcal{U}\ P \sqcup \top\ \mathcal{U}\ Q \sqcap (-\ P)\ \mathcal{U}\ Q$
by (*simp add*: *ac-simps inf.absorb2 tls.until.mono*)
also have ... = $(-\ P)\ \mathcal{U}\ P \sqcap (-\ Q)\ \mathcal{U}\ P \sqcup (-\ Q)\ \mathcal{U}\ Q \sqcap (-\ P)\ \mathcal{U}\ Q$
by (*simp add*: *tls.until.weakenL*)
also have ... = $(-\ (P \sqcup Q))\ \mathcal{U}\ (P \sqcup Q)$
by (*simp add*: *ac-simps tls.until.cl-sup flip*: *tls.until.infL*)
also have ... = *?rhs*
by (*simp add*: *tls.eventually-def tls.until.weakenL*)
finally show *?thesis* **.**
qed

lemmas *simps* =
tls.until.expansiveR
tls.until.botL
tls.until.botR
tls.until.absorb
tls.until.absorb-supL
tls.until.absorb-supR
tls.until.untilL
tls.until.untilR

setup ‹*Sign.parent-path*›

interpretation *eventually*: *closure-complete-lattice-distributive-class tls.eventually*
unfolding *tls.eventually-def*
by (*simp add*: *tls.until.closure-complete-lattice-distributive-class-axioms*)

lemma *eventually-alt-def*:
shows $\Diamond P = (-P)\ \mathcal{U}\ P$
by (*simp add*: *tls.eventually-def tls.until.weakenL*)

setup ‹*Sign.mandatory-path eventually*›

lemma *transfer*[*transfer-rule*]:
shows *rel-fun* (*pcr-tls* (=) (=) (=)) (*pcr-tls* (=) (=) (=)) *raw.eventually tls.eventually*
unfolding *tls.eventually-def raw.eventually-def* **by** *transfer-prover*

lemma *bot*:
shows $\Diamond \bot = \bot$
by (*simp add*: *tls.eventually-def tls.until.simps*)

lemma *bot-conv*:
shows $\Diamond P = \bot \longleftrightarrow P = \bot$
by (*auto simp*: *tls.eventually.bot bot.extremum-uniqueI*[*OF order.trans*[*OF tls.eventually.expansive*]])

lemmas *top* = *tls.eventually.cl-top*

lemmas *monotone* = *tls.eventually.monotone-cl*
lemmas *mono* = *tls.eventually.mono-cl*

lemmas *Sup* = *tls.eventually.cl-Sup*[*simplified tls.eventually.bot*, *simplified*]
lemmas *sup* = *tls.eventually.Sup*[**where** *X*={*P*, *Q*} **for** *P Q*, *simplified*]

lemmas *Inf-le* = *tls.eventually.cl-Inf-le*
lemmas *inf-le* = *tls.eventually.cl-inf-le*

lemma *neg*:
  shows $-\Diamond P = \Box(-P)$
by (*simp add*: *tls.always-def*)

lemma *boolean-implication-le*:
  shows $\Diamond P \longrightarrow_B \Diamond Q \leq \Diamond(P \longrightarrow_B Q)$
by (*simp add*: *boolean-implication.conv-sup tls.eventually.sup*)
   (*meson le-supI1 compl-mono order.trans le-supI1 tls.eventually.expansive*)

lemmas *simps* =
  tls.eventually.bot
  tls.eventually.top
  tls.eventually.expansive
  tls.eventually-def[*symmetric*]

lemma *terminated*:
  shows $\Diamond$*tls.terminated* = $\bigsqcup$(*tls.singleton* ‘ {ω. *tfinite* (*behavior.rest* ω)})
by *transfer*
   (*auto elim*!: *behavior.stuttering.clE*
         dest: *iffD2*[*OF behavior.natural.tfinite*]
         simp: *raw.eventually.terminated behavior.stuttering.cl-bot raw.singleton-def collapse.tfinite*)

lemma *always-imp-le*:
  shows $P \longrightarrow_\Box Q \leq \Diamond P \longrightarrow_B \Diamond Q$
by (*simp add*: *tls.always-def boolean-implication.conv-sup flip*: *shunt2*)
   (*metis inf-commute order.refl shunt2 sup.commute tls.eventually.mono tls.eventually.sup*)

lemma *until*:
  shows $\Diamond(P\ \mathcal{U}\ Q) = \Diamond Q$
by (*meson antisym tls.eventually.cl tls.eventually.mono tls.until.eventually-le tls.until.expansiveR*)

setup ‹*Sign.parent-path*›

lemma *always-alt-def*:
  shows $\Box P = P\ \mathcal{W}\ \bot$
by (*simp add*: *tls.unless-def tls.until.simps*)

setup ‹*Sign.mandatory-path always*›

lemma *transfer*[*transfer-rule*]:
  shows *rel-fun* (*pcr-tls* (=) (=) (=)) (*pcr-tls* (=) (=) (=)) *raw.always tls.always*
unfolding *tls.always-def raw.always-def* **by** *transfer-prover*

tls.always is an interior operator

lemma *idempotent*[*simp*]:
  shows $\Box\Box P = \Box P$
by (*simp add*: *tls.always-def*)

lemma *contractive*:
  shows $\Box P \leq P$
by (*simp add*: *tls.always-def compl-le-swap2 tls.eventually.expansive*)

lemma *monotone*[*iff*]:
  shows *mono tls.always*
by (*simp add*: *tls.always-def monoI tls.eventually.mono*)

lemmas *strengthen*[*strg*] = *st-monotone*[*OF tls.always.monotone*]
lemmas *mono*[*trans*] = *monoD*[*OF tls.always.monotone*]

lemma *bot*:
  shows $\Box\bot = \bot$
by (*simp add*: *tls.always-def tls.eventually.simps*)

lemma *top*:
  shows $\Box\top = \top$
by (*simp add*: *tls.always-def tls.eventually.simps*)

lemma *top-conv*:
  shows $\Box P = \top \longleftrightarrow P = \top$
by (*auto simp*: *tls.always.top intro*: *top.extremum-uniqueI*[*OF order.trans*[*OF - tls.always.contractive*]])

lemma *Sup-le*:
  shows $\bigsqcup(tls.always\ `\ X) \leq \Box(\bigsqcup X)$
by (*simp add*: *SupI tls.always.mono*)

lemma *sup-le*:
  shows $\Box P \sqcup \Box Q \leq \Box(P \sqcup Q)$
by (*simp add*: *tls.always.mono*)

lemma *Inf*:
  shows $\Box(\bigsqcap X) = \bigsqcap(tls.always\ `\ X)$
by (*rule iffD1*[*OF compl-eq-compl-iff*])
   (*simp add*: *tls.always-def image-image tls.eventually.Sup uminus-Inf*)

lemma *inf*:
  shows $\Box(P \sqcap Q) = \Box P \sqcap \Box Q$
by (*simp add*: *tls.always-def tls.eventually.sup*)

lemma *neg*:
  shows $-\Box P = \Diamond(-P)$
by (*simp add*: *tls.always-def*)

lemma *always-necessitation*:
  assumes $\models P$
  shows $\models \Box P$
using *assms* **by** (*simp add*: *tls.valid-def tls.always.top*)

lemma *valid-conv*:
  shows $\models \Box P \longleftrightarrow \models P$
by (*simp add*: *tls.valid-def tls.always.top-conv*)

lemma *always-imp-le*:
  shows $P \longrightarrow_\Box Q \leq \Box P \longrightarrow_B \Box Q$
by (*simp add*: *galois.conj-imp.lower-upper-contractive tls.always.mono*
      flip: *galois.conj-imp.galois tls.always.inf*)

lemma *eventually-le*:
shows $\Box P \leq \Diamond P$
using *tls.always.contractive tls.eventually.cl tls.eventually.mono* **by** *blast*

lemma *not-until-le*: — Warford et al. (2020, (81))
shows $\Box P \leq -(Q\ \mathcal{U}\ (-P))$
by (*simp add*: *compl-le-swap1 tls.always.neg tls.until.eventually-le*)

lemmas *simps* =
tls.always.bot
tls.always.top
tls.always.contractive
tls.always-alt-def[*symmetric*]

setup ‹*Sign.parent-path*›

lemma *until-alwaysR-le*: — Warford et al. (2020, (140))
shows $P\ \mathcal{U}\ \Box Q \leq \Box(P\ \mathcal{U}\ Q)$
by *transfer* (*rule raw.until.alwaysR-le*)

lemma *until-alwaysR*: — Warford et al. (2020, (141))
shows $P\ \mathcal{U}\ \Box P = \Box P$
by (*simp add*: *order.eq-iff order.trans*[*OF tls.until-alwaysR-le*] *tls.until.simps*)

lemma *eventually-always-always-eventually-le*: — Warford et al. (2020, (145))
shows $\Diamond\Box P \leq \Box\Diamond P$
by (*simp add*: *tls.eventually-def tls.until-alwaysR-le*)

lemma *always-inf-eventually-eventually-le*:
shows $\Box P \sqcap \Diamond Q \leq \Diamond(P \sqcap Q)$
by (*simp add*: *shunt1 order.trans*[*OF - tls.eventually.always-imp-le*] *boolean-implication.mp*
tls.always.mono
flip: *boolean-implication-def*)

lemma *always-always-imp*: — Kröger and Merz (2008, §2.2: T33 frame)
shows $\models \Box P \longrightarrow_B \Box Q \longrightarrow_B \Box(P \sqcap Q)$
by (*simp add*: *tls.validI tls.always.inf flip*: *boolean-implication.infL*)

lemma *always-eventually-imp*: — Kröger and Merz (2008, §2.2: T34 frame)
shows $\models \Box P \longrightarrow_B \Diamond Q \longrightarrow_B \Diamond(P \sqcap Q)$
by (*simp add*: *tls.validI boolean-implication.mp tls.always-inf-eventually-eventually-le*)

lemma *always-imp-always-generalization*: — Kröger and Merz (2008, §2.2: T35)
shows $\Box P \longrightarrow_\Box Q \leq \Box P \longrightarrow_B \Box Q$
by (*simp add*: *order.trans*[*OF tls.always.always-imp-le*])

lemma *always-imp-eventually-generalization*: — Kröger and Merz (2008, §2.2: T36)
shows $P \longrightarrow_\Box \Diamond Q \leq \Diamond P \longrightarrow_B \Diamond Q$
by (*metis tls.eventually.always-imp-le tls.eventually.idempotent*(*1*))

The following show that there is no point nesting *tls.always* and *tls.eventually* more than two deep.

lemma *always-eventually-always-absorption*: — Kröger and Merz (2008, §2.2: T37)
shows $\Diamond\Box\Diamond P = \Box\Diamond P$
by (*metis order.eq-iff tls.eventually.expansive tls.eventually.idempotent*(*1*)
tls.eventually-always-always-eventually-le)

lemma *eventually-always-eventually-absorption*: — Kröger and Merz (2008, §2.2: T38)

shows $\Box\Diamond\Box P = \Diamond\Box P$
by (*metis tls.always.neg tls.always-def tls.always-eventually-always-absorption*)

lemma *always-imp-always-eventually-le*: — Warford et al. (2020, (157))
shows $P \longrightarrow_{\Box} Q \leq \Box\Diamond P \longrightarrow_B \Box\Diamond Q$
by (*simp add*: *order.trans*[*OF - tls.always.always-imp-le*]
order.trans[*OF - tls.always.mono*[*OF tls.eventually.always-imp-le*]])

lemma *always-imp-eventually-always-le*: — Warford et al. (2020, (158))
shows $P \longrightarrow_{\Box} Q \leq \Diamond\Box P \longrightarrow_B \Diamond\Box Q$
by (*simp add*: *order.trans*[*OF - tls.eventually.always-imp-le*]
order.trans[*OF - tls.always.mono*[*OF tls.always.always-imp-le*]])

lemma *always-eventually-inf-le*:
shows $\Box\Diamond(P \sqcap Q) \leq \Box\Diamond P \sqcap \Box\Diamond Q$ — Warford et al. (2020, (159))
by (*simp add*: *tls.always.mono tls.eventually.mono*)

lemma *eventually-always-sup-le*:
shows $\Diamond\Box P \sqcap \Diamond\Box Q \leq \Diamond\Box(P \sqcup Q)$ — Warford et al. (2020, (160))
by (*simp add*: *le-infI2 tls.always.mono tls.eventually.mono*)

lemma *always-eventually-sup*: — Warford et al. (2020, (161))
fixes $P :: ('a, 's, 'v)$ *tls*
shows $\Box\Diamond(P \sqcup Q) = \Box\Diamond P \sqcup \Box\Diamond Q$ (**is** *?lhs* = *?rhs*)
proof(*rule antisym*)
show *?lhs* $\leq$ *?rhs*
proof *transfer*
fix $P\ Q :: ('a, 's, 'v)$ *behavior.t set*
have $\exists\omega'{\in}P.\ \exists i.$ *behavior.dropn* $i\ \omega_j = Some\ \omega'$
if $\forall i\ \omega'.$ *behavior.dropn* $i\ \omega = Some\ \omega' \longrightarrow (\exists\omega''{\in}P \cup Q.\ \exists i.$ *behavior.dropn* $i\ \omega' = Some\ \omega'')$
and *behavior.dropn* $i\ \omega = Some\ \omega_i$
and $\forall\omega'{\in}Q.\ \forall i.$ *behavior.dropn* $i\ \omega_i \neq Some\ \omega'$
and *behavior.dropn* $j\ \omega = Some\ \omega_j$
for $\omega\ i\ j\ \omega_i\ \omega_j$
using *spec*[**where** *x*=*max i j*, *OF that*(*1*)] *that*(*2,3,4*)
by (*clarsimp simp*: *nat-le-iff-add split*: *split-asm-max*;
metis add-diff-inverse-nat behavior.dropn.dropn bind.bind-lunit order.asym)
then show *raw.always* (*raw.eventually* $(P \cup Q)$)
$\subseteq$ *raw.always* (*raw.eventually* P) $\cup$ *raw.always* (*raw.eventually* Q)
by (*clarsimp simp*: *raw.eventually-alt-def raw.always-alt-def*)
qed
show *?rhs* $\leq$ *?lhs*
by (*simp add*: *tls.eventually.sup order.trans*[*OF - tls.always.sup-le*])
qed

lemma *eventually-always-inf*: — Warford et al. (2020, (162))
shows $\Diamond\Box(P \sqcap Q) = \Diamond\Box P \sqcap \Diamond\Box Q$
by (*subst compl-eq-compl-iff*[*symmetric*])
(*simp add*: *tls.always.neg tls.always-eventually-sup tls.eventually.neg*)

lemma *eventual-latching*: — Warford et al. (2020, (163))
shows $\Diamond\Box(P \longrightarrow_B \Box Q) = \Diamond\Box(-P) \sqcup \Diamond\Box Q$ (**is** *?lhs* = *?rhs*)
proof(*rule antisym*)
show *?lhs* $\leq$ *?rhs*
by (*rule order.trans*[*OF tls.eventually.mono*[*OF tls.always-imp-always-eventually-le*]])
(*simp add*: *boolean-implication.conv-sup tls.always.neg*
tls.eventually.neg tls.eventually.sup tls.eventually-always-eventually-absorption)
have $\Diamond\Box Q \leq \Diamond\Box(-P \sqcup \Box Q)$

```
    apply (rule order.trans[OF tls.eventually.mono[OF eq-refl[OF tls.always.idempotent[symmetric]]]])
    apply (rule tls.eventually.mono[OF tls.always.mono])
    apply simp
    done
  then show ?rhs ≤ ?lhs
    by (simp add: le-sup-iff boolean-implication.conv-sup monoD)
qed

setup ‹Sign.mandatory-path unless›

lemma transfer[transfer-rule]:
  shows rel-fun (pcr-tls (=) (=) (=)) (rel-fun (pcr-tls (=) (=) (=)) (pcr-tls (=) (=) (=)))
              (λP Q. raw.until P Q ∪ raw.always P)
              tls.unless
unfolding tls.unless-def by transfer-prover

lemma neg: — Warford et al. (2020, (170))
  shows −(P 𝒲 Q) = (−Q) 𝒰 (−P ⊓ −Q)
by transfer (rule raw.unless.neg)

lemma alwaysR-le: — Warford et al. (2020, (177))
  shows P 𝒲 □Q ≤ □(P 𝒲 Q)
by (simp add: tls.unless-def order.trans[OF tls.until-alwaysR-le] tls.always.mono
          order.trans[OF - tls.always.sup-le])

lemma sup-le: — Warford et al. (2020, (206))
  shows P 𝒲 Q ≤ P ⊔ Q
by (rule iffD1[OF compl-le-compl-iff]) (simp add: tls.unless.neg tls.until.expansive)

lemma ordering: — Warford et al. (2020, (252))
  shows ⊨ (−P) 𝒲 Q ⊔ (−Q) 𝒲 P
by (simp add: ac-simps tls.validI tls.unless-def tls.until.ordering tls.eventually.sup flip: tls.eventually.neg)

setup ‹Sign.parent-path›

setup ‹Sign.mandatory-path until›

lemma eq-unless-inf-eventually:
  shows P 𝒰 Q = (P 𝒲 Q) ⊓ ◇Q
by transfer (force simp: raw.until-def raw.eventually-def raw.always-alt-def behavior.dropn.shorterD)

lemma always-strengthen-le: — Warford et al. (2020, (83))
  shows □P ⊓ (Q 𝒰 R) ≤ (P ⊓ Q) 𝒰 (P ⊓ R)
by transfer
   (clarsimp simp: raw.until-def raw.always-alt-def;
    fastforce simp: behavior.dropn.shorterD del: exI intro!: exI)

lemma until-weakI:
  shows □P ⊓ ◇Q ≤ P 𝒰 Q (is ?lhs ≤ ?rhs) — Warford et al. (2020, (84))
by (simp add: tls.eventually-def order.trans[OF tls.until.always-strengthen-le] tls.until.mono)

lemma always-impL: — Warford et al. (2020, (86))
  shows P ⟶□ P' ⊓ P 𝒰 Q ≤ P' 𝒰 Q (is ?thesis1)
    and P 𝒰 Q ⊓ P ⟶□ P' ≤ P' 𝒰 Q (is ?thesis2)
proof −
  show ?thesis1
    by (rule order.trans[OF tls.until.always-strengthen-le])
       (simp add: tls.until.mono boolean-implication.shunt1)
```

then show *?thesis2*
by (*simp add*: *inf-commute*)
qed

lemma *always-impR*: — Warford et al. (2020, (85))
shows $Q \longrightarrow_\Box Q' \sqcap P \mathcal{U} Q \leq P \mathcal{U} Q'$ (**is** *?thesis1*)
and $P \mathcal{U} Q \sqcap Q \longrightarrow_\Box Q' \leq P \mathcal{U} Q'$ (**is** *?thesis2*)
proof −
show *?thesis1*
by (*rule order.trans*[*OF tls.until.always-strengthen-le*])
(*simp add*: *tls.until.mono boolean-implication.shunt1*)
then show *?thesis2*
by (*simp add*: *inf-commute*)
qed

lemma *neg*: — Warford et al. (2020, (173))
shows $-(P \mathcal{U} Q) = (-Q) \mathcal{W} (-P \sqcap -Q)$
unfolding *tls.unless-def*
by (*simp flip*: *tls.until.eq-unless-inf-eventually tls.unless.neg tls.eventually.neg*
boolean-algebra.de-Morgan-conj)

setup ‹*Sign.parent-path*›

setup ‹*Sign.mandatory-path state-prop*›

lemmas *monotone* = *raw.state-prop.monotone*[*transferred*]
lemmas *strengthen*[*strg*] = *st-monotone*[*OF tls.state-prop.monotone*]
lemmas *mono* = *monoD*[*OF tls.state-prop.monotone*]

lemma *Sup*:
shows *tls.state-prop* $(\bigsqcup X) = \bigsqcup$ (*tls.state-prop* ‘ X)
by *transfer* (*simp add*: *raw.state-prop.Sup behavior.stuttering.cl-bot*)

lemma *Inf*:
shows *tls.state-prop* $(\bigsqcap X) = \bigsqcap$ (*tls.state-prop* ‘ X)
by *transfer* (*simp add*: *raw.state-prop.Inf*)

lemmas *simps* = *raw.state-prop.simps*[*transferred*]

setup ‹*Sign.parent-path*›

setup ‹*Sign.mandatory-path terminated*›

lemma *not-bot*:
shows *tls.terminated* $\neq \bot$
by *transfer*
(*simp add*: *raw.terminated-def exI*[**where** *x*=*behavior.B undefined* (*TNil undefined*)] *behavior.sset.simps*)

lemma *not-top*:
shows *tls.terminated* $\neq \top$
by *transfer*
(*fastforce simp*: *raw.terminated-def*
dest: *subsetD*[*OF Set.equalityD2*, **where** *c*=*behavior.B undefined* (*trepeat* (*undefined*, *undefined*))])

lemma *inf-always*:
shows *tls.terminated* $\sqcap \Box P =$ *tls.terminated* $\sqcap P$
by (*rule antisym*[*OF inf.mono*[*OF order.refl tls.always.contractive*]])
(*transfer*; *simp add*: *raw.terminated.inf-always-le*)

lemma *always-le-conv*:
  shows *tls.terminated* $\leq \Box P \longleftrightarrow$ *tls.terminated* $\leq P$
by (*simp add*: *inf.order-iff tls.terminated.inf-always*)

lemma *inf-eventually*:
  shows *tls.terminated* $\sqcap \Diamond P =$ *tls.terminated* $\sqcap P$ (**is** *?lhs* = *?rhs*)
proof(*rule antisym*[*OF - inf.mono*[*OF order.refl tls.eventually.expansive*]])
  have *tls.terminated* $\sqcap -P \leq$ *tls.terminated* $\sqcap -\Diamond P$
    by (*simp add*: *tls.terminated.inf-always tls.eventually.neg*)
  then show *?lhs* $\leq$ *?rhs*
    by (*simp add*: *boolean-implication.shunt1 boolean-implication.imp-trivialI*)
qed

lemma *eventually-le-conv*:
  shows *tls.terminated* $\leq$ *tls.eventually* $P \longleftrightarrow$ *tls.terminated* $\leq P$
by (*simp add*: *inf.order-iff tls.terminated.inf-eventually*)

lemma *eq-always-terminated*:
  shows *tls.terminated* $= \Box$*tls.terminated*
by (*rule order.antisym*[*OF - tls.always.contractive*])
   (*simp add*: *tls.terminated.always-le-conv*)

setup ‹*Sign.parent-path*›

16.5.1 Leads-to and leads-to-via

So-called *response* properties are of the form $P \longrightarrow_\Box \Diamond Q$ (pronounced "P leads to Q", written $P \rightsquigarrow Q$) (Manna and Pnueli 1991). This connective is similar to the "ensures" modality of Chandy and Misra (1989, §3.4.4). Jackson (1998) used the more general "P leads to Q via I" form $P \longrightarrow_\Box I \mathcal{U} Q$ to establish liveness properties in a sequential setting.

lemma *leads-to-refl*:
  shows $\models P \rightsquigarrow P$
by (*simp add*: *tls.validI boolean-implication.shunt-top tls.always.top-conv tls.eventually.expansive*
        top.extremum-unique)

lemma *leads-to-mono*:
  assumes $P' \leq P$
  assumes $Q \leq Q'$
  shows $P \rightsquigarrow Q \leq P' \rightsquigarrow Q'$
by (*simp add*: *assms boolean-implication.mono tls.always.mono tls.eventually.mono*)

lemma *leads-to-supL*:
  shows $(P \rightsquigarrow R) \sqcap (Q \rightsquigarrow R) \leq (P \sqcup Q) \rightsquigarrow R$
by (*simp add*: *boolean-implication.conv-sup sup-inf-distrib2 tls.always.inf*)

lemma *always-imp-leads-to*:
  shows $P \longrightarrow_\Box Q \leq P \rightsquigarrow Q$
by (*simp add*: *boolean-implication.mono tls.always.mono tls.eventually.expansive*)

lemma *leads-to-eventually*:
  shows $\Diamond P \sqcap (P \rightsquigarrow Q) \leq \Diamond Q$
by (*simp add*: *galois.conj-imp.galois tls.always-imp-eventually-generalization*)

lemma *leads-to-leads-to-via*:
  shows $P \longrightarrow_\Box Q \mathcal{U} R \leq P \rightsquigarrow R$
by (*simp add*: *boolean-implication.mono tls.always.mono tls.until.eventually-le*)

lemma *leads-to-trans*:
shows $P \leadsto Q \sqcap Q \leadsto R \leq P \leadsto R$ (**is** *?lhs* $\leq$ *?rhs*)
proof −
have *?lhs* $\leq P \leadsto Q \sqcap \Box(Q \leadsto R)$
by (*simp add*: *tls.always.simps*)
also have $\ldots \leq P \leadsto Q \sqcap \Diamond Q \leadsto R$
by (*meson order.refl inf-mono tls.always.mono tls.always-imp-eventually-generalization*)
also have $\ldots \leq$ *?rhs*
by (*simp add*: *boolean-implication.trans tls.always.mono flip*: *tls.always.inf*)
finally show *?thesis* **.**
qed

lemma *leads-to-via-weakenR*:
shows $Q \longrightarrow_\Box Q' \sqcap P \longrightarrow_\Box I \mathcal{U} Q \leq P \longrightarrow_\Box I \mathcal{U} Q'$
by *transfer*
(*clarsimp simp*: *raw.always-alt-def raw.until-def boolean-implication.set-alt-def*;
metis behavior.dropn.dropn Option.bind.bind-lunit)

lemma *leads-to-via-supL*: — useful for case distinctions
shows $P \longrightarrow_\Box I \mathcal{U} Q \sqcap P' \longrightarrow_\Box I' \mathcal{U} Q \leq P \sqcup P' \longrightarrow_\Box (I \sqcup I') \mathcal{U} Q$
by (*simp add*: *boolean-implication.conv-sup ac-simps le-infI2 le-supI2*
monoD[*OF tls.always.monotone*] *tls.until.mono*)

lemma *leads-to-via-trans*:
shows $(P \longrightarrow_\Box I \mathcal{U} Q) \sqcap (Q \longrightarrow_\Box I' \mathcal{U} R) \leq P \longrightarrow_\Box (I \sqcup I') \mathcal{U} R$ (**is** *?lhs* $\leq$ *?rhs*)
proof −
have *?lhs* $\leq \Box(P \longrightarrow_B I \mathcal{U} (I' \mathcal{U} R))$
by (*subst inf.commute*) (*rule tls.leads-to-via-weakenR*)
also have $\ldots \leq$ *?rhs*
by (*strengthen ord-to-strengthen*(*1*)[*OF tls.until.supL-ordering-le*]) (*rule order.refl*)
finally show *?thesis* **.**
qed

lemma *leads-to-via-disj*: — more like a chaining rule
shows $(P \longrightarrow_\Box I \mathcal{U} Q) \sqcap (Q \longrightarrow_\Box I' \mathcal{U} R) \leq (P \sqcup Q \longrightarrow_\Box (I \sqcup I') \mathcal{U} R)$
by (*simp add*: *boolean-implication-def inf.coboundedI2 le-supI2 tls.always.mono tls.until.mono*)

16.5.2 Fairness

A few renderings of weak fairness. van Glabbeek and Höfner (2019) call this "response to insistence" as a generalisation of weak fairness.

definition *weakly-fair* :: ('*a*, '*s*, '*v*) *tls* ⇒ ('*a*, '*s*, '*v*) *tls* ⇒ ('*a*, '*s*, '*v*) *tls* **where**
weakly-fair enabled taken = $\Box$*enabled* $\longrightarrow_\Box \Diamond$*taken*

lemma *weakly-fair-def2*:
shows *tls.weakly-fair enabled taken* = $\Box(-(\Box(\mathit{enabled} \sqcap -\mathit{taken})))$
by (*simp add*: *tls.weakly-fair-def tls.always-def tls.eventually.sup*)

lemma *weakly-fair-def3*:
shows *tls.weakly-fair enabled taken* = $\Diamond\Box$*enabled* $\longrightarrow_B \Box\Diamond$*taken*
by (*simp add*: *tls.weakly-fair-def boolean-implication.conv-sup*
tls.always.neg tls.always-eventually-sup tls.eventually.neg
flip: *tls.eventually.sup*)

lemma *weakly-fair-def4*:
shows *tls.weakly-fair enabled taken* = $\Box\Diamond(\mathit{enabled} \longrightarrow_B \mathit{taken})$
by (*simp add*: *tls.weakly-fair-def boolean-implication.conv-sup tls.always.neg tls.eventually.sup*)

setup ‹*Sign.mandatory-path weakly-fair*›

lemma *mono*:
  assumes $P' \leq P$
  assumes $Q \leq Q'$
  shows *tls.weakly-fair P Q* $\leq$ *tls.weakly-fair P' Q'*
unfolding *tls.weakly-fair-def*
apply (*strengthen ord-to-strengthen*(*1*)[*OF assms*(*1*)])
apply (*strengthen ord-to-strengthen*(*1*)[*OF assms*(*2*)])
apply (*rule order.refl*)
done

lemma *strengthen*[*strg*]:
  assumes *st-ord* ($\neg F$) *P P'*
  assumes *st-ord F Q Q'*
  shows *st-ord F* (*tls.weakly-fair P Q*) (*tls.weakly-fair P' Q'*)
using *assms* **by** (*cases F*) (*auto simp*: *tls.weakly-fair.mono*)

lemma *weakly-fair-triv*:
  shows $\Box\Diamond(-enabled) \leq$ *tls.weakly-fair enabled taken*
by (*simp add*: *tls.weakly-fair-def3 boolean-implication.conv-sup tls.always.neg tls.eventually.neg*)

lemma *mp*:
  shows *tls.weakly-fair enabled taken* $\sqcap$ $\Box$*enabled* $\leq$ $\Diamond$*taken*
by (*simp add*: *tls.weakly-fair-def boolean-implication.shunt1 tls.always.contractive*)

setup ‹*Sign.parent-path*›

setup ‹*Sign.mandatory-path always*›

lemma *weakly-fair*:
  shows $\Box$(*tls.weakly-fair enabled taken*) = *tls.weakly-fair enabled taken*
by (*simp add*: *tls.weakly-fair-def tls.always.simps*)

setup ‹*Sign.parent-path*›

setup ‹*Sign.mandatory-path eventually*›

lemma *weakly-fair*:
  shows $\Diamond$(*tls.weakly-fair enabled taken*) = *tls.weakly-fair enabled taken*
by (*simp add*: *tls.weakly-fair-def4 tls.always-eventually-always-absorption*)

setup ‹*Sign.parent-path*›

Similarly for strong fairness. van Glabbeek and Höfner (2019) call this "response to persistence" as a generalisation of strong fairness.

definition *strongly-fair* :: ('*a*, '*s*, '*v*) *tls* $\Rightarrow$ ('*a*, '*s*, '*v*) *tls* $\Rightarrow$ ('*a*, '*s*, '*v*) *tls* **where**
  strongly-fair enabled taken = $\Box\Diamond$*enabled* $\longrightarrow_{\Box}$ $\Diamond$*taken*

lemma *strongly-fair-def2*:
  shows *tls.strongly-fair enabled taken* = $\Box(-\Box(\Diamond enabled \sqcap -taken))$
by (*simp add*: *tls.strongly-fair-def boolean-implication.conv-sup tls.always.neg tls.eventually.sup*)

lemma *strongly-fair-def3*:
  shows *tls.strongly-fair enabled taken* = $\Box\Diamond$*enabled* $\longrightarrow_B$ $\Box\Diamond$*taken*
by (*simp add*: *tls.strongly-fair-def boolean-implication.conv-sup tls.always.neg tls.eventually.neg*
    tls.always-eventually-sup tls.eventually-always-eventually-absorption

flip: *tls.eventually.sup*)

setup ‹*Sign.mandatory-path strongly-fair*›

lemma *mono*:
assumes $P' \leq P$
assumes $Q \leq Q'$
shows *tls.strongly-fair P Q* $\leq$ *tls.strongly-fair P' Q'*
unfolding *tls.strongly-fair-def*
apply (*strengthen ord-to-strengthen*(*1*)[*OF assms*(*1*)])
apply (*strengthen ord-to-strengthen*(*1*)[*OF assms*(*2*)])
apply (*rule order.refl*)
done

lemma *strengthen*[*strg*]:
assumes *st-ord* (¬*F*) *P P'*
assumes *st-ord F Q Q'*
shows *st-ord F* (*tls.strongly-fair P Q*) (*tls.strongly-fair P' Q'*)
using *assms* **by** (*cases F*) (*auto simp*: *tls.strongly-fair.mono*)

lemma *supL*: — does not hold for *tls.weakly-fair*
shows *tls.strongly-fair* (*enabled1* ⊔ *enabled2*) *taken*
= (*tls.strongly-fair enabled1 taken* ⊓ *tls.strongly-fair enabled2 taken*)
by (*simp add*: *boolean-implication.conv-sup sup-inf-distrib2 tls.always.inf tls.always-eventually-sup*
tls.strongly-fair-def)

lemma *weakly-fair-le*:
shows *tls.strongly-fair enabled taken* $\leq$ *tls.weakly-fair enabled taken*
by (*simp add*: *tls.strongly-fair-def3 tls.weakly-fair-def3 boolean-implication.mono*
tls.eventually-always-always-eventually-le)

lemma *always-enabled-weakly-fair-strongly-fair*:
shows □*enabled* $\leq$ *tls.weakly-fair enabled taken* $\longleftrightarrow_B$ *tls.strongly-fair enabled taken*
by (*simp add*: *boolean-eq-def boolean-implication-def*)

setup ‹*Sign.parent-path*›

setup ‹*Sign.mandatory-path always*›

lemma *strongly-fair*:
shows □(*tls.strongly-fair enabled taken*) = *tls.strongly-fair enabled taken*
by (*simp add*: *tls.strongly-fair-def*)

setup ‹*Sign.parent-path*›

setup ‹*Sign.mandatory-path eventually*›

lemma *strongly-fair*:
shows ◇(*tls.strongly-fair enabled taken*) = *tls.strongly-fair enabled taken*
by (*simp add*: *tls.strongly-fair-def2 tls.always.neg tls.always-eventually-always-absorption*)

setup ‹*Sign.parent-path*›

setup ‹*Sign.parent-path*›

16.6 Safety Properties

We now carve the safety properties out of the (*'a*, *'s*, *'v*) *tls* lattice.

References:

- Alpern and Schneider (1985); Alpern, Demers, and Schneider (1986); Schneider (1987, §2)
  - observes that Lamport's earlier definitions do not work without stuttering
  - provides the now standard definition that works with and without stuttering
- Abadi and Lamport (1991, §2.2): topological definitions and intuitions
- Sistla (1994, §2.2)

We go a different way: we establish a Galois connection with ('*a*, '*s*, '*v*) *spec*.
Observations:

- our safety closure for ('*a*, '*s*, '*v*) *tls* introduces infinite sequences to stand for the prefixes in ('*a*, '*s*, '*v*) *spec*
  - i.e., the non-termination of trace σ (*trace.term* σ = *None*) is represented by a behavior ending with *trace.final* σ infinitely stuttered
  - Abadi and Lamport (1991, §2.1) consider these behaviors to represent terminating processes

setup ‹*Sign.mandatory-path raw*›

definition *to-spec* :: ('*a*, '*s*, '*v*) *behavior.t set* ⇒ ('*a*, '*s*, '*v*) *trace.t set* **where**
to-spec T = {*behavior.take i ω* |*ω i*. *ω* ∈ *T*}

definition *from-spec* :: ('*a*, '*s*, '*v*) *trace.t set* ⇒ ('*a*, '*s*, '*v*) *behavior.t set* **where**
from-spec S = {*ω* . ∀ *i*. *behavior.take i ω* ∈ *S*}

interpretation *safety*: *galois.powerset raw.to-spec raw.from-spec*
by *standard* (*fastforce simp*: *raw.to-spec-def raw.from-spec-def*)

setup ‹*Sign.mandatory-path from-spec*›

lemma *empty*:
shows *raw.from-spec* {} = {}
by (*simp add*: *raw.from-spec-def*)

lemma *singleton*:
shows *raw.from-spec* (*Safety-Logic.raw.singleton σ*)
= ⋃(*raw.singleton* ‘ {*ω* . ∀ *i*. *behavior.take i ω* ∈ *Safety-Logic.raw.singleton σ*}) (**is** *?lhs* = *?rhs*)
proof(*rule antisym*)
show *?lhs* ⊆ *?rhs* **by** (*force simp*: *raw.from-spec-def TLS.raw.singleton-def*)
show *?rhs* ⊆ *?lhs*
by (*clarsimp simp*: *raw.from-spec-def TLS.raw.singleton-def Safety-Logic.raw.singleton-def*
elim!: *behavior.stuttering.clE*)
(*metis behavior.stuttering.equiv.takeE raw.spec.closed raw.spec.closed.stuttering-closed*
trace.stuttering.clI trace.stuttering.closed-conv)
qed

lemma *sup*:
assumes *P* ∈ *raw.spec.closed*
assumes *Q* ∈ *raw.spec.closed*
shows *raw.from-spec* (*P* ∪ *Q*) = *raw.from-spec P* ∪ *raw.from-spec Q*
by (*rule antisym*[*OF - raw.safety.sup-upper-le*])
(*clarsimp simp*: *raw.from-spec-def*;
meson behavior.take.mono downwards.closed-in linorder-le-cases
raw.spec.closed.downwards-closed[*OF assms*(*1*)] *raw.spec.closed.downwards-closed*[*OF assms*(*2*)])

setup ‹*Sign.parent-path*›

setup ‹*Sign.mandatory-path to-spec*›

lemma *singleton*:
  shows *raw.to-spec* (*TLS.raw.singleton* ω)
      = ($\bigcup i$. *Safety-Logic.raw.singleton* (*behavior.take i* ω)) (**is** *?lhs* = *?rhs*)
proof(*rule antisym*)
  show *?lhs* $\subseteq$ *?rhs*
    by (*fastforce simp*: *TLS.raw.singleton-def raw.to-spec-def*
                Safety-Logic.raw.singleton-def raw.spec.cl-def
              elim: *behavior.stuttering.clE behavior.stuttering.equiv.takeE*[*OF sym*]
                trace.stuttering.clI[*OF - sym, rotated*])
  show *?rhs* $\subseteq$ *?lhs*
    by (*fastforce simp*: *Safety-Logic.raw.singleton-def raw.spec.cl-def TLS.raw.singleton-def*
                raw.to-spec-def trace.less-eq-take-def trace.take.behavior.take
              elim: *downwards.clE trace.stuttering.clE trace.stuttering.equiv.behavior.takeE*)
qed

setup ‹*Sign.parent-path*›

setup ‹*Sign.mandatory-path safety*›

lemma *cl-altI*:
  assumes $\bigwedge i$. $\exists \omega' \in P$. *behavior.take i* ω = *behavior.take i* ω'
  shows $\omega \in$ *raw.safety.cl P*
using *assms* **by** (*fastforce simp*: *raw.safety.cl-def raw.from-spec-def raw.to-spec-def*)

lemma *cl-altE*:
  assumes $\omega \in$ *raw.safety.cl P*
  obtains ω' **where** $\omega' \in P$ **and** *behavior.take i* ω = *behavior.take i* ω'
proof(*atomize-elim*, *cases enat i* $\leq$ *tlength* (*behavior.rest* ω))
  case *True* **with** *assms* **show** $\exists \omega'$. $\omega' \in P \wedge$ *behavior.take i* ω = *behavior.take i* ω'
    by (*clarsimp simp*: *raw.safety.cl-def raw.from-spec-def raw.to-spec-def*)
      (*metis behavior.take.length behavior.take.sel*(*3*) *ttake-eq-None-conv*(*1*)
          min.absorb2 min-enat2-conv-enat the-enat.simps)
next
  case *False* **with** *assms* **show** $\exists \omega'$. $\omega' \in P \wedge$ *behavior.take i* ω = *behavior.take i* ω'
    by (*clarsimp simp*: *raw.safety.cl-def raw.from-spec-def raw.to-spec-def*)
      (*metis behavior.continue.take-drop-id behavior.take.continue-id leI*)
qed

lemma *cl-alt-def*: — Alpern et al. (1986): the classical definition: ω belongs to the safety closure of P if every prefix of ω can be extended to a behavior in P
  shows *raw.safety.cl P* = $\{\omega$. $\forall i$. $\exists \beta$. *behavior.take i* ω @$-_B$ $\beta \in P\}$ (**is** *?lhs* = *?rhs*)
proof(*rule antisym*)
  show *?lhs* $\subseteq$ *?rhs*
    by *clarsimp* (*metis behavior.continue.take-drop-id raw.safety.cl-altE*)
  show *?rhs* $\subseteq$ *?lhs*
  proof(*clarify intro*!: *raw.safety.cl-altI*)
    fix ω *i*
    assume $\forall j$. $\exists \beta$. *behavior.take j* ω @$-_B$ $\beta \in P$
    then show $\exists \omega' \in P$. *behavior.take i* ω = *behavior.take i* ω'
      by (*force dest*: *spec*[**where** *x*=*i*]
            intro: *exI*[**where** *x*=*i*] *rev-bexI*
             simp: *behavior.take.continue trace.take.behavior.take trace.continue.self-conv*
                ttake-eq-None-conv length-ttake
            split: *option.split enat.split*)

qed
qed

lemma *closed-alt-def*: — If ω is not in P then some prefix of ω has irretrievably gone wrong
shows *raw.safety.closed* = $\{P.\ \forall\omega.\ \omega \notin P \longrightarrow (\exists i.\ \forall\beta.\ behavior.take\ i\ \omega\ @{-}_B\ \beta \notin P)\}$
unfolding *raw.safety.closed-def raw.safety.cl-alt-def* **by** *fast*

lemma *closed-alt-def2*: — Contraposition gives the customary prefix-closure definition
shows *raw.safety.closed* = $\{P.\ \forall\omega.\ (\forall i.\ \exists\beta.\ behavior.take\ i\ \omega\ @{-}_B\ \beta \in P) \longrightarrow \omega \in P\}$
unfolding *raw.safety.closed-alt-def* **by** *fast*

lemma *closedI2*:
assumes $\bigwedge\omega.\ (\bigwedge i.\ \exists\beta.\ behavior.take\ i\ \omega\ @{-}_B\ \beta \in P) \Longrightarrow \omega \in P$
shows $P \in raw.safety.closed$
using *assms* **unfolding** *raw.safety.closed-alt-def2* **by** *fast*

lemma *closedE2*:
assumes $P \in raw.safety.closed$
assumes $\bigwedge i.\ \omega \notin P \Longrightarrow \exists\beta.\ behavior.take\ i\ \omega\ @{-}_B\ \beta \in P$
shows $\omega \in P$
using *assms* **unfolding** *raw.safety.closed-alt-def2* **by** *blast*

setup ‹*Sign.mandatory-path cl*›

lemma *state-prop*:
shows *raw.safety.cl* (*raw.state-prop P*) = *raw.state-prop P*
by (*simp add*: *raw.safety.cl-alt-def raw.state-prop-def*)

lemma *terminated-iff*:
assumes $\omega \in raw.terminated$
shows $\omega \in raw.safety.cl\ P \longleftrightarrow \omega \in P$ (**is** *?lhs* $\longleftrightarrow$ *?rhs*)
proof(*rule iffI*)
from *assms* **obtain** *i* **where** *tlength* (*behavior.rest* ω) = *enat i*
by (*clarsimp simp*: *raw.terminated-def tfinite-tlength-conv*)
then show *?lhs* $\Longrightarrow$ *?rhs*
by (*metis raw.safety.cl-altE*[**where** *i*=*Suc i*]
behavior.continue.take-drop-id behavior.take.continue-id enat-ord-simps(*2*) *lessI*)
qed (*simp add*: *raw.safety.expansive'*)

lemma *terminated*:
shows *raw.safety.cl raw.terminated* = *raw.idle* $\cup$ *raw.terminated* (**is** *?lhs* = *?rhs*)
proof(*rule antisym*[*OF subsetI subsetI*])
fix ω
assume $\omega \in$ *?lhs*
then have *snd* (*tnth* (*behavior.rest* ω) *i*) = *behavior.init* ω
if *enat i* < *tlength* (*behavior.rest* ω)
for *i*
using *that*
by (*clarsimp simp*: *raw.terminated-def behavior.take-def behavior.split-all behavior.sset.simps*
split-def
simp del: *ttake.simps*
elim!: *raw.safety.cl-altE*[**where** *i*=*Suc i*])
(*metis* (*no-types*, *lifting*) *Suc-ile-eq in-tset-conv-tnth nth-ttake*
doubleton-eq-iff insert-image insert-absorb2 lessI subset-singletonD ttake-eq-None-conv(*1*))
then have *behavior.sset* $\omega \subseteq \{$*behavior.init* $\omega\}$
by (*cases* ω) (*clarsimp simp*: *behavior.sset.simps tset-conv-tnth*)
then show $\omega \in$ *?rhs*
by (*simp add*: *raw.idle-alt-def raw.terminated-def*)

```
next
  show ω ∈ ?lhs if ω ∈ ?rhs for ω
    using that
  proof(cases rule: UnE[consumes 1, case-names idle terminated])
    case idle show ?thesis
    proof(rule raw.safety.cl-altI)
      fix i
      let ?ω' = behavior.take i ω @−_B TNil undefined
      from idle have ?ω' ∈ raw.terminated
        by (auto simp: raw.idle-alt-def raw.terminated-def behavior.sset.continue
                  dest: subsetD[OF behavior.sset.take-le]
                 split: option.split)
      moreover
      from idle have behavior.take i ω = behavior.take i ?ω'
        by (simp add: raw.idle-alt-def behavior.take.continue trace.take.behavior.take
                      length-ttake tfinite-tlength-conv)
      ultimately show ∃ω'∈raw.terminated. behavior.take i ω = behavior.take i ω'
        by blast
    qed
  qed (auto intro: raw.safety.expansive')
qed

lemma le-terminated-bot:
  assumes P ∈ behavior.stuttering.closed
  assumes raw.safety.cl P ⊆ raw.terminated
  shows P = {}
proof(rule ccontr)
  assume ‹P ≠ {}› then obtain ω where ω ∈ P by blast
  let ?ω' = behavior.B (behavior.init ω) (trepeat (undefined, behavior.init ω))
  from ‹ω ∈ P› have ?ω' ∈ raw.safety.cl P
    by (fastforce intro: exI[where x=behavior.rest ω]
                         behavior.stuttering.f-closedI[OF ‹P ∈ behavior.stuttering.closed›]
                  simp: raw.safety.cl-alt-def behavior.take.trepeat behavior.continue.simps
                         behavior.natural.tshift collapse.tshift trace.natural'.replicate
                         trace.final'.replicate
                         behavior.stuttering.f-closed[OF ‹P ∈ behavior.stuttering.closed›]
             simp flip: behavior.natural-def)
  moreover have ?ω' ∉ raw.terminated
    by (simp add: raw.terminated-def)
  moreover note ‹raw.safety.cl P ⊆ raw.terminated›
  ultimately show False by blast
qed

lemma always-le:
  shows raw.safety.cl (raw.always P) ⊆ raw.always (raw.safety.cl P)
unfolding raw.always-alt-def raw.safety.cl-alt-def subset-iff mem-Collect-eq
proof(intro allI impI)
  fix ω i ω' j
  assume ∗: ∀ i. ∃ β. ∀ k ω'. behavior.dropn k (behavior.take i ω @−_B β) = Some ω' ⟶ ω' ∈ P
     and ∗∗: behavior.dropn i ω = Some ω'
  from spec[where x=i + j, OF ∗] ∗∗ behavior.take.dropn[OF ∗∗, where j=j]
  show ∃ β. behavior.take j ω' @−_B β ∈ P
    by (clarsimp dest!: spec[where x=i])
       (subst (asm) behavior.dropn.continue-shorter;
        force simp: length-ttake trace.dropn.behavior.take
              dest: behavior.dropn.eq-Some-tlengthD
             split: enat.split)
qed
```

lemma *eventually*:
assumes $P \neq \bot$
shows *raw.safety.cl* (*raw.eventually* P)
$= -$*raw.eventually raw.terminated* $\cup$ *raw.eventually* P (**is** *?lhs* = *?rhs*)
proof(*rule antisym*[*OF subsetI iffD2*[*OF Un-subset-iff*, *simplified conj-explode*, *rule-format*, *OF subsetI*]])
show $\omega \in$ *?rhs* **if** $\omega \in$ *?lhs* **for** ω
proof(*cases tlength* (*behavior.rest* ω))
case (*enat i*) **with** *that* **show** *?thesis*
by (*fastforce dest*: *spec*[**where** *x*=*Suc i*]
simp: *raw.safety.cl-alt-def raw.terminated-def behavior.take.continue-id*)
qed (*simp add*: *raw.eventually.terminated tfinite-tlength-conv*)
from *assms* **obtain** ω_P **where** $\omega_P \in P$ **by** *blast*
show $\omega \in$ *?lhs* **if** $\omega \in -$*raw.eventually raw.terminated* **for** ω
proof(*intro raw.safety.cl-altI exI bexI*)
fix *i*
let *?ω′* = *behavior.take i ω* @$-_B$ *TCons* (*undefined*, *behavior.init* ω_P) (*behavior.rest* ω_P)
from ‹$\omega_P \in P$› ‹$\omega \in -$*raw.eventually raw.terminated*› **show** *?ω′* $\in$ *raw.eventually P*
unfolding *raw.eventually.terminated*
by (*auto intro*!: *exI*[**where** *x*=*Suc i*]
simp: *raw.eventually-alt-def tfinite-tlength-conv behavior.dropn.continue*
length-ttake ttake-eq-None-conv)
from ‹$\omega \in -$*raw.eventually raw.terminated*› **show** *behavior.take i ω* = *behavior.take i ?ω′*
by (*simp add*: *raw.eventually.terminated behavior.take.continue trace.take.behavior.take*
length-ttake tfinite-tlength-conv
split: *enat.split*)
qed
show *raw.eventually P* $\subseteq$ *?lhs*
by (*fast intro*!: *order.trans*[*OF - raw.safety.expansive*])
qed

setup ‹*Sign.parent-path*›

setup ‹*Sign.mandatory-path closed*›

lemma *always-eventually*:
assumes $P \in$ *raw.safety.closed*
assumes $\forall i.\ \exists j{\geq}i.\ \exists \beta.$ *behavior.take j* ω @$-_B$ $\beta \in P$
shows $\omega \in P$
using *assms*(*1*)
proof(*rule raw.safety.closedE2*)
fix *i*
from *spec*[*OF assms*(*2*), **where** *x*=*i*] **obtain** *j* β **where** $i \leq j$ **and** *behavior.take j* ω @$-_B$ $\beta \in P$
by *blast*
then show $\exists \beta.$ *behavior.take i* ω @$-_B$ $\beta \in P$ **if** $\omega \notin P$
using *that*
by (*clarsimp simp*: *tdropn-tshift2 behavior.continue.tshift2 behavior.continue.take-drop-shorter length-ttake*
behavior.continue.term-Some behavior.take.term-Some-conv ttake-eq-Some-conv
split: *enat.split split-min*
intro!: *exI*[**where** *x*=*tdropn i* (*behavior.rest* (*behavior.take j* ω @$-_B$ β))])
qed

lemma *sup*:
assumes $P \in$ *raw.safety.closed*
assumes $Q \in$ *raw.safety.closed*
shows $P \cup Q \in$ *raw.safety.closed*
by (*clarsimp simp*: *raw.safety.closed-alt-def2*)
(*meson assms raw.safety.closed.always-eventually sup.cobounded1 sup.cobounded2*)

lemma *unless*: — Sistla (1994, §3.1) – minimality is irrelevant
assumes $P \in$ *raw.safety.closed*
assumes $Q \in$ *raw.safety.closed*
shows *raw.unless P Q* $\in$ *raw.safety.closed*
proof(*rule raw.safety.closedI2*)
fix ω **assume** $*$: $\exists\,\beta$. *behavior.take i* ω @$-_B$ $\beta \in$ *raw.unless P Q* **for** *i*
show $\omega \in$ *raw.unless P Q*
proof(*cases* $\forall\, i\ j\ \omega'.\ \exists\,\beta$. *behavior.dropn i* ω = *Some* $\omega' \longrightarrow$ *behavior.take j* ω' @$-_B$ $\beta \in P$)
case *True*
with ‹$P \in$ *raw.safety.closed*› **have** *behavior.dropn i* ω = *Some* $\omega' \longrightarrow \omega' \in P$ **for** *i* ω'
by (*blast intro*: *raw.safety.closedE2*)
then show *?thesis*
by (*simp add*: *raw.always-alt-def*)
next
case *False*
then obtain ω' *k l*
where $**$: *behavior.dropn k* ω = *Some* ω' $\forall\,\beta$. *behavior.take l* ω' @$-_B$ $\beta \notin P$
by *clarsimp*
{
fix *i* β
assume *kli*: $k + l \le i$
moreover
note $**$
moreover
from *kli* **have** $\exists\, j.\ i - k = l + j$ **by** *presburger*
moreover
from ‹*behavior.dropn k* ω = *Some* ω'› *kli*
have $***$: *k* $\le$ *length* (*trace.rest* (*behavior.take i* ω))
by (*fastforce simp*: *length-ttake split*: *enat.splits*
dest: *behavior.dropn.eq-Some-tlengthD*)
ultimately have $****$: $\forall\,\omega''$. *behavior.dropn k* (*behavior.take i* ω @$-_B$ β) = *Some* $\omega'' \longrightarrow \omega'' \notin P$
by (*force simp*: *behavior.dropn.continue-shorter trace.dropn.behavior.take behavior.take.add*
simp flip: *behavior.continue.tshift2*)
{
assume *PQ*: *behavior.take i* ω @$-_B$ $\beta \in$ *raw.unless P Q*
from $****$ *PQ* **obtain** *m*
where $m \le k$
and $\forall\,\omega'$. *behavior.dropn m* (*behavior.take i* ω @$-_B$ β) = *Some* $\omega' \longrightarrow \omega' \in Q$
and $\forall\, p{<}m$. ($\forall\,\omega'$. *behavior.dropn p* (*behavior.take i* ω @$-_B$ β) = *Some* $\omega' \longrightarrow \omega' \in P$)
by (*auto 6 0 simp*: *raw.until-def raw.always-alt-def*)
(*metis behavior.dropn.shorterD leI nle-le option.sel*)
with *kli* $***$
have ($\exists\, m{\le}k$. ($\forall\,\omega'$. *behavior.dropn m* ω = *Some* $\omega' \longrightarrow$ *behavior.take* ($i - m$) ω' @$-_B$ $\beta \in Q$)
$\wedge$ ($\forall\, p{<}m$. ($\forall\,\omega'$. *behavior.dropn p* ω = *Some* $\omega' \longrightarrow$ *behavior.take* ($i - p$) ω' @$-_B$ $\beta \in P$)))
by (*clarsimp simp*: *exI*[**where** *x*=*m*] *behavior.dropn.continue-shorter trace.dropn.behavior.take*)
}
}
then have $\forall\, i.\ \exists\, n{\ge}i.\ \exists\, m{\le}k.\ \exists\,\beta$. ($\forall\,\omega'$. *behavior.dropn m* ω = *Some* $\omega' \longrightarrow$ *behavior.take* ($n - m$) ω' @$-_B$ $\beta \in Q$)
$\wedge$ ($\forall\, p{<}m.\ \forall\,\omega'$. *behavior.dropn p* ω = *Some* $\omega' \longrightarrow$ *behavior.take* ($n - p$) ω' @$-_B$ $\beta \in P$)
using $*$ **by** (*metis nle-le*)
then obtain *m*
where $m \le k\ \forall\, i.\ \exists\, n{\ge}i.\ \exists\,\beta$. ($\forall\,\omega'$. *behavior.dropn m* ω = *Some* $\omega' \longrightarrow$ *behavior.take* ($n - m$) ω' @$-_B$ $\beta \in Q$)
$\wedge$ ($\forall\, p{<}m.\ \forall\,\omega'$. *behavior.dropn p* ω = *Some* $\omega' \longrightarrow$ *behavior.take* ($n - p$) ω' @$-_B$ $\beta \in P$)
by (*clarsimp simp*: *always-eventually-pigeonhole*)

```
    with behavior.dropn.shorterD[OF ‹behavior.dropn k ω = Some ω'› ‹m ≤ k›]
        raw.safety.closed.always-eventually[OF ‹P ∈ raw.safety.closed›]
        raw.safety.closed.always-eventually[OF ‹Q ∈ raw.safety.closed›]
    show ω ∈ raw.unless P Q
      apply −
      apply clarsimp
      apply (rule raw.untilI, assumption)
       apply (meson add-le-imp-le-diff)
      apply (metis add-le-imp-le-diff option.sel behavior.dropn.shorterD[OF - less-imp-le])
      done
  qed
qed

setup ‹Sign.parent-path›

setup ‹Sign.parent-path›

setup ‹Sign.parent-path›

setup ‹Sign.mandatory-path downwards.closed›

lemma to-spec:
  shows range raw.to-spec ⊆ downwards.closed
by (fastforce elim: downwards.clE simp: raw.to-spec-def trace.less-eq-take-def trace.take.behavior.take)

setup ‹Sign.parent-path›

setup ‹Sign.mandatory-path trace.stuttering.closed›

lemma to-spec:
  shows raw.to-spec ‘ behavior.stuttering.closed ⊆ trace.stuttering.closed
by (fastforce simp: raw.to-spec-def
              elim: trace.stuttering.clE trace.stuttering.equiv.E trace.stuttering.equiv.behavior.takeE
              dest: behavior.stuttering.closed-in)

setup ‹Sign.parent-path›

setup ‹Sign.mandatory-path raw.spec.closed›

lemma to-spec:
  shows raw.to-spec ‘ behavior.stuttering.closed ⊆ raw.spec.closed
using downwards.closed.to-spec trace.stuttering.closed.to-spec by (blast intro: raw.spec.closed.I)

setup ‹Sign.parent-path›

setup ‹Sign.mandatory-path behavior.stuttering.closed›

lemma from-spec:
  shows raw.from-spec ‘ trace.stuttering.closed
     ⊆ (behavior.stuttering.closed :: ('a, 's, 'v) behavior.t set set)
proof −
  have ∗: behavior.take i ω₂ ∈ P
    if ω₁ ≃T ω₂ and ∀ i. behavior.take i ω₁ ∈ P and P ∈ trace.stuttering.closed
   for ω₁ ω₂ i and P :: ('a, 's, 'v) trace.t set
    using that(2−)
    by − (rule behavior.stuttering.equiv.takeE[OF sym[OF ‹ω₁ ≃T ω₂›], where i=i];
          fastforce intro: trace.stuttering.closed-in)
  show ?thesis
```

by (*fastforce simp*: *raw.from-spec-def elim*: *behavior.stuttering.clE* ∗)
qed

lemma *safety-cl*:
assumes $P \in$ *behavior.stuttering.closed*
shows *raw.safety.cl* $P \in$ *behavior.stuttering.closed*
unfolding *raw.safety.cl-def* **using** *assms*
by (*blast intro*: *subsetD*[*OF behavior.stuttering.closed.from-spec*]
subsetD[*OF trace.stuttering.closed.to-spec*])

setup ‹*Sign.parent-path*›

setup ‹*Sign.mandatory-path tls*›

lift-definition *to-spec* :: ('*a*, '*s*, '*v*) *tls* ⇒ ('*a*, '*s*, '*v*) *spec* **is** *raw.to-spec*
using *raw.spec.closed.to-spec* **by** *blast*

lift-definition *from-spec* :: ('*a*, '*s*, '*v*) *spec* ⇒ ('*a*, '*s*, '*v*) *tls* **is** *raw.from-spec*
by (*meson image-subset-iff behavior.stuttering.closed.from-spec raw.spec.closed.stuttering-closed*)

interpretation *safety*: *galois.complete-lattice-class tls.to-spec tls.from-spec*
by *standard* (*transfer*; *simp add*: *raw.safety.galois*)

setup ‹*Sign.mandatory-path from-spec*›

lemma *singleton*:
notes *spec.singleton.transfer*[*transfer-rule*]
shows *tls.from-spec* (*spec.singleton* σ)
$= \bigsqcup$(*tls.singleton* ‘ {ω . $\forall i$. *behavior.take* i $\omega \in$ *Safety-Logic.raw.singleton* σ})
by *transfer* (*simp add*: *behavior.stuttering.cl-bot raw.from-spec.singleton*)

lemmas *bot* = *raw.from-spec.empty*[*transferred*]

lemma *sup*:
shows *tls.from-spec* ($P \sqcup Q$) = *tls.from-spec* $P \sqcup$ *tls.from-spec* Q
by *transfer* (*rule raw.from-spec.sup*)

lemmas *Inf* = *tls.safety.upper-Inf*
lemmas *inf* = *tls.safety.upper-inf*

setup ‹*Sign.parent-path*›

setup ‹*Sign.mandatory-path to-spec*›

lemma *singleton*:
notes *spec.singleton.transfer*[*transfer-rule*]
shows *tls.to-spec* (*tls.singleton* ω) = ($\bigsqcup i$. *spec.singleton* (*behavior.take* i ω))
by *transfer* (*simp add*: *raw.to-spec.singleton*)

lemmas *bot* = *tls.safety.lower-bot*

lemmas *Sup* = *tls.safety.lower-Sup*
lemmas *sup* = *tls.safety.lower-sup*

setup ‹*Sign.parent-path*›

setup ‹*Sign.mandatory-path safety*›

setup ‹*Sign.mandatory-path cl*›

lemma *transfer*[*transfer-rule*]:
  shows *rel-fun* (*pcr-tls* (=) (=) (=)) (*pcr-tls* (=) (=) (=)) *raw.safety.cl tls.safety.cl*
unfolding *raw.safety.cl-def tls.safety.cl-def* **by** *transfer-prover*

lemma *bot*[*iff*]:
  shows *tls.safety.cl* $\bot = \bot$
by (*simp add*: *tls.safety.cl-def tls.from-spec.bot tls.safety.lower-bot*)

lemma *sup*:
  shows *tls.safety.cl* $(P \sqcup Q)$ = *tls.safety.cl* $P \sqcup$ *tls.safety.cl* Q
by (*simp add*: *tls.safety.cl-def tls.from-spec.sup tls.to-spec.sup*)

lemmas *state-prop* = *raw.safety.cl.state-prop*[*transferred*]
lemmas *always-le* = *raw.safety.cl.always-le*[*transferred*]

lemma *eventually*: — all the infinite traces and any finite ones that satisfy $\Diamond P$
  assumes $P \neq \bot$
  shows *tls.safety.cl* $(\Diamond P) = -\Diamond$*tls.terminated* $\sqcup \Diamond P$
using *assms* **by** *transfer* (*rule raw.safety.cl.eventually*)

lemma *terminated-iff*:
  assumes $\langle\!|\omega|\!\rangle_T \leq$ *tls.terminated*
  shows $\langle\!|\omega|\!\rangle_T \leq$ *tls.safety.cl* $P \longleftrightarrow \langle\!|\omega|\!\rangle_T \leq P$ (**is** *?lhs* $\longleftrightarrow$ *?rhs*)
using *assms*
by *transfer*
   (*simp add*: *raw.singleton-def behavior.stuttering.least-conv raw.safety.cl.terminated-iff*
          behavior.stuttering.closed.safety-cl behavior.stuttering.closed.raw.terminated)

lemma *terminated*:
  shows *tls.safety.cl tls.terminated* = *tls.idle* $\sqcup$ *tls.terminated*
by *transfer* (*simp add*: *raw.safety.cl.terminated*)

lemma *not-terminated*:
  shows *tls.safety.cl* $(-$ *tls.terminated*$)$ = $-$ *tls.terminated* (**is** *?lhs* = *?rhs*)
proof −
  have *?lhs* = *tls.safety.cl* $(\Diamond(-$ *tls.terminated*$))$
    by (*simp flip*: *tls.always.neg tls.terminated.eq-always-terminated*)
  also have ... = $-\Diamond$*tls.terminated* $\sqcup \Diamond(-$ *tls.terminated*$)$
    by (*metis tls.safety.cl.eventually tls.terminated.not-top*
          boolean-algebra.compl-zero boolean-algebra-class.boolean-algebra.double-compl)
  also have ... = *?rhs*
    by (*simp add*: *sup.absorb2 tls.eventually.expansive*
        flip: *tls.always.neg tls.terminated.eq-always-terminated*)
  finally show *?thesis* **.**
qed

lemma *le-terminated-conv*:
  shows *tls.safety.cl* $P \leq$ *tls.terminated* $\longleftrightarrow P = \bot$ (**is** *?lhs* $\longleftrightarrow$ *?rhs*)
proof(*rule iffI*)
  show *?lhs* $\Longrightarrow$ *?rhs*
    by *transfer* (*rule raw.safety.cl.le-terminated-bot*)
  show *?rhs* $\Longrightarrow$ *?lhs*
    by *simp*
qed

setup ‹*Sign.parent-path*›

setup ‹*Sign.mandatory-path closed*›

lemma *transfer*[*transfer-rule*]:
  shows *rel-set* (*pcr-tls* (=) (=) (=))
              (*behavior.stuttering.closed* ∩ *raw.safety.closed*)
              tls.safety.closed (**is** *rel-set - ?lhs ?rhs*)
proof(*rule rel-setI*)
  fix *X* **assume** *X* ∈ *?lhs* **then show** ∃ *Y*∈*?rhs*. *pcr-tls* (=) (=) (=) *X Y*
    by (*metis* (*no-types*, *opaque-lifting*) *raw.safety.cl-def raw.safety.closed-conv tls.safety.closed-upper*
          tls.from-spec.rep-eq TLS-inverse cr-tls-def tls.pcr-cr-eq tls.to-spec.rep-eq Int-iff)
next
  fix *Y* **assume** *Y* ∈ *?rhs* **then show** ∃ *X*∈*?lhs*. *pcr-tls* (=) (=) (=) *X Y*
    by (*metis tls.safety.cl-def tls.safety.closed-conv tls.from-spec.rep-eq*
          tls.pcr-cr-eq cr-tls-def unTLS raw.safety.closed-upper Int-iff)
qed

lemma *bot*:
  shows ⊥ ∈ *tls.safety.closed*
by (*simp add*: *tls.safety.closed-clI*)

lemma *sup*:
  assumes *P* ∈ *tls.safety.closed*
  assumes *Q* ∈ *tls.safety.closed*
  shows *P* ⊔ *Q* ∈ *tls.safety.closed*
by (*simp add*: *assms tls.safety.closed-clI tls.safety.cl.sup flip*: *tls.safety.closed-conv*)

lemmas *inf* = *tls.safety.closed-inf*

lemma *boolean-implication*:
  assumes −*P* ∈ *tls.safety.closed*
  assumes *Q* ∈ *tls.safety.closed*
  shows *P* ⟶$_B$ *Q* ∈ *tls.safety.closed*
by (*simp add*: *assms boolean-implication.conv-sup tls.safety.closed.sup*)

lemma *state-prop*:
  shows *tls.state-prop P* ∈ *tls.safety.closed*
by (*simp add*: *tls.safety.closed-clI tls.safety.cl.state-prop*)

lemma *not-terminated*:
  shows − *tls.terminated* ∈ *tls.safety.closed*
by (*simp add*: *tls.safety.closed-clI tls.safety.cl.not-terminated*)

lemma *unless*:
  assumes *P* ∈ *tls.safety.closed*
  assumes *Q* ∈ *tls.safety.closed*
  shows *tls.unless P Q* ∈ *tls.safety.closed*
using *assms* **by** *transfer* (*blast intro*: *raw.safety.closed.unless*)

lemma *always*:
  assumes *P* ∈ *tls.safety.closed*
  shows *tls.always P* ∈ *tls.safety.closed*
by (*simp add*: *assms tls.always-alt-def tls.safety.closed.bot tls.safety.closed.unless*)

setup ‹*Sign.parent-path*›

setup ‹*Sign.mandatory-path cl*›

lemma *until-unless-le*:
assumes $P \in$ *tls.safety.closed*
assumes $Q \in$ *tls.safety.closed*
shows *tls.safety.cl* (*tls.until P Q*) $\leq$ *tls.unless P Q*
by (*simp add*: *order.trans*[*OF tls.safety.cl-inf-le*] *tls.until.eq-unless-inf-eventually*
flip: *tls.safety.closed-conv*[*OF tls.safety.closed.unless*[*OF assms*]])

setup ‹*Sign.parent-path*›

setup ‹*Sign.parent-path*›

setup ‹*Sign.mandatory-path singleton*›

lemma *to-spec-le-conv*[*tls.singleton.le-conv*]:
notes *spec.singleton.transfer*[*transfer-rule*]
shows $\langle\!|\sigma|\!\rangle \leq$ *tls.to-spec P* $\longleftrightarrow$ $(\exists\, \omega\ i.\ \langle\!|\omega|\!\rangle_T \leq P \wedge \sigma =$ *behavior.take i* $\omega)$
by *transfer*
(*simp add*: *TLS.raw.singleton-def behavior.stuttering.least-conv Safety-Logic.raw.singleton-def*
raw.spec.least-conv[*OF subsetD*[*OF raw.spec.closed.to-spec*]];
fastforce simp: *raw.to-spec-def*)

lemma *from-spec-le-conv*[*tls.singleton.le-conv*]:
notes *spec.singleton.transfer*[*transfer-rule*]
shows $\langle\!|\omega|\!\rangle_T \leq$ *tls.from-spec P* $\longleftrightarrow$ $(\forall\, i.\ \langle\!|$*behavior.take i* $\omega|\!\rangle \leq P)$
by *transfer*
(*simp add*: *TLS.raw.singleton-def Safety-Logic.raw.singleton-def raw.spec.least-conv*
behavior.stuttering.least-conv
subsetD[*OF behavior.stuttering.closed.from-spec*
imageI[*OF raw.spec.closed.stuttering-closed*]];
simp add: *raw.from-spec-def*)

lemma *safety-cl-le-conv*[*tls.singleton.le-conv*]:
shows $\langle\!|\omega|\!\rangle_T \leq$ *tls.safety.cl P* $\longleftrightarrow$ $(\forall\, i.\ \exists \omega'.\ \langle\!|\omega'|\!\rangle_T \leq P \wedge$ *behavior.take i* $\omega =$ *behavior.take i* $\omega')$
by *transfer*
(*simp add*: *TLS.raw.singleton-def behavior.stuttering.least-conv behavior.stuttering.closed.safety-cl*;
fastforce intro: *raw.safety.cl-altI*
elim: *raw.safety.cl-altE*)

setup ‹*Sign.parent-path*›

setup ‹*Sign.parent-path*›

16.7 Maps

setup ‹*Sign.mandatory-path tls*›

definition *map* :: $('a \Rightarrow 'b) \Rightarrow ('s \Rightarrow 't) \Rightarrow ('v \Rightarrow 'w) \Rightarrow ('a, 's, 'v)$ *tls* $\Rightarrow ('b, 't, 'w)$ *tls* **where**
map af sf vf P = $\bigsqcup$(*tls.singleton* ‘ *behavior.map af sf vf* ‘ $\{\sigma.\ \langle\!|\sigma|\!\rangle_T \leq P\}$)

definition *invmap* :: $('a \Rightarrow 'b) \Rightarrow ('s \Rightarrow 't) \Rightarrow ('v \Rightarrow 'w) \Rightarrow ('b, 't, 'w)$ *tls* $\Rightarrow ('a, 's, 'v)$ *tls* **where**
invmap af sf vf P = $\bigsqcup$(*tls.singleton* ‘ *behavior.map af sf vf* $-$‘ $\{\sigma.\ \langle\!|\sigma|\!\rangle_T \leq P\}$)

abbreviation *amap* ::$('a \Rightarrow 'b) \Rightarrow ('a, 's, 'v)$ *tls* $\Rightarrow ('b, 's, 'v)$ *tls* **where**
amap af $\equiv$ *tls.map af id id*
abbreviation *ainvmap* ::$('a \Rightarrow 'b) \Rightarrow ('b, 's, 'v)$ *tls* $\Rightarrow ('a, 's, 'v)$ *tls* **where**
ainvmap af $\equiv$ *tls.invmap af id id*
abbreviation *smap* ::$('s \Rightarrow 't) \Rightarrow ('a, 's, 'v)$ *tls* $\Rightarrow ('a, 't, 'v)$ *tls* **where**
smap sf $\equiv$ *tls.map id sf id*

abbreviation *sinvmap* ::('*s* ⇒ '*t*) ⇒ ('*a*, '*t*, '*v*) *tls* ⇒ ('*a*, '*s*, '*v*) *tls* **where**
  sinvmap sf ≡ *tls.invmap id sf id*
abbreviation *vmap* ::('*v* ⇒ '*w*) ⇒ ('*a*, '*s*, '*v*) *tls* ⇒ ('*a*, '*s*, '*w*) *tls* **where** — aka *liftM*
  vmap vf ≡ *tls.map id id vf*
abbreviation *vinvmap* ::('*v* ⇒ '*w*) ⇒ ('*a*, '*s*, '*w*) *tls* ⇒ ('*a*, '*s*, '*v*) *tls* **where**
  vinvmap vf ≡ *tls.invmap id id vf*

interpretation *map-invmap*: *galois.complete-lattice-distributive-class*
  tls.map af sf vf
  tls.invmap af sf vf **for** *af sf vf*
proof *standard*
  show *tls.map af sf vf P* ≤ *Q* ⟷ *P* ≤ *tls.invmap af sf vf Q* (**is** *?lhs* ⟷ *?rhs*) **for** *P Q*
  proof(*rule iffI*)
    show *?lhs* ⟹ *?rhs*
      by (*fastforce simp*: *tls.map-def tls.invmap-def intro*: *tls.singleton-le-extI*)
    show *?rhs* ⟹ *?lhs*
      by (*fastforce simp*: *tls.map-def tls.invmap-def tls.singleton-le-conv*
                dest: *order.trans*[*of - P*] *behavior.stuttering.equiv.map*[**where** *af*=*af* **and** *sf*=*sf* **and** *vf*=*vf*]
                cong: *tls.singleton-cong*)
  qed
  show *tls.invmap af sf vf* (⨆ *X*) ≤ ⨆ (*tls.invmap af sf vf* ‘ *X*) **for** *X*
    by (*fastforce simp*: *tls.invmap-def*)
qed

setup ‹*Sign.mandatory-path singleton*›

lemma *map-le-conv*[*tls.singleton.le-conv*]:
  shows ⟪ω⟫$_T$ ≤ *tls.map af sf vf P* ⟷ (∃ω′. ⟪ω′⟫$_T$ ≤ *P* ∧ ⟪ω⟫$_T$ ≤ ⟪*behavior.map af sf vf* ω′⟫$_T$)
by (*simp add*: *tls.map-def*)

lemma *invmap-le-conv*[*tls.singleton.le-conv*]:
  shows ⟪ω⟫$_T$ ≤ *tls.invmap af sf vf P* ⟷ ⟪*behavior.map af sf vf* ω⟫$_T$ ≤ *P*
by (*simp add*: *tls.invmap-def tls.singleton-le-conv*)
   (*metis behavior.natural.map-natural tls.singleton-eq-conv*)

setup ‹*Sign.parent-path*›

setup ‹*Sign.mandatory-path map*›

lemmas *bot* = *tls.map-invmap.lower-bot*

lemmas *monotone* = *tls.map-invmap.monotone-lower*
lemmas *mono* = *monotoneD*[*OF tls.map.monotone*]

lemmas *Sup* = *tls.map-invmap.lower-Sup*
lemmas *sup* = *tls.map-invmap.lower-sup*

lemmas *Inf-le* = *tls.map-invmap.lower-Inf-le* — Converse does not hold
lemmas *inf-le* = *tls.map-invmap.lower-inf-le* — Converse does not hold

lemmas *invmap-le* = *tls.map-invmap.lower-upper-contractive*

lemma *singleton*:
  shows *tls.map af sf vf* ⟪ω⟫$_T$ = ⟪*behavior.map af sf vf* ω⟫$_T$
by (*auto simp*: *tls.map-def order.eq-iff tls.singleton-le-conv intro*: *behavior.stuttering.equiv.map*)

lemma *top*:
  assumes *surj af*

assumes *surj sf*
assumes *surj vf*
shows *tls.map af sf vf* $\top = \top$
by (*rule antisym*)
(*auto simp*: *assms tls.singleton.top tls.map.Sup tls.map.singleton surj-f-inv-f*
intro: *exI*[**where** *x*=*behavior.map* (*inv af*) (*inv sf*) (*inv vf*) σ **for** σ])

lemma *id*:
shows *tls.map id id id P* = *P*
and *tls.map* ($\lambda x.\ x$) ($\lambda x.\ x$) ($\lambda x.\ x$) *P* = *P*
by (*simp-all add*: *tls.map-def flip*: *id-def*)

lemma *comp*:
shows *tls.map af sf vf* ∘ *tls.map ag sg vg* = *tls.map* (*af* ∘ *ag*) (*sf* ∘ *sg*) (*vf* ∘ *vg*) (**is** *?lhs* = *?rhs*)
and *tls.map af sf vf* (*tls.map ag sg vg P*) = *tls.map* ($\lambda a.$ *af* (*ag a*)) ($\lambda s.$ *sf* (*sg s*)) ($\lambda v.$ *vf* (*vg v*)) *P* (**is** *?thesis1*)
proof −
have *?lhs P* = *?rhs P* **for** *P*
by (*rule tls.singleton.exhaust*[**where** *x*=*P*])
(*simp add*: *tls.map.Sup tls.map.singleton map-prod.comp image-image comp-def*)
then show *?lhs* = *?rhs* **and** *?thesis1* **by** (*simp-all add*: *comp-def*)
qed

lemmas *map* = *tls.map.comp*

setup ‹*Sign.parent-path*›

setup ‹*Sign.mandatory-path invmap*›

lemmas *bot* = *tls.map-invmap.upper-bot*
lemmas *top* = *tls.map-invmap.upper-top*

lemmas *monotone* = *tls.map-invmap.monotone-upper*
lemmas *mono* = *monotoneD*[*OF tls.invmap.monotone*]

lemmas *Sup* = *tls.map-invmap.upper-Sup*
lemmas *sup* = *tls.map-invmap.upper-sup*

lemmas *Inf* = *tls.map-invmap.upper-Inf*
lemmas *inf* = *tls.map-invmap.upper-inf*

lemma *singleton*:
shows *tls.invmap af sf vf* ⟪ω⟫$_T$ = $\bigsqcup$ (*tls.singleton* ‘ {ω'. ⟪*behavior.map af sf vf* ω'⟫$_T$ ≤ ⟪ω⟫$_T$})
by (*simp add*: *tls.invmap-def*)

lemma *id*:
shows *tls.invmap id id id P* = *P*
and *tls.invmap* ($\lambda x.\ x$) ($\lambda x.\ x$) ($\lambda x.\ x$) *P* = *P*
unfolding *id-def*[*symmetric*] **by** (*metis tls.map.id*(*1*) *tls.map-invmap.lower-upper-lower*(*2*))+

lemma *comp*:
shows *tls.invmap af sf vf* (*tls.invmap ag sg vg P*) = *tls.invmap* ($\lambda x.$ *ag* (*af x*)) ($\lambda s.$ *sg* (*sf s*)) ($\lambda v.$ *vg* (*vf v*)) *P*
(**is** *?lhs P* = *?rhs P*)
and *tls.invmap af sf vf* ∘ *tls.invmap ag sg vg* = *tls.invmap* (*ag* ∘ *af*) (*sg* ∘ *sf*) (*vg* ∘ *vf*) (**is** *?thesis1*)
proof −
show *?lhs P* = *?rhs P* **for** *P*
by (*auto intro*: *tls.singleton.antisym tls.singleton-le-extI simp*: *tls.singleton.le-conv*)
then show *?thesis1*
by (*simp add*: *fun-eq-iff comp-def*)

qed

lemmas *invmap* = *tls.invmap.comp*

setup ‹*Sign.parent-path*›

setup ‹*Sign.mandatory-path to-spec*›

lemma *map*:
  shows *tls.to-spec* (*tls.map af sf vf P*) = *spec.map af sf vf* (*tls.to-spec P*)
by (*rule tls.singleton.exhaust*[*of P*])
  (*simp add*: *tls.map.Sup tls.map.singleton spec.map.Sup spec.map.singleton image-image*
      tls.to-spec.singleton tls.to-spec.Sup behavior.take.map)

setup ‹*Sign.parent-path*›

setup ‹*Sign.parent-path*›

16.8 Abadi's axioms for TLA

The axioms for "propositional" TLA due to Abadi (1990) hold in this model. These are complete for *tls.always* and *tls.eventually*.
Observations:

- Abadi says that the temporal system is D aka S4.3Dum; see Goldblatt (1992, §8)
  - the only interesting axiom here is 5: the discrete-time Dummett axiom
- "propositional" means that actions are treated separately; we omit this part as we don't have actions ala TLA

setup ‹*Sign.mandatory-path tls.Abadi*›

lemma *Ax1*:
  shows $\models \Box(P \longrightarrow_B Q) \longrightarrow_B \Box P \longrightarrow_B \Box Q$
by (*simp add*: *tls.valid-def boolean-implication.shunt-top tls.always.always-imp-le*)

lemma *Ax2*:
  shows $\models \Box P \longrightarrow_B P$
by (*simp add*: *tls.valid-def boolean-implication.shunt-top tls.always.contractive*)

lemma *Ax3*:
  shows $\models \Box P \longrightarrow_B \Box\Box P$
by (*simp add*: *tls.validI*)

lemma *Ax4*:
  — "a classical way to express that time is linear – that any two instants in the future are ordered" Warford et al. (2020, (254) Lemmon formula)
  shows $\models \Box(\Box P \longrightarrow_B Q) \sqcup \Box(\Box Q \longrightarrow_B P)$
proof –
  have $\models (-\Box P)\ \mathcal{W}\ \Box Q \sqcup (-\Box Q)\ \mathcal{W}\ \Box P$ **by** (*rule tls.unless.ordering*)
  also have ... $\leq \Box((-\Box P)\ \mathcal{W}\ \Box Q) \sqcup \Box((-\Box Q)\ \mathcal{W}\ \Box P)$
    by (*metis sup-mono tls.always.idempotent tls.unless.alwaysR-le*)
  also have ... $\leq \Box(-\Box P \sqcup Q) \sqcup \Box(-\Box Q \sqcup P)$
    by (*strengthen ord-to-strengthen*(*1*)[*OF tls.unless.sup-le*])
      (*meson order.refl sup-mono tls.always.contractive tls.always.mono*)
  also have ... $= \Box(\Box P \longrightarrow_B Q) \sqcup \Box(\Box Q \longrightarrow_B P)$
    by (*simp add*: *boolean-implication.conv-sup*)

```
  finally show ?thesis .
qed

lemma Ax5:
  — "expresses the discreteness of time" See also Warford et al. (2020, §4.1 "the Dummett formula"): for them "next" encodes discreteness
  fixes P :: ('a, 's, 'v) tls
  shows ⊨ □(□(P ⟶B □P) ⟶B P) ⟶B ◇□P ⟶B P (is ⊨ ?goal)
proof −
  have raw-Ax5: raw.always (raw.eventually (P ∩ raw.eventually (−P)) ∪ P)
                  ∩ raw.eventually (raw.always P)
                ⊆ P (is ?lhs ⊆ ?rhs)
   for P :: ('a, 's, 'v) behavior.t set
proof(rule subsetI)
  fix ω assume ω ∈ ?lhs
  from IntD2[OF ‹ω ∈ ?lhs›]
  obtain i
    where ∃ω'. behavior.dropn i ω = Some ω' ∧ ω' ∈ raw.always P
    by (force simp: raw.always-alt-def raw.eventually-alt-def)
  then obtain i
    where i: ∃ω'. behavior.dropn i ω = Some ω' ∧ ω' ∈ raw.always P
      and ∀j<i. ∀ω'. behavior.dropn j ω = Some ω' ⟶ ω' ∉ raw.always P
    using ex-has-least-nat[where k=i and P=λi. ∃ω'. behavior.dropn i ω = Some ω' ∧ ω' ∈ raw.always P and m=id]
    by (auto dest: leD)
  have ∃ω'. behavior.dropn (i − j) ω = Some ω' ∧ ω' ∈ raw.always P for j
  proof(induct j)
    case (Suc j) show ?case
    proof(cases j < i)
      case True show ?thesis
      proof(rule ccontr)
        assume ∄ω'. behavior.dropn (i − Suc j) ω = Some ω' ∧ ω' ∈ raw.always P
        with ‹∃ω'. behavior.dropn i ω = Some ω' ∧ ω' ∈ raw.always P›
        have ∃ω'. behavior.dropn (i − Suc j) ω = Some ω' ∧ ω' ∉ raw.always P
          using behavior.dropn.shorterD[OF - diff-le-self] by blast
        then obtain k where ∃ω'. behavior.dropn (i − Suc j + k) ω = Some ω' ∧ ω' ∉ P
          by (clarsimp simp: raw.always-alt-def behavior.dropn.add behavior.dropn.Suc) blast
        with Suc.hyps ‹j < i›
        have ∃ω'. behavior.dropn (i − Suc j) ω = Some ω' ∧ ω' ∉ P
          by (fastforce simp: raw.always-alt-def behavior.dropn.add
                        split: nat-diff-split-asm
                         dest: spec[where x=k − 1])
        with ‹j < i› IntD1[OF ‹ω ∈ ?lhs›]
        obtain m n where ∃ω' ω'' ω'''. behavior.dropn (i − Suc j) ω = Some ω' ∧ ω' ∉ P
                                     ∧ behavior.dropn m ω' = Some ω''        ∧ ω'' ∈ P
                                     ∧ behavior.dropn n ω'' = Some ω'''      ∧ ω''' ∉ P
          by (simp add: raw.always-alt-def raw.eventually-alt-def)
             (blast dest: spec[where x=i − Suc j])
        with ‹j < i› Suc.hyps
        show False
          by (clarsimp simp: raw.always-alt-def dest!: spec[where x=m + n − 1] split: nat-diff-split-asm)
             (metis behavior.dropn.Suc behavior.dropn.bind-tl-commute behavior.dropn.dropn bind.bind-lunit)
      qed
    qed (use Suc.hyps in simp)
  qed (use i in simp)
  from this[of i] show ω ∈ P
    by (fastforce simp: raw.always-alt-def dest: spec[where x=0])
  qed
```

show *?thesis*
proof(*rule tls.validI*)
have $\Box(\Diamond(P \sqcap \Diamond(-P)) \sqcup P) \sqcap \Diamond\Box P \leq P$
by (*rule raw-Ax5*[*transferred*])
then have $\Box(\Diamond(P \sqcap \Diamond(-P)) \sqcup P) \sqcap \Diamond\Box P \leq P$
by (*simp add*: *boolean-implication.conv-sup tls.always.neg*)
then show $\top \leq$ *?goal*
by − (*intro iffD1*[*OF boolean-implication.shunt1*];
simp add: *boolean-implication.conv-sup tls.always.neg*)
qed
qed

lemma *Ax6*:
assumes $\models P$
shows $\models \Box P$
by (*rule tls.always.always-necessitation*[*OF assms*])

— Ax7: propositional tautologies: given by the *boolean-algebra* instance

lemma *Ax8*:
assumes $\models P$
assumes $\models P \longrightarrow_B Q$
shows $\models Q$
by (*rule tls.valid.rev-mp*[*OF assms*])

setup ‹*Sign.parent-path*›

16.9 Tweak syntax

unbundle *tls.no-syntax*
no-notation *tls.singleton* (‹$\langle$-$\rangle_T$›)

setup ‹*Sign.mandatory-path tls*›

bundle *extra-syntax*
begin
notation *tls.singleton* (‹$\langle$-$\rangle_T$› [*0*])
notation *tls.from-spec* (‹$(\!|$-$|\!)$› [*0*])
end

setup ‹*Sign.parent-path*›

17 Atomic sections

By restricting the environment to stuttering steps we can consider arbitrary processes to be atomic, i.e., free of interference.

setup ‹*Sign.mandatory-path spec*›

definition *atomic* :: *'a* ⇒ (*'a*, *'s*, *'v*) *spec* ⇒ (*'a*, *'s*, *'v*) *spec* **where**
atomic a P = *P* ⊓ *spec.rel* ({*a*} × *UNIV*)

setup ‹*Sign.mandatory-path idle*›

lemma *atomic-le-conv*[*spec.idle-le*]:
shows *spec.idle* ≤ *spec.atomic a P* ⟷ *spec.idle* ≤ *P*
by (*simp add*: *spec.atomic-def spec.idle.rel-le*)

setup ‹*Sign.parent-path*›

setup ‹*Sign.mandatory-path term*›

setup ‹*Sign.mandatory-path none*›

lemma *atomic*:
  shows *spec.term.none* (*spec.atomic a P*) = *spec.atomic a* (*spec.term.none P*)
by (*simp add*: *spec.atomic-def spec.term.none.inf spec.term.none.inf-rel*)

setup ‹*Sign.parent-path*›

setup ‹*Sign.mandatory-path all*›

lemma *atomic*:
  shows *spec.term.all* (*spec.atomic a P*) = *spec.atomic a* (*spec.term.all P*)
by (*simp add*: *spec.atomic-def spec.term.all.inf spec.term.all.rel*)

setup ‹*Sign.parent-path*›

setup ‹*Sign.parent-path*›

setup ‹*Sign.mandatory-path atomic*›

lemma *bot*[*simp*]:
  shows *spec.atomic a* ⊥ = ⊥
by (*simp add*: *spec.atomic-def*)

lemma *top*[*simp*]:
  shows *spec.atomic a* ⊤ = *spec.rel* ({*a*} × *UNIV*)
by (*simp add*: *spec.atomic-def*)

lemma *contractive*:
  shows *spec.atomic a P* ≤ *P*
by (*simp add*: *spec.atomic-def*)

lemma *idempotent*[*simp*]:
  shows *spec.atomic a* (*spec.atomic a P*) = *spec.atomic a P*
by (*simp add*: *spec.atomic-def*)

lemma *monotone*:
  shows *mono* (*spec.atomic a*)
by (*simp add*: *spec.atomic-def monoI le-infI1*)

lemmas *strengthen*[*strg*] = *st-monotone*[*OF spec.atomic.monotone*]
lemmas *mono* = *monotoneD*[*OF spec.atomic.monotone*]
lemmas *mono2mono*[*cont-intro*, *partial-function-mono*]
  = *monotone2monotone*[*OF spec.atomic.monotone*, *simplified*, *of orda P* **for** *orda P*]

lemma *Sup*:
  shows *spec.atomic a* (⨆ *X*) = ⨆ (*spec.atomic a* ‘ *X*)
by (*simp add*: *spec.atomic-def ac-simps heyting.inf-Sup-distrib*)

lemmas *sup* = *spec.atomic.Sup*[**where** *X*={*P*, *Q*} **for** *P Q*, *simplified*]

lemma *mcont2mcont*[*cont-intro*]:
  assumes *mcont luba orda Sup* (≤) *P*
  shows *mcont luba orda Sup* (≤) (λ*x*. *spec.atomic a* (*P x*))

by (*simp add*: *spec.atomic-def assms*)

lemma *Inf-not-empty*:
  assumes $X \neq \{\}$
  shows *spec.atomic a* $(\bigsqcap X) = \bigsqcap$ (*spec.atomic a* ‘ X)
using *assms* **by** (*simp add*: *spec.atomic-def INF-inf-const2*)

lemmas *inf* = *spec.atomic.Inf-not-empty*[**where** X=$\{P, Q\}$ **for** P Q, *simplified*]

lemma *idle*:
  shows *spec.atomic a spec.idle* = *spec.idle*
by (*simp add*: *spec.atomic-def inf.absorb1 spec.idle.rel-le*)

lemma *action*:
  shows *spec.atomic a* (*spec.action F*) = *spec.action* ($F \cap$ *UNIV* $\times$ ($\{a\}$ $\times$ *UNIV* $\cup$ *UNIV* $\times$ *Id*))
by (*simp add*: *spec.atomic-def spec.action.inf-rel-reflcl*)

lemma *return*:
  shows *spec.atomic a* (*spec.return v*) = *spec.return v*
by (*simp add*: *spec.return-def spec.atomic.action Times-Int-Times*)

lemma *bind*:
  shows *spec.atomic a* ($f \gg\!= g$) = *spec.atomic a f* $\gg\!=$ (λv. *spec.atomic a* (g v))
by (*simp add*: *spec.atomic-def spec.bind.inf-rel ac-simps*)

lemma *map-le*:
  fixes *af* :: $'a \Rightarrow 'b$
  shows *spec.map af sf vf* (*spec.atomic a P*) $\leq$ *spec.atomic* (*af a*) (*spec.map af sf vf P*)
by (*auto simp*: *spec.atomic-def spec.map.inf-rel*
      intro!: *spec.map.mono inf.mono order.refl spec.rel.mono*)

lemma *invmap*:
  fixes *af* :: $'a \Rightarrow 'b$
  shows *spec.atomic a* (*spec.invmap af sf vf P*) $\leq$ *spec.invmap af sf vf* (*spec.atomic* (*af a*) *P*)
by (*auto simp*: *spec.atomic-def spec.invmap.inf spec.invmap.rel*
      intro!: *le-infI2 spec.rel.mono*)

lemma *rel*:
  shows *spec.atomic a* (*spec.rel r*) = *spec.rel* ($r \cap \{a\} \times$ *UNIV*)
by (*simp add*: *spec.atomic-def flip*: *spec.rel.inf*)

lemma *interference*:
  shows *spec.atomic* (*proc a*) (*spec.rel* ($\{env\} \times$ *UNIV*)) = *spec.rel* $\{\}$
by (*simp add*: *spec.atomic.rel flip*: *Times-Int-distrib1*)

setup ‹*Sign.mandatory-path cam*›

lemma *cl*:
  shows *spec.atomic* (*proc a*) (*spec.cam.cl* ($\{env\} \times$ *UNIV*) *P*) = *spec.atomic* (*proc a*) *P*
by (*simp add*: *spec.cam.cl-def spec.atomic.sup spec.atomic.bind spec.atomic.interference*
        spec.rel.empty spec.term.none.bind spec.term.none.Sup spec.term.none.return
        image-image spec.bind.botR spec.bind.idleR sup-iff-le
      flip: *spec.term.none.atomic spec.term.all.atomic*)

setup ‹*Sign.parent-path*›

setup ‹*Sign.mandatory-path interference*›

lemma *cl*:
  shows *spec.atomic* (*proc a*) (*spec.interference.cl* ({*env*} × *UNIV*) *P*) = *spec.return* () ≫ *spec.atomic* (*proc a*) *P*
by (*simp add*: *spec.interference.cl-def UNIV-unit spec.atomic.bind spec.atomic.interference*
        spec.rel.empty spec.atomic.cam.cl spec.bind.return spec.atomic.return)

setup ‹*Sign.parent-path*›

setup ‹*Sign.parent-path*›

setup ‹*Sign.parent-path*›

setup ‹*Sign.mandatory-path prog*›

lift-definition *atomic* :: ('*s*, '*v*) *prog* ⇒ ('*s*, '*v*) *prog* **is**
  λ*P*. *spec.interference.cl* ({*env*} × *UNIV*) (*spec.atomic self P*) **..**

setup ‹*Sign.mandatory-path atomic*›

lemma *bot*[*simp*]:
  shows *prog.atomic* ⊥ = ⊥
by *transfer*
   (*simp add*: *spec.interference.cl.bot spec.atomic.interference spec.interference.cl.rel*
      flip: *spec.term.none.atomic spec.term.none.interference.cl*)

lemma *contractive*:
  shows *prog.atomic P* ≤ *P*
by *transfer* (*simp add*: *spec.atomic.contractive spec.interference.least*)

lemma *idempotent*[*simp*]:
  shows *prog.atomic* (*prog.atomic P*) = *prog.atomic P*
by *transfer* (*metis spec.atomic.idempotent spec.atomic.interference.cl spec.interference.closed-conv*)

lemma *monotone*:
  shows *mono prog.atomic*
by (*rule monoI*) (*transfer*; *simp add*: *spec.atomic.mono spec.interference.mono-cl*)

lemmas *strengthen*[*strg*] = *st-monotone*[*OF prog.atomic.monotone*]
lemmas *mono* = *monotoneD*[*OF prog.atomic.monotone*]
lemmas *mono2mono*[*cont-intro*, *partial-function-mono*] = *monotone2monotone*[*OF prog.atomic.monotone*, *simplified*, *of orda P* **for** *orda P*]

lemma *Sup*:
  shows *prog.atomic* (⨆ *X*) = ⨆ (*prog.atomic* ‘ *X*)
by *transfer*
   (*simp add*: *spec.atomic.Sup spec.atomic.sup spec.interference.cl-Sup spec.interference.cl-sup*
        image-image spec.interference.cl.bot spec.atomic.interference spec.interference.cl.rel
      flip: *spec.term.none.atomic spec.term.none.interference.cl*)

lemmas *sup* = *prog.atomic.Sup*[**where** *X*={*P*, *Q*} **for** *P Q*, *simplified*]

lemma *mcont*:
  shows *mcont Sup* (≤) *Sup* (≤) *prog.atomic*
by (*simp add*: *mcontI contI prog.atomic.Sup*)

lemmas *mcont2mcont*[*cont-intro*] = *mcont2mcont*[*OF prog.atomic.mcont*, *of luba orda P* **for** *luba orda P*]

lemma *Inf-le*:

shows *prog.atomic* ($\bigsqcap X$) $\leq \bigsqcap$ (*prog.atomic* ‘ X)
by *transfer* (*simp add*: *Inf-lower le-INF-iff spec.atomic.mono spec.interference.mono-cl*)

lemmas *inf-le* = *prog.atomic.Inf-le*[**where** X={P, Q} **for** P Q, *simplified*]

lemma *action*:
shows *prog.atomic* (*prog.action* F) = *prog.action* F
by *transfer*
(*simp add*: *spec.atomic.interference.cl spec.atomic.action spec.bind.returnL spec.idle.action-le*;
rule arg-cong; *blast*)

lemma *return*:
shows *prog.atomic* (*prog.return* v) = *prog.return* v
by (*simp add*: *prog.return-def prog.atomic.action*)

lemma *bind-le*:
shows *prog.atomic* ($f \gg\!= g$) $\leq$ *prog.atomic* $f \gg\!=$ (λv. *prog.atomic* (g v))
by *transfer*
(*simp add*: *spec.atomic.bind spec.bind.mono*
spec.interference.closed.bind spec.interference.expansive spec.interference.least)

setup ‹*Sign.parent-path*›

setup ‹*Sign.mandatory-path p2s*›

lemmas *atomic* = *prog.atomic.rep-eq*

setup ‹*Sign.parent-path*›

setup ‹*Sign.parent-path*›

17.1 Inhabitation

setup ‹*Sign.mandatory-path inhabits.spec*›

lemma *atomic*:
assumes P $-s$, $xs\rightarrow$ P'
assumes *trace.steps'* s xs $\subseteq$ {a} $\times$ *UNIV*
shows *spec.atomic* a P $-s$, $xs\rightarrow$ *spec.atomic* a P'
unfolding *spec.atomic-def* **by** (*rule inhabits.inf*[*OF assms*(*1*) *inhabits.spec.rel.rel*[*OF assms*(*2*)]])

lemma *atomic-term*:
assumes P $-s$, $xs\rightarrow$ *spec.return* v
assumes *trace.steps'* s xs $\subseteq$ {a} $\times$ *UNIV*
shows *spec.atomic* a P $-s$, $xs\rightarrow$ *spec.return* v
by (*rule inhabits.spec.atomic*[**where** P'=*spec.return* v, *simplified spec.atomic.return*, *OF assms*])

lemma *atomic-diverge*:
assumes P $-s$, $xs\rightarrow$ $\bot$
assumes *trace.steps'* s xs $\subseteq$ {a} $\times$ *UNIV*
shows *spec.atomic* a P $-s$, $xs\rightarrow$ $\bot$
by (*rule inhabits.spec.atomic*[**where** P'=$\bot$, *simplified spec.atomic.bot*, *OF assms*])

setup ‹*Sign.parent-path*›

setup ‹*Sign.mandatory-path inhabits.prog*›

lemma *atomic-term*:

assumes *prog.p2s P −s, xs→ spec.return v*
assumes *trace.steps′ s xs ⊆ {self} × UNIV*
shows *prog.p2s (prog.atomic P) −s, xs→ spec.return v*
unfolding *prog.p2s.atomic*
by (*rule inhabits.mono*[*OF spec.interference.expansive order.refl*
inhabits.spec.atomic-term[*OF assms*]])

lemma *atomic-diverge*:
assumes *prog.p2s P −s, xs→ ⊥*
assumes *trace.steps′ s xs ⊆ {self} × UNIV*
shows *prog.p2s (prog.atomic P) −s, xs→ ⊥*
unfolding *prog.p2s.atomic*
by (*rule inhabits.mono*[*OF spec.interference.expansive order.refl*
inhabits.spec.atomic-diverge[*OF assms*]])

setup ‹*Sign.parent-path*›

17.2 Assume/guarantee

setup ‹*Sign.mandatory-path ag.prog*›

lemma *atomic*:
assumes *prog.p2s c ≤ ⦃P⦄, Id ⊢ G, ⦃Q⦄*
assumes *P*: *stable A P*
assumes *Q*: *⋀v. stable A (Q v)*
shows *prog.p2s (prog.atomic c) ≤ ⦃P⦄, A ⊢ G, ⦃Q⦄*
apply (*subst ag.assm-heyting*[**where** *A=A* **and** *r=A*, *simplified*, *symmetric*])
apply (*simp add*: *prog.p2s.atomic*)
apply (*strengthen ord-to-strengthen*[*OF assms*(*1*)])
apply (*simp add*: *spec.atomic-def heyting ac-simps spec.interference.cl.inf-rel inf-sup-distrib Times-Int-Times*
flip: *spec.rel.inf*)
using *assms*
apply (*force intro*: *order.trans*[*OF - spec.interference.cl-ag-le*[**where** *A=A* **and** *r=A*, *simplified*]]
spec.interference.cl.mono[*OF order.refl*] *ag.pre-a*
simp add: *heyting*[*symmetric*] *ag.assm-heyting*[**where** *r={}*, *simplified*])
done

setup ‹*Sign.parent-path*›

18 Exceptions

A sketch of how we might handle exceptions in this framework.

setup ‹*Sign.mandatory-path raw*›

type-synonym (*'s*, *'x*, *'v*) *exn* = (*'s*, *'x* + *'v*) *prog*

definition *action* :: (*'v* × *'s* × *'s*) *set* ⇒ (*'s*, *'x*, *'v*) *raw.exn* **where**
action = *prog.action* ∘ *image* (*map-prod Inr id*)

definition *return* :: *'v* ⇒ (*'s*, *'x*, *'v*) *raw.exn* **where**
return = *prog.return* ∘ *Inr*

definition *throw* :: *'x* ⇒ (*'s*, *'x*, *'v*) *raw.exn* **where**
throw = *prog.return* ∘ *Inl*

definition *catch* :: (*'s*, *'x*, *'v*) *raw.exn* ⇒ (*'x* ⇒ (*'s*, *'x*, *'v*) *raw.exn*) ⇒ (*'s*, *'x*, *'v*) *raw.exn* **where**
catch f handler = *f* ⤜ *case-sum handler raw.return*

definition *bind* :: ('*s*, '*x*, '*v*) *raw.exn* ⇒ ('*v* ⇒ ('*s*, '*x*, '*v*) *raw.exn*) ⇒ ('*s*, '*x*, '*v*) *raw.exn* **where**
bind f g = *f* ⤜ *case-sum raw.throw g*

definition *parallel* :: ('*s*, '*x*, *unit*) *raw.exn* ⇒ ('*s*, '*x*, *unit*) *raw.exn* ⇒ ('*s*, '*x*, *unit*) *raw.exn* **where**
parallel P Q = (*P* ⤜ *case-sum* ⊥ *prog.return* ∥ *Q* ⤜ *case-sum* ⊥ *prog.return*) ⤜ *raw.return*

setup ‹*Sign.mandatory-path bind*›

lemma *bind*:
shows *raw.bind* (*raw.bind f g*) *h* = *raw.bind f* (λ*x*. *raw.bind* (*g x*) *h*)
by (*simp add*: *raw.bind-def prog.bind.bind sum.case-distrib*[**where** *h*=λ*f*. *f* ⤜ *case-sum raw.throw h*])
(*simp add*: *raw.throw-def comp-def prog.bind.return cong*: *sum.case-cong*)

lemma *return*:
shows *returnL*: *raw.bind* (*raw.return v*) = (λ*g*. *g v*)
and *returnR*: *raw.bind f raw.return* = *f*
by (*simp-all add*: *fun-eq-iff raw.bind-def raw.return-def raw.throw-def prog.bind.return case-sum-Inl-Inr-L*)

lemma *throwL*:
shows *raw.bind* (*raw.throw x*) = (λ*g*. *raw.throw x*)
by (*simp add*: *fun-eq-iff raw.bind-def raw.throw-def prog.bind.return*)

setup ‹*Sign.parent-path*›

setup ‹*Sign.mandatory-path catch*›

lemma *catch*:
shows *raw.catch* (*raw.catch f handler*$_1$) *handler*$_2$ = *raw.catch f* (λ*x*. *raw.catch* (*handler*$_1$ *x*) *handler*$_2$)
by (*simp add*: *raw.catch-def prog.bind.bind sum.case-distrib*[**where** *h*=λ*f*. *f* ⤜ *case-sum handler*$_2$ *raw.return*])
(*simp add*: *raw.return-def comp-def prog.bind.return cong*: *sum.case-cong*)

lemma *returnL*:
shows *raw.catch* (*raw.return v*) = (λ*handler*. *raw.return v*)
by (*simp add*: *fun-eq-iff raw.catch-def raw.return-def prog.bind.return*)

lemma *throw*:
shows *throwL*: *raw.catch* (*raw.throw x*) = (λ*g*. *g x*)
and *throwR*: *raw.catch f raw.throw* = *f*
by (*simp-all add*: *fun-eq-iff raw.catch-def raw.return-def raw.throw-def prog.bind.return case-sum-Inl-Inr-L*)

setup ‹*Sign.parent-path*›

setup ‹*Sign.mandatory-path parallel*›

lemma *commute*:
shows *raw.parallel P Q* = *raw.parallel Q P*
by (*simp add*: *raw.parallel-def prog.parallel.commute*)

lemma *assoc*:
shows *raw.parallel P* (*raw.parallel Q R*) = *raw.parallel* (*raw.parallel P Q*) *R*
by (*simp add*: *raw.parallel-def raw.return-def prog.bind.bind prog.bind.return prog.parallel.assoc*)

lemma *return*:
shows *raw.parallel* (*raw.return* ()) *P* = *raw.catch P* (λ*x*. ⊥) (**is** *?thesis1*)
and *raw.parallel P* (*raw.return* ()) = *raw.catch P* (λ*x*. ⊥) (**is** *?thesis2*)
proof −
show *?thesis1*

by (*simp add*: *raw.parallel-def raw.return-def*
prog.bind.bind prog.bind.return prog.parallel.return prog.bind.botL
sum.case-distrib[**where** *h*=λ*f*. *f* ⤜ *prog.return* ∘ *Inr*]
flip: *raw.catch-def*[*unfolded raw.return-def o-def*]
cong: *sum.case-cong*)
then show *?thesis2*
by (*simp add*: *raw.parallel.commute*)
qed

lemma *throw*:
shows *raw.parallel* (*raw.throw x*) *P* = *raw.bind* (*raw.catch P* (λ*x*. ⊥)) (λ*x*. ⊥) (**is** *?thesis1*)
and *raw.parallel P* (*raw.throw x*) = *raw.bind* (*raw.catch P* (λ*x*. ⊥)) (λ*x*. ⊥) (**is** *?thesis2*)
proof −
show *?thesis1*
by (*simp add*: *raw.parallel-def raw.throw-def raw.bind-def raw.return-def raw.catch-def*
prog.bind.bind prog.bind.return prog.bind.botL prog.parallel.bot
sum.case-distrib[**where** *h*=λ*f*. *prog.bind f* (λ*x*. ⊥)]
sum.case-distrib[**where** *h*=λ*f*. *f* ⤜ *case-sum* (*prog.return* ∘ *Inl*) (λ*x*. ⊥)]
cong: *sum.case-cong*)
then show *?thesis2*
by (*simp add*: *raw.parallel.commute*)
qed

setup ‹*Sign.parent-path*›

setup ‹*Sign.parent-path*›

typedef ('*s*, '*x*, '*v*) *exn* = *UNIV* :: ('*s*, '*x*, '*v*) *raw.exn set*
by *blast*

setup-lifting *type-definition-exn*

instantiation *exn* :: (*type*, *type*, *type*) *complete-distrib-lattice*
begin

lift-definition *bot-exn* :: ('*s*, '*x*, '*v*) *exn* **is** ⊥ .
lift-definition *top-exn* :: ('*s*, '*x*, '*v*) *exn* **is** ⊤ .
lift-definition *sup-exn* :: ('*s*, '*x*, '*v*) *exn* ⇒ ('*s*, '*x*, '*v*) *exn* ⇒ ('*s*, '*x*, '*v*) *exn* **is** *sup* .
lift-definition *inf-exn* :: ('*s*, '*x*, '*v*) *exn* ⇒ ('*s*, '*x*, '*v*) *exn* ⇒ ('*s*, '*x*, '*v*) *exn* **is** *inf* .
lift-definition *less-eq-exn* :: ('*s*, '*x*, '*v*) *exn* ⇒ ('*s*, '*x*, '*v*) *exn* ⇒ *bool* **is** *less-eq* .
lift-definition *less-exn* :: ('*s*, '*x*, '*v*) *exn* ⇒ ('*s*, '*x*, '*v*) *exn* ⇒ *bool* **is** *less* .
lift-definition *Inf-exn* :: ('*s*, '*x*, '*v*) *exn set* ⇒ ('*s*, '*x*, '*v*) *exn* **is** *Inf* .
lift-definition *Sup-exn* :: ('*s*, '*x*, '*v*) *exn set* ⇒ ('*s*, '*x*, '*v*) *exn* **is** *Sup* .

instance by *standard* (*transfer*; *auto intro*: *Inf-lower InfI le-supI1 SupI SupE Inf-Sup*)+

end

setup ‹*Sign.mandatory-path exn*›

lift-definition *action* :: ('*v* × '*s* × '*s*) *set* ⇒ ('*s*, '*x*, '*v*) *exn* **is** *raw.action* .
lift-definition *return* :: '*v* ⇒ ('*s*, '*x*, '*v*) *exn* **is** *raw.return* .
lift-definition *throw* :: '*x* ⇒ ('*s*, '*x*, '*v*) *exn* **is** *raw.throw* .
lift-definition *catch* :: ('*s*, '*x*, '*v*) *exn* ⇒ ('*x* ⇒ ('*s*, '*x*, '*v*) *exn*) ⇒ ('*s*, '*x*, '*v*) *exn* **is** *raw.catch* .
lift-definition *bind* :: ('*s*, '*x*, '*v*) *exn* ⇒ ('*v* ⇒ ('*s*, '*x*, '*v*) *exn*) ⇒ ('*s*, '*x*, '*v*) *exn* **is** *raw.bind* .
lift-definition *parallel* :: ('*s*, '*x*, *unit*) *exn* ⇒ ('*s*, '*x*, *unit*) *exn* ⇒ ('*s*, '*x*, *unit*) *exn* **is** *raw.parallel* .

adhoc-overloading

Monad-Syntax.bind ⇌ *exn.bind*
adhoc-overloading
parallel ⇌ *exn.parallel*

setup ‹*Sign.mandatory-path bind*›

lemma *bind*:
shows $f \gg\!= g \gg\!= h = exn.bind\ f\ (\lambda x.\ g\ x \gg\!= h)$
by (*transfer* (*rule raw.bind.bind*))

lemma *return*:
shows *returnL*: $(\gg\!=)\ (exn.return\ v) = (\lambda g.\ g\ v)$ (**is** *?thesis1*)
and *returnR*: $f \gg\!= exn.return = f$ (**is** *?thesis2*)
by (*transfer*; *rule raw.bind.return*)+

lemma *throwL*:
shows $(\gg\!=)\ (exn.throw\ x) = (\lambda g.\ exn.throw\ x)$
by (*transfer* (*rule raw.bind.throwL*))

setup ‹*Sign.parent-path*›

setup ‹*Sign.mandatory-path catch*›

lemma *catch*:
shows $exn.catch\ (exn.catch\ f\ handler_1)\ handler_2 = exn.catch\ f\ (\lambda x.\ exn.catch\ (handler_1\ x)\ handler_2)$
by (*transfer* (*rule raw.catch.catch*))

lemma *returnL*:
shows $exn.catch\ (exn.return\ v) = (\lambda handler.\ exn.return\ v)$
by (*transfer* (*rule raw.catch.returnL*))

lemma *throw*:
shows *throwL*: $exn.catch\ (exn.throw\ x) = (\lambda g.\ g\ x)$
and *throwR*: $exn.catch\ f\ exn.throw = f$
by (*transfer*; *rule raw.catch.throw*)+

setup ‹*Sign.parent-path*›

setup ‹*Sign.mandatory-path parallel*›

lemma *commute*:
shows $exn.parallel\ P\ Q = exn.parallel\ Q\ P$
by (*transfer* (*rule raw.parallel.commute*))

lemma *assoc*:
shows $exn.parallel\ P\ (exn.parallel\ Q\ R) = exn.parallel\ (exn.parallel\ P\ Q)\ R$
by (*transfer* (*rule raw.parallel.assoc*))

lemma *return*:
shows *returnL*: $exn.return\ ()\ \|\ P = exn.catch\ P\ \bot$
and *returnR*: $P\ \|\ exn.return\ () = exn.catch\ P\ \bot$
unfolding *bot-fun-def* **by** (*transfer*; *rule raw.parallel.return*)+

lemma *throw*:
shows *throwL*: $exn.throw\ x\ \|\ P = exn.catch\ P\ \bot \gg\!= \bot$
and *throwR*: $P\ \|\ exn.throw\ x = exn.catch\ P\ \bot \gg\!= \bot$
unfolding *bot-fun-def* **by** (*transfer*; *rule raw.parallel.throw*)+

setup ‹*Sign.parent-path*›

setup ‹*Sign.parent-path*›

19 Assume/Guarantee rule sets

The rules in *ConcurrentHOL.Refinement* are deficient in various ways:

- redundant stability requirements
- interleaving of program decomposition with stability goals
- insufficiently instantiated

The following are some experimental rules aimed at practical assume/guarantee reasoning.

19.1 Implicit stabilisation

We can define a relation *ceilr P* to be the largest (weakest assumption) for which *P* is stable. This always yields a preorder (i.e., it is reflexive and transitive). Later we use this to inline stability side conditions into assume/guarantee rules (§19.1.1).

This relation is not very pleasant to work with: it is not monotonic and does not have many useful algebraic properties. However it suffices to defer the checking of assumes (see §19.1.1).

This is a cognate of the *strongest guarantee* used by de Roever et al. (2001, Definition 8.31) in their completeness proof for the rely-guarantee method.

definition *ceilr* :: *'a pred* ⇒ *'a rel* **where**
  ceilr P = ⨆{*r*. *stable r P*}

lemma *ceilr-alt-def*:
  shows *ceilr P* = {(*s*, *s'*). *P s* ⟶ *P s'*}
by (*auto simp*: *ceilr-def stable-def monotone-def le-bool-def order-eq-iff Sup-upper*)

lemma *ceilrE*[*elim*]:
  assumes (*x*, *y*) ∈ *ceilr P*
  assumes *P x*
  shows *P y*
using *assms* **by** (*simp add*: *ceilr-alt-def*)

setup ‹*Sign.mandatory-path ceilr*›

named-theorems *simps* ‹*simp rules for* ***const***‹*ceilr*››

lemma *bot*[*ceilr.simps*]:
  shows *ceilr* ⊥ = *UNIV*
by (*simp-all add*: *ceilr-alt-def*)

lemma *top*[*ceilr.simps*]:
  shows *ceilr* ⊤ = *UNIV*
by (*simp-all add*: *ceilr-alt-def*)

lemma *const*[*ceilr.simps*]:
  shows *ceilr* ⟨*c*⟩ = *UNIV*
    and *ceilr* (*P* ∧ ⟨*c*⟩) = (*if c then ceilr P else UNIV*)
    and *ceilr* (⟨*c*⟩ ∧ *P*) = (*if c then ceilr P else UNIV*)
    and *ceilr* (*P* ∧ ⟨*c*⟩ ∧ *P'*) = (*if c then ceilr* (*P* ∧ *P'*) *else UNIV*)
by (*simp-all add*: *ceilr-alt-def*)

lemma *Id-le*:
  shows $Id \subseteq ceilr\ P$
by (*auto simp*: *ceilr-alt-def*)

lemmas *refl*[*iff*] = *ceilr.Id-le*[*folded refl-alt-def*]

lemma *trans*[*iff*]:
  shows *trans* (*ceilr P*)
by (*simp add*: *ceilr-alt-def trans-def*)

lemma *stable*[*stable.intro*]:
  shows *stable* (*ceilr P*) *P*
by (*simp add*: *ceilr-def stable.Union-conv*)

lemma *largest*[*stable.intro*]:
  assumes *stable r P*
  shows $r \subseteq ceilr\ P$
by (*simp add*: *ceilr-def assms Sup-upper*)

lemma *disj-subseteq*: — Converse does not hold
  shows $ceilr\ (P \vee Q) \subseteq ceilr\ P \cup ceilr\ Q$
by (*fastforce simp*: *ceilr-alt-def*)

lemma *Ex-subseteq*: — Converse does not hold
  shows $ceilr\ (\exists\, x.\ P\ x) \subseteq (\bigcup x.\ ceilr\ (P\ x))$
by (*fastforce simp*: *ceilr-alt-def*)

lemma *conj-subseteq*: — Converse does not hold
  shows $ceilr\ P \cap ceilr\ Q \subseteq ceilr\ (P \wedge Q)$
by (*fastforce simp*: *ceilr-alt-def*)

lemma *All-subseteq*: — Converse does not hold
  shows $(\bigcap x.\ ceilr\ (P\ x)) \subseteq ceilr\ (\forall\, x.\ P\ x)$
by (*fastforce simp*: *ceilr-alt-def*)

lemma *const-implies*[*ceilr.simps*]:
  shows $ceilr\ (\langle P \rangle \longrightarrow Q) = (\mathit{if}\ P\ \mathit{then}\ ceilr\ Q\ \mathit{else}\ UNIV)$
by (*simp add*: *ceilr.simps*)

lemma *Id-proj-on*:
  shows $(\bigcap c.\ ceilr\ (\langle c \rangle = f)) = Id_f$
    and $(\bigcap c.\ ceilr\ (f = \langle c \rangle)) = Id_f$
by (*fastforce simp*: *ceilr-alt-def*)+

setup ‹*Sign.parent-path*›

setup ‹*Sign.mandatory-path stable*›

lemma *Inter-ceilr*:
  shows *stable* $(\bigcap v.\ ceilr\ (Q\ v))\ (Q\ v)$
by (*rule antimonoD*[**where** *y*=$ceilr\ (Q\ v)$, *OF stable.antimono-rel*, *unfolded le-bool-def*, *rule-format*, *rotated*])
   (*auto simp*: *ceilr.stable*)

setup ‹*Sign.parent-path*›

We can internalize the stability conditions; see §19.1.1 for further discussion.

setup ‹*Sign.mandatory-path ag.prog*›

lemma *p2s-s2p-ag-ceilr*:
shows *prog.p2s* (*prog.s2p* ({|*P*|}, *ceilr P* ∩ (⋂ *v*. *ceilr* (*Q v*)) ⊢ *G*, {|*Q*|}))
= {|*P*|}, *ceilr P* ∩ (⋂ *v*. *ceilr* (*Q v*)) ⊢ *G*, {|*Q*|}
by (*simp add*: *ag.prog.p2s-s2p-ag-stable ceilr.stable stable.Inter-ceilr stable.infI*)

setup ‹*Sign.parent-path*›

19.1.1 Assume/guarantee rules using implicit stability

We use *ceilr* to incorporate stability side conditions directly into the assume/guarantee rules. In other words, instead of working with arbitrary relations, we work with the largest (most general) *assume* that makes the relevant predicates *stable*.

In practice this allows us to defer all stability obligations to the end of a proof, which may be in any convenient context (typically a function). This approach could be considered a semantic version of how Zakowski, Cachera, Demange, Petri, Pichardie, Jagannathan, and Vitek (2019) split sequential and assume/guarantee reasoning. See Vafeiadis (2008, §4) for a discussion on when to check stability.

We defer the *guarantee* proofs by incorporating them into preconditions. This also allows control flow context to be accumulated.

These are backchaining ("weakest precondition") rules: the guarantee and post condition need to be instantiated and the rules instantiate assume and pre condition schematics.

Note that the rule for (≫=) duplicates stability goals.

See §22 for an example of using these rules.

setup ‹*Sign.mandatory-path iag*›

named-theorems *intro* ‹*safe backchaining intro rules*›

lemma *init*:
assumes *c* ≤ {|*P*|}, *A* ⊢ *G*, {|*Q*|}
assumes ⋀*s*. *P' s* ⟹ *P s*
assumes *A'* ⊆ *A* — these rules use *ceilr* which always yields a reflexive relation (*ceilr.refl*
shows *c* ≤ {|*P'*|}, *A'* ⊢ *G*, {|*Q*|}
using *assms*(*2*,*3*) **by** (*auto intro*: *ag.mono order.trans*[*OF assms*(*1*)])

lemmas *mono* = *ag.mono*

lemmas *gen-asm* = *ag.gen-asm*

lemmas *pre* = *ag.pre*
lemmas *pre-pre* = *ag.pre-pre*
lemmas *pre-post* = *ag.pre-post*
lemmas *pre-ag* = *ag.pre-ag*
lemmas *pre-a* = *ag.pre-a*
lemmas *pre-g* = *ag.pre-g*

lemmas *post-imp* = *ag.post-imp*

lemmas *conj-lift* = *ag.conj-lift*
lemmas *disj-lift* = *ag.disj-lift*
lemmas *all-lift* = *ag.all-lift*

lemmas *augment-a* = *ag.augment-a*
lemmas *augment-post* = *ag.augment-post*
lemmas *augment-post-imp* = *ag.augment-post-imp*

lemmas *stable-augment-base* = *ag.stable-augment-base*
lemmas *stable-augment* = *ag.stable-augment*
lemmas *stable-augment-post* = *ag.stable-augment-post*

lemmas *stable-augment-frame = ag.stable-augment-frame*

lemma *bind*[*iag.intro*]:
assumes $\bigwedge v.$ *prog.p2s* $(g\ v) \leq \{\!|Q'\ v|\!\}$, $A_2\ v \vdash G$, $\{\!|Q|\!\}$
assumes *prog.p2s* $f \leq \{\!|P|\!\}$, $A_1 \vdash G$, $\{\!|Q'|\!\}$
shows *prog.p2s* $(f \gg\!= g) \leq \{\!|P|\!\}$, $A_1 \cap (\bigcap v.\ A_2\ v) \vdash G$, $\{\!|Q|\!\}$
by (*auto simp*: *prog.p2s.simps intro*: *assms ag.spec.bind ag.pre*)

lemmas *rev-bind = iag.bind*[*rotated*]

lemma *read*[*iag.intro*]:
shows *prog.p2s* $(prog.read\ F) \leq \{\!|\lambda s.\ Q\ (F\ s)\ s|\!\}$, $ceilr\ (\lambda s.\ Q\ (F\ s)\ s) \cap (\bigcap s.\ ceilr\ (Q\ (F\ s))) \vdash G$, $\{\!|Q|\!\}$
by (*rule ag.pre*[*OF ag.prog.action*[**where** $P{=}\lambda s.\ Q\ (F\ s)\ s$
and $Q{=}Q$
and $A{=}ceilr\ (\lambda s.\ Q\ (F\ s)\ s) \cap (\bigcap s.\ ceilr\ (Q\ (F\ s)))$
and $G{=}G$]];
fastforce intro: *stable.intro stable.Inter-ceilr stable.infI*)

lemma *return*[*iag.intro*]:
shows *prog.p2s* $(prog.return\ v) \leq \{\!|Q\ v|\!\}$, $ceilr\ (Q\ v) \vdash G$, $\{\!|Q|\!\}$
unfolding *prog.return-alt-def* **by** (*rule iag.init*[*OF iag.read*]; *fastforce*)

lemma *write*[*iag.intro*]: — this is where *guarantee* obligations arise
shows *prog.p2s* $(prog.write\ F)$
$\leq \{\!|(\lambda s.\ Q\ ()\ (F\ s) \wedge (s, F\ s) \in G)|\!\}$, $ceilr\ (\lambda s.\ Q\ ()\ (F\ s) \wedge (s, F\ s) \in G) \cap ceilr\ (Q\ ()) \vdash G$, $\{\!|Q|\!\}$
by (*rule ag.prog.action*; *fastforce intro*: *stable.intro stable.Inter-ceilr stable.infI1 stable.infI2*)

lemma *parallel*: — not in the *iag* format; instantiate the first two assumptions
assumes *prog.p2s* $c_1 \leq \{\!|P_1|\!\}$, $A_1 \vdash G_1$, $\{\!|Q_1|\!\}$
assumes *prog.p2s* $c_2 \leq \{\!|P_2|\!\}$, $A_2 \vdash G_2$, $\{\!|Q_2|\!\}$
assumes $\bigwedge s.\ [\![Q_1\ ()\ s;\ Q_2\ ()\ s]\!] \Longrightarrow Q\ ()\ s$
assumes $G_2 \subseteq A_1$
assumes $G_1 \subseteq A_2$
assumes $G_1 \cup G_2 \subseteq G$
shows *prog.p2s* $(prog.parallel\ c_1\ c_2) \leq \{\!|P_1 \wedge P_2|\!\}$, $A_1 \cap A_2 \vdash G$, $\{\!|Q|\!\}$
by (*rule order.trans*[*OF ag.prog.parallel*[*OF iag.pre-a*[*OF assms(1)*] *iag.pre-a*[*OF assms(2)*],
where $A{=}A_1 \cap A_2$, *simplified*, *OF* ‹$G_2 \subseteq A_1$› ‹$G_1 \subseteq A_2$›]];
use assms(3,6) **in** ‹*force intro*: *iag.mono*›)

lemmas *local = ag.prog.local* — not in the *iag* format

lemma *if*[*iag.intro*]:
assumes $b \Longrightarrow$ *prog.p2s* $c_1 \leq \{\!|P_1|\!\}$, $A_1 \vdash G$, $\{\!|Q|\!\}$
assumes $\neg b \Longrightarrow$ *prog.p2s* $c_2 \leq \{\!|P_2|\!\}$, $A_2 \vdash G$, $\{\!|Q|\!\}$
shows *prog.p2s* $(if\ b\ then\ c_1\ else\ c_2) \leq \{\!|if\ b\ then\ P_1\ else\ P_2|\!\}$, $A_1 \cap A_2 \vdash G$, $\{\!|Q|\!\}$
using *assms* **by** (*fastforce intro*: *ag.pre-ag*)

lemma *case-option*[*iag.intro*]:
assumes $x = None \Longrightarrow$ *prog.p2s none* $\leq \{\!|P_n|\!\}$, $A_n \vdash G$, $\{\!|Q|\!\}$
assumes $\bigwedge v.\ x = Some\ v \Longrightarrow$ *prog.p2s* $(some\ v) \leq \{\!|P_s\ v|\!\}$, $A_s\ v \vdash G$, $\{\!|Q|\!\}$
shows *prog.p2s* $(case\text{-}option\ none\ some\ x) \leq \{\!|case\ x\ of\ None \Rightarrow P_n \mid Some\ v \Rightarrow P_s\ v|\!\}$, $case\text{-}option\ A_n\ A_s\ x$
$\vdash G$, $\{\!|Q|\!\}$
using *assms* **by** (*fastforce intro*: *ag.pre-ag split*: *option.split*)

lemma *case-sum*[*iag.intro*]:
assumes $\bigwedge v.\ x = Inl\ v \Longrightarrow$ *prog.p2s* $(left\ v) \leq \{\!|P_l\ v|\!\}$, $A_l\ v \vdash G$, $\{\!|Q|\!\}$
assumes $\bigwedge v.\ x = Inr\ v \Longrightarrow$ *prog.p2s* $(right\ v) \leq \{\!|P_r\ v|\!\}$, $A_r\ v \vdash G$, $\{\!|Q|\!\}$
shows *prog.p2s* $(case\text{-}sum\ left\ right\ x) \leq \{\!|case\text{-}sum\ P_l\ P_r\ x|\!\}$, $case\text{-}sum\ A_l\ A_r\ x \vdash G$, $\{\!|Q|\!\}$

using *assms* **by** (*fastforce intro*: *ag.pre-ag split*: *sum.split*)

lemma *case-list*[*iag.intro*]:
assumes $x = [] \Longrightarrow$ *prog.p2s nil* $\leq \{\!|P_n|\!\}$, $A_n \vdash G$, $\{\!|Q|\!\}$
assumes $\bigwedge v\ vs.\ x = v \# vs \Longrightarrow$ *prog.p2s* (*cons v vs*) $\leq \{\!|P_c\ v\ vs|\!\}$, $A_c\ v\ vs \vdash G$, $\{\!|Q|\!\}$
shows *prog.p2s* (*case-list nil cons x*) $\leq \{\!|$*case-list* $P_n\ P_c\ x|\!\}$, *case-list* $A_n\ A_c\ x \vdash G$, $\{\!|Q|\!\}$
using *assms* **by** (*fastforce intro*: *ag.pre-ag split*: *list.split*)

lemma *while*:
fixes $c :: 'k \Rightarrow ('s, 'k + 'v)$ *prog*
assumes *c*: $\bigwedge k.$ *prog.p2s* $(c\ k) \leq \{\!|P\ k|\!\}$, $A \vdash G$, $\{\!|$*case-sum I Q*$|\!\}$
shows *prog.p2s* (*prog.while c k*) $\leq \{\!|\langle \forall\ v\ s.\ I\ v\ s \longrightarrow P\ v\ s\rangle \wedge I\ k|\!\}$, $A \cap (\bigcap v.$ *ceilr* $(Q\ v)) \vdash G$, $\{\!|Q|\!\}$
by (*rule iag.gen-asm*)
(*rule ag.prog.while*[*OF ag.pre-a*[*OF c*]]; *blast intro*: *stable.Inter-ceilr stable.infI2*)

lemmas *whenM* = *iag.if*[**where** c_1=*c* **and** A_1=*A* **and** P_1=*P*, *OF - iag.return*[**where** *v*=()]] **for** *A c P*

setup ‹*Sign.parent-path*›

19.2 Refinement with relational assumes

Two sets of refinement rules:

- relational assumes
- relational assumes and *prog.sinvmap* (inverse state abstraction)

setup ‹*Sign.mandatory-path rar.prog*›

lemma *bind*:
assumes $\bigwedge v.$ *prog.p2s* $(g\ v) \leq \{\!|Q'\ v|\!\}$, *ag.assm A* $\Vdash$ *prog.p2s* $(g'\ v)$, $\{\!|Q|\!\}$
assumes *prog.p2s f* $\leq \{\!|P|\!\}$, *ag.assm A* $\Vdash$ *prog.p2s f'*, $\{\!|Q'|\!\}$
shows *prog.p2s* $(f \gg= g) \leq \{\!|P|\!\}$, *ag.assm A* $\Vdash$ *prog.p2s* $(f' \gg= g')$, $\{\!|Q|\!\}$
by (*rule refinement.prog.bind*[*OF refinement.pre-a*[*OF assms*(*1*) *refinement.spec.bind.res.rel-le*[*OF order.refl*]]])
(*simp add*: *spec.term.all.rel assms*(*2*))

lemmas *rev-bind* = *rar.prog.bind*[*rotated*]

lemma *action*:
fixes $F :: ('v \times 's \times 's)$ *set*
fixes $F' :: ('v \times 's \times 's)$ *set*
assumes *Q*: $\bigwedge v\ s\ s'.\ [\![P\ s;\ (v, s, s') \in F]\!] \Longrightarrow Q\ v\ s'$
assumes *F'*: $\bigwedge v\ s\ s'.\ [\![P\ s;\ (v, s, s') \in F]\!] \Longrightarrow (v, s, s') \in F'$
assumes *sP*: *stable A P*
assumes *sQ*: $\bigwedge v\ s\ s'.\ [\![P\ s;\ (v, s, s') \in F]\!] \Longrightarrow$ *stable A* $(Q\ v)$
shows *prog.p2s* (*prog.action F*) $\leq \{\!|P|\!\}$, *ag.assm A* $\Vdash$ *prog.p2s* (*prog.action F'*), $\{\!|Q|\!\}$
by (*rule refinement.prog.action*)
(*use assms* **in** ‹*simp-all add*: *spec.steps.rel stable-def monotone-def*›)

lemma *return*:
assumes *sQ*: *stable A* $(Q\ v)$
shows *prog.p2s* (*prog.return v*) $\leq \{\!|Q\ v|\!\}$, *ag.assm A* $\Vdash$ *prog.p2s* (*prog.return v*), $\{\!|Q|\!\}$
using *assms* **by** (*simp add*: *refinement.prog.return spec.steps.rel stable-def monotone-def*)

lemma *parallel-refinement*:
assumes c_1: *prog.p2s* $c_1 \leq \{\!|P_1|\!\}$, *ag.assm* $(A \cup G_2) \Vdash$ *prog.p2s* $(c_1' \sqcap$ *prog.rel* $G_1)$, $\{\!|Q_1|\!\}$
assumes c_2: *prog.p2s* $c_2 \leq \{\!|P_2|\!\}$, *ag.assm* $(A \cup G_1) \Vdash$ *prog.p2s* $(c_2' \sqcap$ *prog.rel* $G_2)$, $\{\!|Q_2|\!\}$
shows *prog.p2s* $(c_1 \parallel c_2) \leq \{\!|P_1 \wedge P_2|\!\}$, *ag.assm A* $\Vdash$ *prog.p2s* $(c_1' \sqcap$ *prog.rel* $G_1 \parallel c_2' \sqcap$ *prog.rel* $G_2)$, $\{\!|\lambda v.\ Q_1\ v \wedge Q_2\ v|\!\}$

apply (*rule refinement.prog.parallel*[*OF refinement.pre-a*[*OF* c_1] *refinement.pre-a*[*OF* c_2]])
apply (*rule order.trans*[*OF refinement.spec.env-hyp.mono*[*OF order.refl*] *ag.spec.Parallel-assm*[**where** *a*=*True* **and** *as*=*UNIV* **and** G=λa. *if a then* G_1 *else* G_2, *simplified*]];
simp add: *ac-simps prog.p2s.simps*; *fail*)
apply (*rule order.trans*[*OF refinement.spec.env-hyp.mono*[*OF order.refl*] *ag.spec.Parallel-assm*[**where** *a*=*False* **and** *as*=*UNIV* **and** G=λa. *if a then* G_1 *else* G_2, *simplified*]];
simp add: *ac-simps prog.p2s.simps*; *fail*)
done

lemma *parallel*:
assumes *prog.p2s* $c_1 \leq \{\!|P_1|\!\}$, *ag.assm* $(A \cup G_2) \Vdash$ *prog.p2s* c_1', $\{\!|Q_1|\!\}$
assumes *prog.p2s* $c_1 \leq \{\!|P_1|\!\}$, $A \cup G_2 \vdash G_1$, $\{\!|\top|\!\}$
assumes *prog.p2s* $c_2 \leq \{\!|P_2|\!\}$, *ag.assm* $(A \cup G_1) \Vdash$ *prog.p2s* c_2', $\{\!|Q_2|\!\}$
assumes *prog.p2s* $c_2 \leq \{\!|P_2|\!\}$, $A \cup G_1 \vdash G_2$, $\{\!|\top|\!\}$
shows *prog.p2s* $(c_1 \parallel c_2) \leq \{\!|P_1 \wedge P_2|\!\}$, *ag.assm* $A \Vdash$ *prog.p2s* $(c_1' \parallel c_2')$, $\{\!|\lambda v.\ Q_1\ v \wedge Q_2\ v|\!\}$
by (*rule order.trans*[*OF rar.prog.parallel-refinement*
refinement.mono[*OF order.refl order.refl prog.p2s.mono*[*OF prog.parallel.mono*[*OF inf.cobounded1 inf.cobounded1*]] *order.refl*]])
(*force simp*: *prog.p2s.simps refinement.infR-conv*[**where** Q_2=$\top$, *simplified*]
simp flip: *ag.refinement*
intro: *assms*)+

lemma *while*:
fixes $c :: 'k \Rightarrow ('s, 'k + 'v)$ *prog*
fixes $c' :: 'k \Rightarrow ('s, 'k + 'v)$ *prog*
assumes *c*: $\bigwedge k$. *prog.p2s* $(c\ k) \leq \{\!|P\ k|\!\}$, *ag.assm* $A \Vdash$ *prog.p2s* $(c'\ k)$, $\{\!|$*case-sum I Q*$|\!\}$
assumes *IP*: $\bigwedge s\ v.\ I\ v\ s \Longrightarrow P\ v\ s$
assumes *sQ*: $\bigwedge v$. *stable* A $(Q\ v)$
shows *prog.p2s* $(prog.while\ c\ k) \leq \{\!|I\ k|\!\}$, *ag.assm* $A \Vdash$ *prog.p2s* $(prog.while\ c'\ k)$, $\{\!|Q|\!\}$
proof −
have $\forall k$. *prog.p2s* $(prog.while\ c\ k) \leq \{\!|P\ k|\!\}$, *ag.assm* $A \Vdash$ *prog.p2s* $(prog.while\ c'\ k)$, $\{\!|Q|\!\}$
proof(*induct rule*: *prog.while.fixp-induct*[**where** P=λR. $\forall k$. *prog.p2s* $(R\ c\ k) \leq \{\!|P\ k|\!\}$, *ag.assm* $A \Vdash$ *prog.p2s* $(prog.while\ c'\ k)$, $\{\!|Q|\!\}$, *case-names adm bot step*])
case (*step R*) **from** *sQ* **show** *?case*
apply (*subst prog.while.simps*)
apply (*intro allI rar.prog.rev-bind*[*OF c*]
refinement.pre-pre[*OF refinement.prog.case-sum*[*OF step*[*rule-format*]
rar.prog.return[*OF sQ*]]])
apply (*simp add*: *IP split*: *sum.splits*)
done
qed *simp-all*
then show *?thesis*
by (*meson IP refinement.pre-pre*)
qed

lemma *app*:
fixes *xs* :: *'a list*
fixes *f* :: *'a* ⇒ (*'s*, *unit*) *prog*
fixes *P* :: *'a list* ⇒ *'s pred*
assumes $\bigwedge x\ ys\ zs.\ xs = ys\ @\ x\ \#\ zs \Longrightarrow$ *prog.p2s* $(f\ x) \leq \{\!|P\ ys|\!\}$, *ag.assm* $A \Vdash$ *prog.p2s* $(f'\ x)$, $\{\!|\lambda\text{-}.\ P\ (ys\ @\ [x])|\!\}$
assumes $\bigwedge ys$. *prefix ys xs* $\Longrightarrow$ *stable* A $(P\ ys)$
shows *prog.p2s* $(prog.app\ f\ xs) \leq \{\!|P\ []|\!\}$, *ag.assm* $A \Vdash$ *prog.p2s* $(prog.app\ f'\ xs)$, $\{\!|\lambda\text{-}.\ P\ xs|\!\}$
using *assms*
by (*induct xs rule*: *rev-induct*;
force intro: *rar.prog.return*
simp: *prog.app.append prog.app.simps spec.steps.rel prog.bind.return rar.prog.rev-bind*)

lemmas *if* = *refinement.prog.if*[**where** *A*=*ag.assm A* **for** *A*]
lemmas *case-option* = *refinement.prog.case-option*[**where** *A*=*ag.assm A* **for** *A*]

setup ‹*Sign.parent-path*›

setup ‹*Sign.mandatory-path rair.prog*›

abbreviation (*input*) *absfn sf c* ≡ *prog.p2s* (*prog.sinvmap sf c*)

lemma *bind*:
  assumes ⋀*v*. *prog.p2s* (*g v*) ≤ ⦃*Q*′ *v*⦄, *ag.assm A* ⊨ *rair.prog.absfn sf* (*g*′ *v*), ⦃*Q*⦄
  assumes *prog.p2s f* ≤ ⦃*P*⦄, *ag.assm A* ⊨ *rair.prog.absfn sf f*′, ⦃*Q*′⦄
  shows *prog.p2s* (*f* ≫= *g*) ≤ ⦃*P*⦄, *ag.assm A* ⊨ *rair.prog.absfn sf* (*f*′ ≫= *g*′), ⦃*Q*⦄
by (*simp add*: *prog.invmap.bind rar.prog.bind*[*OF assms*])

lemmas *rev-bind* = *rair.prog.bind*[*rotated*]

lemma *action*:
  fixes *F* :: (′*v* × ′*s* × ′*s*) *set*
  fixes *F*′ :: (′*v* × ′*t* × ′*t*) *set*
  fixes *sf* :: ′*s* ⇒ ′*t*
  assumes *Q*: ⋀*v s s*′. ⟦*P s*; (*v*, *s*, *s*′) ∈ *F*⟧ ⟹ *Q v s*′
  assumes *F*′: ⋀*v s s*′. ⟦*P s*; (*v*, *s*, *s*′) ∈ *F*⟧ ⟹ (*v*, *sf s*, *sf s*′) ∈ *F*′
  assumes *sP*: *stable A P*
  assumes *sQ*: ⋀*v s s*′. ⟦*P s*; (*v*, *s*, *s*′) ∈ *F*⟧ ⟹ *stable A* (*Q v*)
  shows *prog.p2s* (*prog.action F*) ≤ ⦃*P*⦄, *ag.assm A* ⊨ *rair.prog.absfn sf* (*prog.action F*′), ⦃*Q*⦄
by (*strengthen ord-to-strengthen*(*2*)[*OF prog.action.invmap-le*])
  (*simp add*: *rar.prog.action assms*)

lemma *return*:
  assumes *sQ*: *stable A* (*Q v*)
  shows *prog.p2s* (*prog.return v*) ≤ ⦃*Q v*⦄, *ag.assm A* ⊨ *rair.prog.absfn sf* (*prog.return v*), ⦃*Q*⦄
using *assms* **by** (*simp add*: *refinement.prog.invmap-return spec.steps.rel stable-def monotone-def*)

lemma *parallel*:
  fixes *sf* :: ′*s* ⇒ ′*t*
  assumes *prog.p2s c*₁ ≤ ⦃*P*₁⦄, *ag.assm* (*A* ∪ *G*₂) ⊨ *rair.prog.absfn sf c*₁′, ⦃*Q*₁⦄
  assumes *prog.p2s c*₁ ≤ ⦃*P*₁⦄, *A* ∪ *G*₂ ⊢ *G*₁, ⦃⊤⦄
  assumes *prog.p2s c*₂ ≤ ⦃*P*₂⦄, *ag.assm* (*A* ∪ *G*₁) ⊨ *rair.prog.absfn sf c*₂′, ⦃*Q*₂⦄
  assumes *prog.p2s c*₂ ≤ ⦃*P*₂⦄, *A* ∪ *G*₁ ⊢ *G*₂, ⦃⊤⦄
  shows *prog.p2s* (*c*₁ ‖ *c*₂) ≤ ⦃*P*₁ ∧ *P*₂⦄, *ag.assm A* ⊨ *rair.prog.absfn sf* (*c*₁′ ‖ *c*₂′), ⦃λ*v*. *Q*₁ *v* ∧ *Q*₂ *v*⦄
unfolding *prog.invmap.parallel* **by** (*rule rar.prog.parallel*[*OF assms*])

lemma *while*:
  fixes *c* :: ′*k* ⇒ (′*s*, ′*k* + ′*v*) *prog*
  fixes *c*′ :: ′*k* ⇒ (′*t*, ′*k* + ′*v*) *prog*
  fixes *sf* :: ′*s* ⇒ ′*t*
  assumes *c*: ⋀*k*. *prog.p2s* (*c k*) ≤ ⦃*P k*⦄, *ag.assm A* ⊨ *rair.prog.absfn sf* (*c*′ *k*), ⦃*case-sum I Q*⦄
  assumes *IP*: ⋀*s v*. *I v s* ⟹ *P v s*
  assumes *sQ*: ⋀*v*. *stable A* (*Q v*)
  shows *prog.p2s* (*prog.while c k*) ≤ ⦃*I k*⦄, *ag.assm A* ⊨ *rair.prog.absfn sf* (*prog.while c*′ *k*), ⦃*Q*⦄
by (*strengthen ord-to-strengthen*(*2*)[*OF prog.while.invmap-le*])
  (*simp add*: *assms map-sum.id rar.prog.while*[*OF c*])

lemma *app*:
  fixes *xs* :: ′*a list*
  fixes *f* :: ′*a* ⇒ (′*s*, *unit*) *prog*
  fixes *P* :: ′*a list* ⇒ ′*s pred*

assumes $\bigwedge x\ ys\ zs.\ xs = ys\ @\ x\ \#\ zs \Longrightarrow prog.p2s\ (f\ x) \leq \{\!|P\ ys|\!\}, ag.assm\ A \Vdash rair.prog.absfn\ sf\ (f'\ x), \{\!|\lambda\text{-}.\ P\ (ys\ @\ [x])|\!\}$
assumes $\bigwedge ys.\ prefix\ ys\ xs \Longrightarrow stable\ A\ (P\ ys)$
shows $prog.p2s\ (prog.app\ f\ xs) \leq \{\!|P\ []|\!\}, ag.assm\ A \Vdash rair.prog.absfn\ sf\ (prog.app\ f'\ xs), \{\!|\lambda\text{-}.\ P\ xs|\!\}$
by (*strengthen ord-to-strengthen*(2)[*OF prog.sinvmap.app-le*])
(*simp add*: *rar.prog.app assms*)

lemma *if*:
assumes $i \Longrightarrow prog.p2s\ t \leq \{\!|P|\!\}, ag.assm\ A \Vdash rair.prog.absfn\ sf\ t', \{\!|Q|\!\}$
assumes $\neg i \Longrightarrow prog.p2s\ e \leq \{\!|P'|\!\}, ag.assm\ A \Vdash rair.prog.absfn\ sf\ e', \{\!|Q|\!\}$
shows $prog.p2s\ (if\ i\ then\ t\ else\ e) \leq \{\!|if\ i\ then\ P\ else\ P'|\!\}, ag.assm\ A \Vdash rair.prog.absfn\ sf\ (if\ i\ then\ t'\ else\ e'), \{\!|Q|\!\}$
using *assms* **by** *fastforce*

lemma *case-option*:
assumes $opt = None \Longrightarrow prog.p2s\ none \leq \{\!|P_n|\!\}, ag.assm\ A \Vdash rair.prog.absfn\ sf\ none', \{\!|Q|\!\}$
assumes $\bigwedge v.\ opt = Some\ v \Longrightarrow prog.p2s\ (some\ v) \leq \{\!|P_s\ v|\!\}, ag.assm\ A \Vdash rair.prog.absfn\ sf\ (some'\ v), \{\!|Q|\!\}$
shows $prog.p2s\ (case\text{-}option\ none\ some\ opt) \leq \{\!|case\ opt\ of\ None \Rightarrow P_n \mid Some\ v \Rightarrow P_s\ v|\!\}, ag.assm\ A \Vdash rair.prog.absfn\ sf\ (case\text{-}option\ none'\ some'\ opt), \{\!|Q|\!\}$
using *assms* **by** (*simp add*: *option.case-eq-if*)

setup ‹*Sign.parent-path*›

setup ‹*Sign.parent-path*›

20 Wickerson, Dodds and Parkinson: explicit stabilisation

Notes on Wickerson, Dodds, and Parkinson (2010) (all references here are to the technical report):

- motivation: techniques for eliding redundant stability conditions
  - the standard rules check the interstitial assertion in $c\ ;\ d$ twice
- they claim in §7 to supersede the "mid stability" of Vafeiadis (2008, §4.1) (wssa, sswa)
- Appendix D:
  - not a complete set of rules
  - AtomR-S does not self-compose: consider $c\ ;\ d$ – the interstitial assertion is either a floor or ceiling
    - every step therefore requires a use of weakening/monotonicity

The basis of their approach is to make assertions a function of a relation (a *rely*). By considering a set of relations, a single rely-guarantee specification can satisfy several call sites. Separately they tweak the RGSep rules of Vafeiadis (2008).
The definitions are formally motivated as follows (§3):

> Our operators can also be defined using Dijkstra's predicate transformer semantics: $\lfloor p \rfloor R$ is the weakest precondition of R^* given postcondition p, while $\lceil p \rceil R$ is the strongest postcondition of R^* given precondition p.

The following adapts their definitions and proofs to our setting.

setup ‹*Sign.mandatory-path wdp*›

definition *floor* :: *'a rel* ⇒ *'a pred* ⇒ *'a pred* **where** — An interior operator, or a closure in the dual lattice
$floor\ r\ P\ s \longleftrightarrow (\forall s'.\ (s, s') \in r^* \longrightarrow P\ s')$

definition *ceiling* :: *'a rel* ⇒ *'a pred* ⇒ *'a pred* **where** — A closure operator

ceiling r P s $\longleftrightarrow$ $(\exists\, s'.\ (s',\ s) \in r^* \wedge P\ s')$

setup ‹*Sign.mandatory-path floor*›

lemma *empty-rel*[*simp*]:
shows *wdp.floor* {} *P* = *P*
by (*simp add*: *wdp.floor-def fun-eq-iff*)

lemma *reflcl*:
shows *wdp.floor* ($r^=$) = *wdp.floor r*
by (*simp add*: *wdp.floor-def fun-eq-iff*)

lemma *const*:
shows *wdp.floor r* ⟨*c*⟩ = ⟨*c*⟩
by (*auto simp*: *wdp.floor-def fun-eq-iff*)

lemma *contractive*:
shows *wdp.floor r P* ≤ *P*
by (*simp add*: *wdp.floor-def le-fun-def le-bool-def*)

lemma *idempotent*:
shows *wdp.floor r* (*wdp.floor r P*) = *wdp.floor r P*
by (*auto simp*: *fun-eq-iff wdp.floor-def dest*: *rtrancl-trans*)

lemma *mono*:
assumes *r'* ⊆ *r*
assumes *P* ≤ *P'*
shows *wdp.floor r P* ≤ *wdp.floor r' P'*
using *assms* **by** (*auto 6 0 simp add*: *wdp.floor-def le-bool-def le-fun-def dest*: *rtrancl-mono*)

lemma *strengthen*[*strg*]:
assumes *st-ord* (¬*F*) *r r'*
assumes *st-ord F P P'*
shows *st-ord F* (*wdp.floor r P*) (*wdp.floor r' P'*)
using *assms* **by** (*cases F*; *simp add*: *wdp.floor.mono*)

lemma *weakest*:
assumes *Q* ≤ *P*
assumes *stable r Q*
shows *Q* ≤ *wdp.floor r P*
using *assms* **by** (*simp add*: *wdp.floor-def stable-def monotone-def le-fun-def le-bool-def*) (*metis rtrancl-induct*)

lemma *Chernoff*:
assumes *P* ≤ *Q*
shows (*wdp.floor r P* ∧ *Q*) ≤ *wdp.floor r Q*
using *assms* **by** (*simp add*: *wdp.floor-def le-fun-def le-bool-def*)

lemma *floor1*:
assumes *r* ⊆ *r'*
shows *wdp.floor r'* (*wdp.floor r P*) = *wdp.floor r' P*
unfolding *wdp.floor-def* **by** (*meson assms rtrancl-cl.cl-mono rtrancl-eq-or-trancl rtrancl-trans*)

lemma *floor2*:
assumes *r* ⊆ *r'*
shows *wdp.floor r* (*wdp.floor r' P*) = *wdp.floor r' P*
by (*metis assms antisym wdp.floor.contractive wdp.floor.idempotent wdp.floor.mono*)

setup ‹*Sign.parent-path*›

interpretation *ceiling*: *closure-complete-lattice-distributive-class wdp.ceiling r* **for** *r*
by *standard* (*auto 5 0 simp*: *wdp.ceiling-def le-fun-def le-bool-def dest*: *rtrancl-trans*)

setup ‹*Sign.mandatory-path ceiling*›

lemma *empty-rel*[*simp*]:
  shows *wdp.ceiling* {} *P* = *P*
by (*simp add*: *wdp.ceiling-def fun-eq-iff*)

lemma *reflcl*:
  shows *wdp.ceiling* ($r^=$) = *wdp.ceiling r*
by (*simp add*: *wdp.ceiling-def fun-eq-iff*)

lemma *const*:
  shows *wdp.ceiling r* ⟨*c*⟩ = ⟨*c*⟩
by (*auto simp*: *wdp.ceiling-def fun-eq-iff*)

lemma *mono*:
  assumes *r* ⊆ *r'*
  assumes *P* ≤ *P'*
  shows *wdp.ceiling r P* ≤ *wdp.ceiling r' P'*
using *assms* **by** (*auto 7 0 simp*: *wdp.ceiling-def le-bool-def le-fun-def dest*: *rtrancl-mono*)

lemma *strengthen*[*strg*]:
  assumes *st-ord F r r'*
  assumes *st-ord F P P'*
  shows *st-ord F* (*wdp.ceiling r P*) (*wdp.ceiling r' P'*)
using *assms* **by** (*cases F*; *simp add*: *wdp.ceiling.mono*)

lemma *strongest*:
  assumes *P* ≤ *Q*
  assumes *stable r Q*
  shows *wdp.ceiling r P* ≤ *Q*
using *assms* **by** (*simp add*: *wdp.ceiling-def stable-def monotone-def le-fun-def le-bool-def*) (*metis rtrancl-induct*)

lemma *ceiling1*:
  assumes *r* ⊆ *r'*
  shows *wdp.ceiling r'* (*wdp.ceiling r P*) = *wdp.ceiling r' P*
unfolding *wdp.ceiling-def* **by** (*meson assms rtrancl-cl.cl-mono rtrancl-eq-or-trancl rtrancl-trans*)

lemma *ceiling2*:
  assumes *r* ⊆ *r'*
  shows *wdp.ceiling r* (*wdp.ceiling r' P*) = *wdp.ceiling r' P*
by (*metis assms antisym wdp.ceiling.ceiling1 wdp.ceiling.expansive wdp.ceiling.idempotent*(*1*))

setup ‹*Sign.parent-path*›

setup ‹*Sign.mandatory-path stable*›

lemma *floor*:
  shows *stable r* (*wdp.floor r P*)
unfolding *wdp.floor-def stable-def monotone-def* **by** (*simp add*: *converse-rtrancl-into-rtrancl le-boolI*)

lemma *ceiling*:
  shows *stable r* (*wdp.ceiling r P*)
by (*fastforce simp*: *wdp.ceiling-def stable-def monotone-def le-bool-def*
        dest: *rtrancl-into-rtrancl*)

lemma *floor-conv*:
assumes *stable r P*
shows *P* = *wdp.floor r P*
using *assms* **unfolding** *wdp.floor-def stable-def monotone-def le-bool-def fun-eq-iff*
by (*metis rtrancl-refl rtrancl-induct*)

lemma *ceiling-conv*:
assumes *stable r P*
shows *P* = *wdp.ceiling r P*
using *assms* **unfolding** *wdp.ceiling-def stable-def monotone-def le-bool-def fun-eq-iff*
by (*metis rtrancl-refl rtrancl-induct*)

setup ‹*Sign.parent-path*›

lemma *floor-alt-def*: — Wickerson et al. (2010, §3)
shows *wdp.floor r P* = ⨆{*Q*. *Q* ≤ *P* ∧ *stable r Q*}
by (*rule antisym*)
(*auto simp*: *Sup-upper wdp.floor.contractive wdp.stable.floor*
intro: *wdp.floor.weakest*[*unfolded le-bool-def le-fun-def*, *rule-format*])

lemma *ceiling-alt-def*: — Wickerson et al. (2010, §3)
shows *wdp.ceiling r P* = ⨅{*Q*. *P* ≤ *Q* ∧ *stable r Q*}
by (*rule antisym*)
(*auto simp*: *Inf-lower wdp.ceiling.expansive wdp.stable.ceiling*
dest: *wdp.ceiling.strongest*[*unfolded le-bool-def le-fun-def*, *rule-format*, *rotated*])

lemma *duality-floor-ceiling*:
shows *wdp.ceiling r* (¬*P*) = (¬*wdp.floor* (*r*$^{-1}$) *P*)
by (*simp add*: *wdp.ceiling-def wdp.floor-def fun-eq-iff rtrancl-converse*)

lemma *ceiling-floor*:
assumes *r* ⊆ *r*′
shows *wdp.ceiling r* (*wdp.floor r*′ *P*) = *wdp.floor r*′ *P*
by (*metis assms wdp.ceiling.ceiling2 wdp.stable.ceiling-conv wdp.stable.floor*)

lemma *floor-ceiling*:
assumes *r* ⊆ *r*′
shows *wdp.floor r* (*wdp.ceiling r*′ *P*) = *wdp.ceiling r*′ *P*
by (*metis assms wdp.floor.floor2 wdp.stable.ceiling wdp.stable.floor-conv*)

lemma *floor-ceilr*:
shows *wdp.floor* (*ceilr P*) *P* = *P*
by (*metis ceilr.stable wdp.stable.floor-conv*)

lemma *ceiling-ceilr*:
shows *wdp.ceiling* (*ceilr P*) *P* = *P*
by (*metis ceilr.stable wdp.stable.ceiling-conv*)

setup ‹*Sign.parent-path*›

20.1 Assume/Guarantee rules

§3.2 traditional assume/guarantee rules **setup** ‹*Sign.mandatory-path wdp*›

lemma *action*: — arbitrary *A*
fixes *F* :: (′*v* × ′*s* × ′*s*) *set*
assumes *Q*: ⋀*v s s*′. ⟦*P s*; (*v*, *s*, *s*′) ∈ *F*⟧ ⟹ *Q v s*′

assumes G: $\bigwedge v\ s\ s'.\ [\![P\ s;\ s \neq s';\ (v,\ s,\ s') \in F]\!] \Longrightarrow (s,\ s') \in G$
shows *prog.p2s* (*prog.action* F) $\leq$ $\{\!|wdp.floor\ A\ P|\!\}$, $A \vdash G$, $\{\!|\lambda v.\ wdp.ceiling\ A\ (Q\ v)|\!\}$
by (*rule ag.prog.action*)
(*auto simp*: *wdp.stable.floor wdp.stable.ceiling*
intro: *G*
dest: *Q*[*rotated*]
elim: *wdp.floor.contractive*[*unfolded le-fun-def le-bool-def*, *rule-format*]
wdp.ceiling.expansive[*unfolded le-fun-def le-bool-def*, *rule-format*])

lemmas *mono* = *ag.mono*
lemmas *bind* = *ag.prog.bind*

etc. – the other rules are stock

setup ‹*Sign.parent-path*›

§4, Appendix C parametric specifications **definition** *pag* :: ('*s rel* ⇒ '*s pred*) ⇒ '*s rel set* ⇒ '*s rel* ⇒ ('*s rel* ⇒ '*v* ⇒ '*s pred*) ⇒ (*sequential*, '*s*, '*v*) *spec* (‹$\{\!|$-$|\!\}$, -/ $\vdash_P$ -, $\{\!|$-$|\!\}$› [0,0,0,0] 100) **where**
$\{\!|P|\!\}$, $As \vdash_P G$, $\{\!|Q|\!\}$ = $(\bigsqcap A{\in}As.\ \{\!|P\ A|\!\},\ A \vdash G,\ \{\!|Q\ A|\!\})$

setup ‹*Sign.mandatory-path pag*›

lemma *empty*:
shows $\{\!|P|\!\}$, $\{\} \vdash_P G$, $\{\!|Q|\!\} = \top$
by (*simp add*: *pag-def*)

lemma *singleton*:
shows $\{\!|P|\!\}$, $\{A\} \vdash_P G$, $\{\!|Q|\!\} = \{\!|P\ A|\!\}$, $A \vdash G$, $\{\!|Q\ A|\!\}$
by (*simp add*: *pag-def*)

lemma *mono*: — strengthening of the WEAKEN rule in Figure 4, needed for the example
assumes $\bigwedge A.\ A \in As' \Longrightarrow P'\ A \leq P\ A$
assumes $As' \leq As$
assumes $G \leq G'$
assumes $\bigwedge A.\ A \in As' \Longrightarrow Q\ A \leq Q'\ A$
shows $\{\!|P|\!\}$, $As \vdash_P G$, $\{\!|Q|\!\} \leq \{\!|P'|\!\}$, $As' \vdash_P G'$, $\{\!|Q'|\!\}$
by (*simp add*: *assms pag-def INF-superset-mono*[*OF* ‹$As' \leq As$› *ag.mono*] *le-funD*)

lemma *action*: — allow assertions to depend on assume A, needed for the example
fixes F :: ('v × 's × 's) *set*
assumes Q: $\bigwedge A\ v\ s\ s'.\ [\![A \in As;\ P\ A\ s;\ (v,\ s,\ s') \in F]\!] \Longrightarrow Q\ A\ v\ s'$
assumes G: $\bigwedge A\ v\ s\ s'.\ [\![A \in As;\ P\ A\ s;\ s \neq s';\ (v,\ s,\ s') \in F]\!] \Longrightarrow (s,\ s') \in G$
shows *prog.p2s* (*prog.action* F) $\leq$ $\{\!|\lambda A.\ wdp.floor\ A\ (P\ A)|\!\}$, $As \vdash_P G$, $\{\!|\lambda A\ v.\ wdp.ceiling\ A\ (Q\ A\ v)|\!\}$
by (*simp add*: *assms pag-def wdp.action INFI*)

lemmas *sup* = *ag.prog.sup*

lemma *bind*:
assumes $\bigwedge v.$ *prog.p2s* $(g\ v) \leq \{\!|\lambda A.\ Q'\ A\ v|\!\}$, $As \vdash_P G$, $\{\!|Q|\!\}$
assumes *prog.p2s* $f \leq \{\!|P|\!\}$, $As \vdash_P G$, $\{\!|Q'|\!\}$
shows *prog.p2s* $(f \gg\!= g) \leq \{\!|P|\!\}$, $As \vdash_P G$, $\{\!|Q|\!\}$
unfolding *pag-def*
by (*fastforce intro*: *INFI ag.prog.bind*[*rotated*]
order.trans[*OF assms*(*2*) *pag.mono*[*OF - - - order.refl*]]
order.trans[*OF assms*(*1*) *pag.mono*[**where** As'={A} **for** A], *simplified pag.singleton*]
simp flip: *pag.singleton*)+

lemma *parallel*:
assumes *prog.p2s* $c_1 \leq \{\!|P_1|\!\}$, $(\cup)\ G_2$ ‘ $A \vdash_P G_1$, $\{\!|Q_1|\!\}$

assumes *prog.p2s* $c_2 \leq \{\!|P_2|\!\}$, $(\cup)$ G_1 ‘ $A \vdash_P G_2$, $\{\!|Q_2|\!\}$
shows *prog.p2s* (*prog.parallel* c_1 c_2)
$\leq \{\!|\lambda R.\ P_1\ (R \cup G_2) \wedge P_2\ (R \cup G_1)|\!\}$, $A \vdash_P G_1 \cup G_2$, $\{\!|\lambda R\ v.\ Q_1\ (R \cup G_2)\ v \wedge Q_2\ (R \cup G_1)\ v|\!\}$
unfolding *pag-def*
by (*force intro*: *INFI ag.prog.parallel order.trans*[*OF assms*(*1*) *pag.mono*]
order.trans[*OF assms*(*2*) *pag.mono*]
simp flip: *pag.singleton*)

etc. – the other rules follow similarly

setup ‹*Sign.parent-path*›

20.2 Examples

There is not always a single (traditional) most general assume/guarantee specification (§2.1).

type-synonym *state* = *int* — just x
abbreviation (*input*) *incr* ≡ *prog.write* ((+) *1*) — atomic increment
abbreviation (*input*) *increases* :: *int rel* **where** *increases* ≡ $\{(x, x').\ x \leq x'\}$

lemma *ag-incr1*: — the precondition is stable as the rely is very strong
shows *prog.p2s incr* $\leq \{\!|(=)\ c|\!\}$, $\{\} \vdash$ *increases*, $\{\!|\langle(=)\ (c + 1)\rangle|\!\}$
by (*rule ag.prog.action*; *simp add*: *stable.empty*)

lemma *ag-incr2*: — note the weaker precondition due to the larger assume
shows *prog.p2s incr* $\leq \{\!|(\leq)\ c|\!\}$, *increases* $\vdash$ *increases*, $\{\!|\langle(\leq)\ (c + 1)\rangle|\!\}$
by (*rule ag.prog.action*; *auto simp*: *stable-def monotone-def le-bool-def*)

lemma *ag-incr1-par-incr1*:
shows *prog.p2s* (*incr* ‖ *incr*) $\leq \{\!|\lambda x.\ c \leq x|\!\}$, *increases* $\vdash$ *increases*, $\{\!|\lambda\text{-}\ x.\ c + 1 \leq x|\!\}$
apply (*rule ag.pre-pre*)
apply (*rule ag.pre-post*)
apply (*rule ag.prog.parallel*[**where** P_1=$\lambda x.\ c \leq x$ **and** P_2=$\lambda x.\ c \leq x$
and Q_1=$\lambda\text{-}\ x.\ c + 1 \leq x$ **and** Q_2=$\lambda\text{-}\ x.\ c + 1 \leq x$
and G_1=*increases* **and** G_2=*increases*, *simplified*])
apply (*rule ag.prog.action*; *simp add*: *stable-def monotone-def le-boolI*; *fail*)
apply (*rule ag.prog.action*; *simp add*: *stable-def monotone-def le-boolI*; *fail*)
apply *simp-all*
done

Using explicit stabilisation we can squash the two specifications for *incr* into a single one (§4).

lemma — postcondition cannot be simplified for arbitrary A
shows *prog.p2s incr* $\leq \{\!|$*wdp.ceiling A* $((=)\ c)|\!\}$, $A \vdash$ *increases*, $\{\!|\langle$*wdp.ceiling A* $(\lambda s.$ *wdp.ceiling A* $((=)\ c)\ (s - 1))\rangle|\!\}$
by (*rule ag.pre-pre*[*OF wdp.action*]) (*simp add*: *wdp.floor-ceiling*)+

— The set of assumes that commute with the increment
abbreviation (*input*) *comm-xpp* :: *int rel set* **where**
comm-xpp ≡ $\{A.\ \forall p\ s.$ *wdp.ceiling A p* $(s - 1) =$ *wdp.ceiling A* $(\lambda s.\ p\ (s - 1))\ s\}$

lemma *pag-incr*: — postcondition can be simplified wrt *comm-xpp*
shows *prog.p2s incr* $\leq \{\!|\lambda A.$ *wdp.ceiling A* $((=)\ c)|\!\}$, *comm-xpp* $\vdash_P$ *increases*, $\{\!|\lambda A.\ \langle$*wdp.ceiling A* $((=)\ (c + 1))\rangle|\!\}$
apply (*rule order.trans*[*OF* - *pag.mono*[*OF* - *order.refl order.refl*]])
apply (*rule pag.action*[**where** P=$\lambda A.$ *wdp.ceiling A* $((=)\ c)$
and Q=$\lambda A\ v\ s.$ *wdp.ceiling A* $((=)\ c)\ (s - 1)$])
apply (*simp-all add*: *wdp.floor-ceiling eq-diff-eq*)
done

— the two earlier specifications can be recovered

lemma
shows *prog.p2s incr* ≤ ⦃(=) *c*⦄, {} ⊢ *increases*, ⦃⟨(=) (*c* + *1*)⟩⦄
apply (*rule order.trans*[*OF pag-incr*])
apply (*subst pag.singleton*[*symmetric*])
apply (*rule pag.mono*; *force*)
done

lemma
shows *prog.p2s incr* ≤ ⦃(≤) *c*⦄, *increases* ⊢ *increases*, ⦃⟨(≤) (*c* + *1*)⟩⦄
apply (*rule order.trans*[*OF pag-incr*[**where** *c*=*c*]])
apply (*subst pag.singleton*[*symmetric*])
apply (*rule pag.mono*; *force simp*: *wdp.ceiling-def order-rtrancl dest*: *zless-imp-add1-zle*)
done

21 Example: inhabitation

The following is a simple example of showing that a specification is inhabited.

lemma
shows ⟪*0*::*nat*, [(*self*, *1*), (*self*, *2*)], *Some* ()⟫
≤ *prog.p2s* (*prog.while* ⟨*prog.write* ((+) *1*) ≫ (*prog.return* (*Inl* ()) ⊔ *prog.return* (*Inr* ()))⟩ ())
apply (*rule inhabits.I*)
apply (*rule inhabits.pre*)
apply (*subst prog.while.simps*)
apply (*simp add*: *prog.bind.bind*)
apply (*rule inhabits.trans*)
apply (*rule inhabits.prog.bind*)
apply (*rule inhabits.prog.action-step*)
apply *force*
apply (*simp add*: *prog.bind.returnL*)
apply (*rule inhabits.trans*)
apply (*rule inhabits.prog.bind*)
apply (*rule inhabits.prog.supL*)
apply (*rule inhabits.tau*)
apply (*simp add*: *spec.idle.p2s-le*; *fail*)
apply (*simp add*: *prog.bind.returnL*)
apply (*subst prog.while.simps*)
apply (*simp add*: *prog.bind.bind*)
apply (*rule inhabits.trans*)
apply (*rule inhabits.prog.bind*)
apply (*rule inhabits.prog.action-step*)
apply *force*
apply (*simp add*: *prog.bind.returnL*)
apply (*rule inhabits.trans*)
apply (*rule inhabits.prog.bind*)
apply (*rule inhabits.prog.supR*)
apply (*rule inhabits.tau*)
apply (*simp add*: *spec.idle.p2s-le*; *fail*)
apply (*simp add*: *prog.bind.returnL*)
apply (*rule inhabits.tau*)
apply (*simp add*: *spec.idle.p2s-le*; *fail*)
apply (*simp add*: *prog.p2s.return spec.interference.cl.return spec.return.rel-le*; *fail*)
apply *simp*
done

22 Example: findP

We demonstrate assume/guarantee reasoning by showing the safety of *findP*, a classic exercise in concurrency verification. It has been treated by at least:

- Karp and Miller (1969, Example 5.1)
- Rosen (1976, §3)
- Owicki and Gries (1976, §4 Example 2)
- Jones (1983, §2.4)
- Xu et al. (1994, §3.1)
- Brookes (1996, p161) (no proof)
- de Roever et al. (2001, Examples 3.57 and 8.26) (atomic guarded commands)
- Dingel (2002, §6.2) (refinement)
- Prensa Nieto (2003, §10) (mechanized, arbitrary number of threads)
- Apt, de Boer, and Olderog (2009, §7.4, §8.6)
- Hayes and Jones (2017, §4) (refinement)

We take the task to be of finding the first element of a given array A that satisfies a given predicate *pred*, if it exists, or yielding *length* A if it does not. This search is performed with two threads: one searching the even indices and the other the odd. There is the possibility of a thread terminating early if it notices that the other thread has found a better candidate than it could.

We generalise previous treatments by allowing the predicate to be specified modularly and to be a function of the state. It is required to be pure, i.e., it cannot change the observable/shared state, though it could have its own local state.

Our search loops are defined recursively; one could just as easily use *prog.while*. We use a list and not an array for simplicity – at this level of abstraction there is no difference – and a mix of variables, where the monadic ones are purely local and the state-based are shared between the threads. The lens allows the array to be a value or reside in the (observable/shared) state.

type-synonym $'s\ state = (nat \times nat) \times 's$

abbreviation $foundE :: nat \Longrightarrow 's\ state$ **where** $foundE \equiv fst_L\ ;_L\ fst_L$
abbreviation $foundO :: nat \Longrightarrow 's\ state$ **where** $foundO \equiv snd_L\ ;_L\ fst_L$

context
  fixes $pred :: 'a \Rightarrow ('s, bool)\ prog$
  fixes $predPre :: 's\ pred$
  fixes $predP :: 'a \Rightarrow 's\ pred$
  fixes $A :: 's\ rel$
  fixes $array :: 'a\ list \Longrightarrow 's$
  — A guarantee of *Id* indicates that *pred a* is observationally pure.
  assumes *iag-pred*: $\bigwedge a.\ prog.p2s\ (pred\ a) \leq \{\!|predPre \wedge \langle a \rangle \in SET\ get_{array}|\!\}, A^{=} \cap Id_{get_{array}} \cap ceilr\ predPre \cap Id_{predP\ a} \vdash Id, \{\!|\lambda rv.\ \langle rv \rangle = predP\ a|\!\}$
begin

abbreviation $array' :: 'a\ list \Longrightarrow 's\ state$ **where** $array' \equiv array\ ;_L\ snd_L$

partial-function (*lfp*) *findP-loop-evens* :: $nat \Rightarrow ('s\ state, unit)\ prog$ **where**
  findP-loop-evens $i =$
    do { $fO \leftarrow prog.read\ get_{foundO}$
      ; $prog.whenM\ (i < fO)$
        (*do* { $v \leftarrow prog.read\ (\lambda s.\ get_{array'}\ s\ !\ i)$

; $b \leftarrow$ *prog.localize* (*pred v*)
; *if b then prog.write* ($\lambda s.$ put_{foundE} *s i*) *else findP-loop-evens* (*i* + *2*)
})
}

partial-function (*lfp*) *findP-loop-odds* :: *nat* ⇒ (*'s state*, *unit*) *prog* **where**
findP-loop-odds i =
do { *fE* ← *prog.read* get_{foundE}
; *prog.whenM* (*i* < *fE*)
(*do* { *v* ← *prog.read* ($\lambda s.$ $get_{array'}$ *s* ! *i*)
; *b* ← *prog.localize* (*pred v*)
; *if b then prog.write* ($\lambda s.$ put_{foundO} *s i*) *else findP-loop-odds* (*i* + *2*)
})
}

definition *findP* :: (*'s*, *nat*) *prog* **where**
findP = *prog.local* (
do { *N* ← *prog.read* (*SIZE* $get_{array'}$)
; *prog.write* ($\lambda s.$ put_{foundE} *s N*)
; *prog.write* ($\lambda s.$ put_{foundO} *s N*)
; (*findP-loop-evens 0* ‖ *findP-loop-odds 1*)
; *fE* ← *prog.read* (get_{foundE})
; *fO* ← *prog.read* (get_{foundO})
; *prog.return* (*min fE fO*)
})

Relies and guarantees **abbreviation** (*input*) A' :: *'s rel* **where** $A' \equiv A^{=} \cap ceilr\ predPre \cap (\bigcap a.\ Id_{predP\ a})$

definition *AE* :: *'s state rel* **where**
$AE = UNIV \times_R A' \cap Id_{get_{array'}} \cap Id_{get_{foundE}} \cap \leq_{get_{foundO}}$

definition *GE* :: *'s state rel* **where**
$GE = Id_{snd} \cap Id_{get_{foundO}} \cap \leq_{get_{foundE}}$

definition *AO* :: *'s state rel* **where**
$AO = UNIV \times_R A' \cap Id_{get_{array'}} \cap Id_{get_{foundO}} \cap \leq_{get_{foundE}}$

definition *GO* :: *'s state rel* **where**
$GO = Id_{snd} \cap Id_{get_{foundE}} \cap \leq_{get_{foundO}}$

lemma *AG-refl-trans*:
shows
refl AE
refl AO
trans A ⟹ *trans AE*
trans A ⟹ *trans AO*
refl GE
refl GO
trans GE
trans GO
unfolding *AE-def AO-def GE-def GO-def*
by (*auto simp*: *refl-inter-conv refl-relprod-conv*
intro!: *trans-Int refl-UnionI refl-INTER trans-INTER*)

lemma *AG-containment*:
shows *GO* ⊆ *AE*
and *GE* ⊆ *AO*
by (*auto simp*: *AE-def AO-def GO-def GE-def refl-onD*[*OF ceilr.refl*])

lemma *G-containment*:
shows $GE \cup GO \subseteq UNIV \times_R Id$
by (*fastforce simp*: *GE-def GO-def*)

Safety proofs **lemma** *ag-findP-loop-evens*:
shows *prog.p2s* (*findP-loop-evens i*)
$\leq \{\!|\langle even\ i\rangle \wedge (\lambda s.\ predPre\ (snd\ s)) \wedge get_{foundE} = SIZE\ get_{array'} \wedge get_{foundO} \leq SIZE\ get_{array'}|\!\}, AE \vdash GE,$
$\{\!|\lambda\text{-}.\ (get_{foundE} < SIZE\ get_{array'} \longrightarrow localize1\ predP\ \$\$\ get_{array'}\ !\ get_{foundE})$
$\wedge (\forall j.\ \langle i \leq j \wedge even\ j\rangle \wedge \langle j\rangle < pred\text{-}min\ get_{foundE}\ get_{foundO} \longrightarrow \neg\ localize1\ predP\ \$\$\ get_{array'}\ !\ \langle j\rangle)|\!\}$
proof(*intro ag.gen-asm*,
induct arbitrary: *i rule*: *findP-loop-evens.fixp-induct*[*case-names bot adm step*])
case (*step R i*) **show** *?case*
apply (*rule iag.init*)
apply (*rule iag.intro*)+
— else branch, recursive call
apply (*rename-tac v va vb*)
apply (*rule-tac P*=$\langle va\rangle = get_{array'}\ !\ \langle i\rangle \wedge \langle vb\rangle = localize1\ predP\ va$
in *iag.stable-augment*[*OF step.hyps*])
apply (*simp add*: ‹*even i*›; *fail*)
apply *clarsimp*
apply (*metis* ‹*even i*› *even-Suc less-Suc-eq not-less*)
apply (*force simp*: *GE-def AE-def stable-def monotone-def*)
— *prog.localize* (*pred ...*)
apply (*rename-tac v va*)
apply (*rule-tac Q*=$\lambda vb.\ (\lambda s.\ predPre\ (snd\ s)) \wedge get_{foundE} = SIZE\ get_{array'} \wedge get_{foundO} \leq SIZE\ get_{array'} \wedge \langle v\rangle \leq SIZE\ get_{array'} \wedge \langle va\rangle = get_{array'}\ !\ \langle i\rangle \wedge \langle vb\rangle = localize1\ predP\ va$
in *ag.post-imp*)
apply (*clarsimp simp*: *GE-def exI*[**where** x=$\langle i\rangle \leq get_{foundE}$]; *fail*)
apply (*rule iag.pre-g*[**where** *G'*=*GE*, *OF iag.stable-augment-post*[*OF iag.augment-a*[**where** *A'*=*AE*, *OF ag.prog.localize-lift*[*OF iag-pred*, *simplified*]]]])
apply (*fastforce simp*: *AE-def stable-def monotone-def*)
apply (*metis AG-refl-trans*(*5*) *refl-alt-def*)
apply (*rule iag.intro*)+
— precondition
apply *force*
— assume
apply (*intro conjI Int-greatest INT-greatest ceilr.largest*)
apply ((*fastforce simp*: *stable-def monotone-def AE-def*)+)[*6*]
apply (*clarsimp simp*: *stable-def monotone-def AE-def GE-def*; *rule exI*[**where** x=$\langle i\rangle \leq get_{foundE}$]; *clarsimp*; *metis*)
apply (*fastforce simp*: *stable-def monotone-def AE-def*)+
done
qed *simp-all*

lemma *ag-findP-loop-odds*:
shows *prog.p2s* (*findP-loop-odds i*)
$\leq \{\!|\langle odd\ i\rangle \wedge (\lambda s.\ predPre\ (snd\ s)) \wedge get_{foundO} = SIZE\ get_{array'} \wedge get_{foundE} \leq SIZE\ get_{array'}|\!\}, AO \vdash GO,$
$\{\!|\lambda\text{-}.\ (get_{foundO} < SIZE\ get_{array'} \longrightarrow localize1\ predP\ \$\$\ get_{array'}\ !\ get_{foundO})$
$\wedge (\forall j.\ \langle i \leq j \wedge odd\ j\rangle \wedge \langle j\rangle < pred\text{-}min\ get_{foundE}\ get_{foundO} \longrightarrow \neg\ localize1\ predP\ \$\$\ get_{array'}\ !\ \langle j\rangle)|\!\}$
proof(*intro ag.gen-asm*,
induct arbitrary: *i rule*: *findP-loop-odds.fixp-induct*[*case-names bot adm step*])
case (*step R i*) **show** *?case*
apply (*rule iag.init*)
apply (*rule iag.intro*)+

— else branch, recursive call
apply (*rename-tac v va vb*)
apply (*rule-tac P=*$\langle va\rangle = get_{array'} \,!\, \langle i\rangle \wedge \langle vb\rangle = localize1\ predP\ va$
in *iag.stable-augment*[*OF step.hyps*])
apply (*simp add*: ‹*odd i*›; *fail*)
apply *clarsimp*
apply (*metis* ‹*odd i*› *even-Suc less-Suc-eq not-less*)
apply (*force simp*: *GO-def AO-def stable-def monotone-def*)
— *prog.localize* (*pred* ...)
apply (*rename-tac v va*)
apply (*rule-tac Q=*$\lambda vb.\ (\lambda s.\ predPre\ (snd\ s)) \wedge get_{foundO} = SIZE\ get_{array'} \wedge get_{foundE} \leq SIZE\ get_{array'} \wedge \langle v\rangle \leq SIZE\ get_{array'} \wedge \langle va\rangle = get_{array'} \,!\, \langle i\rangle \wedge \langle vb\rangle = localize1\ predP\ va$
in *ag.post-imp*)
apply (*clarsimp simp*: *GO-def exI*[**where** $x=\langle i\rangle \leq get_{foundO}$]; *fail*)
apply (*rule iag.pre-g*[**where** *G'=GO*, *OF iag.stable-augment-post*[*OF iag.augment-a*[**where** *A'=AO*, *OF ag.prog.localize-lift*[*OF iag-pred*, *simplified*]]]])
apply (*fastforce simp*: *AO-def stable-def monotone-def*)
apply (*metis AG-refl-trans*(*6*) *refl-alt-def*)
apply (*rule iag.intro*)+
— precondition
apply *force*
— assume
apply (*intro conjI Int-greatest INT-greatest ceilr.largest*)
apply ((*fastforce simp*: *AO-def stable-def monotone-def*)+)[*6*]
apply (*clarsimp simp*: *AO-def GO-def stable-def monotone-def*; *rule exI*[**where** $x=\langle i\rangle \leq get_{foundO}$]; *clarsimp*; *metis*)
apply (*fastforce simp*: *AO-def stable-def monotone-def*)+
done
qed *simp-all*

theorem *ag-findP*:
shows *prog.p2s findP*
$\leq \{\!|predPre|\!\},\ A' \cap Id_{get_{array}}$
$\vdash Id,\ \{\!|\lambda v\ s.\ v = (LEAST\ i.\ i < SIZE\ get_{array}\ s \longrightarrow predP\ (get_{array}\ s\ !\ i)\ s)|\!\}$
unfolding *findP-def*
apply (*rule ag.prog.local*)
apply (*rule iag.init*)
apply (*rule iag.intro*)+
apply (*rule iag.augment-post-imp*[**where** $Q=\lambda v.\ get_{foundE} \leq SIZE\ get_{array'} \wedge get_{foundO} \leq SIZE\ get_{array'}$])
apply (*rule iag.pre-g*[*OF - G-containment*])
apply (*rule iag.stable-augment-frame*)
apply (*rule iag.parallel*[*OF ag-findP-loop-evens ag-findP-loop-odds - AG-containment order.refl*])
— postcondition from *iag.parallel*
apply *clarsimp*
apply (*rule Least-equality*, *linarith*)
subgoal for *x y s z* **by** (*clarsimp simp*: *min-le-iff-disj not-less not-le dest*!: *spec*[**where** *x=z*])
— stability for *iag.stable-augment-frame*
apply (*force simp*: *stable-def monotone-def AE-def AO-def GE-def GO-def*)
apply (*rule iag.intro*)+
— precondition
apply *fastforce*
— assume
apply (*simp*;
intro conjI Int-greatest INT-greatest ceilr.largest;
fastforce simp: *AE-def AO-def stable-def monotone-def*)
done

end

We conclude by showing how we can instantiate the above with a *coprime* predicate.

setup ‹*Sign.mandatory-path gcd*›

type-synonym *'s state* = (*nat* × *nat*) × *'s*

abbreviation *x* :: *nat* ⟹ *'s gcd.state* **where** *x* ≡ fst_L ;$_L$ fst_L
abbreviation *y* :: *nat* ⟹ *'s gcd.state* **where** *y* ≡ snd_L ;$_L$ fst_L

definition *seq* :: *nat* ⇒ *nat* ⇒ (*'s*, *nat*) *prog* **where**
seq a b =
prog.local (
do { *prog.write* (λ*s*. $put_{gcd.x}$ *s a*)
; *prog.write* (λ*s*. $put_{gcd.y}$ *s b*)
; *prog.while* (λ-.
do { *xv* ← *prog.read* $get_{gcd.x}$
; *yv* ← *prog.read* $get_{gcd.y}$
; *if xv* = *yv*
then prog.return (*Inr* ())
else (*do* { (*if xv* < *yv*
then prog.write (λ*s*. $put_{gcd.y}$ *s* (*yv* − *xv*))
else prog.write (λ*s*. $put_{gcd.x}$ *s* (*xv* − *yv*)))
; *prog.return* (*Inl* ()) })
}) ()
; *prog.read* $get_{gcd.x}$
})

setup ‹*Sign.parent-path*›

setup ‹*Sign.mandatory-path ag.gcd*›

lemma *seq*:
shows *prog.p2s* (*gcd.seq a b*) ≤ ⦃⟨*True*⟩⦄, *UNIV* ⊢ *Id*, ⦃λ*v*. ⟨*v* = *gcd a b*⟩⦄
unfolding *gcd.seq-def*
apply (*rule ag.prog.local*)
apply (*rule iag.init*)
apply (*rule iag.intro iag.while*[**where** *I*=λ- *s*. *gcd* ($get_{gcd.x}$ *s*) ($get_{gcd.y}$ *s*) = *gcd a b*])+
— precond
apply (*auto simp*: *gcd-diff1-nat*)[*1*]
apply (*metis gcd.commute gcd-diff1-nat less-or-eq-imp-le*)
— assume
apply (*intro stable.intro stable.local-only INFI infI*)
apply *auto*
done

setup ‹*Sign.parent-path*›

definition *findP-coprime* :: (*nat* × *nat list*, *nat*) *prog* **where**
findP-coprime = *findP* (λ*a*. *prog.read* get_{fst_L} ⤜ *gcd.seq a* ⤜ (λ*c*. *prog.return* (*c* = *1*))) snd_L

lemma *ag-findP-coprime'*:
shows *prog.p2s findP-coprime*
≤ ⦃⟨*True*⟩⦄, *Id*
⊢ *Id*, ⦃λ*rv s*. *rv* = (*LEAST i*. *i* < *length* (get_{snd_L} *s*) ⟶ *coprime* (get_{fst_L} *s*) (get_{snd_L} *s* ! *i*))⦄
unfolding *findP-coprime-def*
apply (*rule iag.init*)
apply (*rule ag-findP*[**where** *A*=*Id* **and** *array*=snd_L **and** *predP*=λ*b s*. *coprime* (get_{fst_L} *s*) *b* **and** *predPre*=⟨*True*⟩])

apply (*rule iag.init*)
apply (*rule iag.intro*)+
apply (*rule-tac* Q=$\langle\langle v\rangle = get_{fst_L}\rangle$ **in** *iag.augment-post-imp*)
apply (*rule iag.stable-augment-frame*)
apply (*rule iag.pre*[*OF ag.gcd.seq*, **where** A'=*Id* **and** P'=$\langle True\rangle$, *simplified*, *OF order.refl*])
apply (*clarsimp simp*: *ac-simps coprime-iff-gcd-eq-1 simp flip*: *One-nat-def*; *fail*)
apply (*force simp*: *stable-def monotone-def*)
apply (*rule iag.intro*)+
apply (*simp*; *intro conjI INT-greatest ceilr.largest*; *fastforce simp*: *stable-def monotone-def*)+
done

lemma *ag-findP-coprime*: — Shuffle the parameter to the precondition, exploiting purity.
shows *prog.p2s findP-coprime*
$\leq$ $\{\!|\langle a\rangle = get_{fst_L}|\!\}$, *Id*
$\vdash$ *Id*, $\{\!|\lambda rv\ s.\ rv = (LEAST\ i.\ i < length\ (get_{snd_L}\ s) \longrightarrow coprime\ a\ (get_{snd_L}\ s\ !\ i))|\!\}$
apply (*rule ag.pre-pre*)
apply (*rule ag.stable-augment-post*[*OF ag-findP-coprime'*])
apply (*fastforce simp*: *stable-def*)+
done

23 Example: data refinement (search)

We show a very simple example of data refinement: implementing sets with functional queues for breadth-first search (BFS). The objective here is to transfer a simple correctness property proven on the abstract level to the concrete level.
Observations:

- there is no concurrency in the BFS: this is just about data refinement
  - however arbitrary interference is allowed
- the abstract level does not require the implementation of the pending set to make progress
- the concrete level does not require a representation invariant
- depth optimality is not shown

References:

- queue ADT: $ISABELLE_HOME/src/Doc/Codegen/Introduction.thy
- BFS verification:
  - J. C. Filliâtre http://toccata.lri.fr/gallery/vstte12_bfs.en.html
  - $AFP/Refine_Monadic/examples/Breadth_First_Search.thy
  - our model is quite different

setup ‹*Sign.mandatory-path pending*›

record ('*a*, '*s*) *interface* =
init :: ('*s*, *unit*) *prog*
enq :: '*a* ⇒ ('*s*, *unit*) *prog*
deq :: ('*s*, '*a option*) *prog*

type-synonym '*a abstract* = '*a set*

definition *abstract* :: ('*a*, '*a pending.abstract* × '*s*) *pending.interface* **where**
abstract =

```
    (| pending.interface.init = prog.write (map-prod ⟨{}⟩ id)
    , pending.interface.enq = λx. prog.write (map-prod (insert x) id)
    , pending.interface.deq = prog.action ({(None, s, s) |s. fst s = {}}
                                  ∪ {(Some x, (insert x X, s), (X, s)) |X s x. True})
    |)

type-synonym 'a concrete = 'a list × 'a list — a queue

fun cdeq-update :: 'a pending.concrete × 's ⇒ 'a option × 'a pending.concrete × 's where
  cdeq-update (([], []), s) = (None, (([], []), s))
| cdeq-update ((xs, []), s) = cdeq-update (([], rev xs), s)
| cdeq-update ((xs, y # ys), s) = (Some y, ((xs, ys), s))

definition concrete :: ('a, 'a pending.concrete × 's) pending.interface where
  concrete =
    (| pending.interface.init = prog.write (map-prod ⟨([], [])⟩ id)
    , pending.interface.enq = λx. prog.write (map-prod (map-prod ((#) x) id) id)
    , pending.interface.deq = prog.det-action pending.cdeq-update
    |)

abbreviation absfn' :: 'a pending.concrete ⇒ 'a list where — queue as a list
  absfn' s ≡ snd s @ rev (fst s)

definition absfn :: 'a pending.concrete ⇒ 'a pending.abstract where
  absfn s = set (pending.absfn' s)

setup ‹Sign.mandatory-path ag›

lemma init:
  fixes Q :: unit ⇒ 'a pending.abstract × 's ⇒ bool
  fixes A :: 's rel
  assumes stable (Id ×R A) (Q ())
  shows prog.p2s (pending.init pending.abstract) ≤ ⦃λs. Q () ({}, snd s)⦄, Id ×R A ⊢ UNIV ×R Id, ⦃Q⦄
using assms by (auto intro: ag.prog.action simp: pending.abstract-def stable-def monotone-def)

lemma enq:
  fixes x :: 'a
  fixes Q :: unit ⇒ 'a pending.abstract × 's ⇒ bool
  fixes A :: 's rel
  assumes stable (Id ×R A) (Q ())
  shows prog.p2s (pending.enq pending.abstract x) ≤ ⦃λs. Q () (insert x (fst s), snd s)⦄, Id ×R A ⊢ UNIV ×R
Id, ⦃Q⦄
using assms by (auto intro: ag.prog.action simp: pending.abstract-def stable-def monotone-def)

lemma deq:
  fixes Q :: 'a option ⇒ 'a pending.abstract × 's ⇒ bool
  fixes A :: 's rel
  assumes ⋀v. stable (Id ×R A) (Q v)
  shows prog.p2s (pending.deq pending.abstract) ≤ ⦃λs. if fst s = {} then Q None s else (∀x X. fst s = insert x
X ⟶ Q (Some x) (X, snd s))⦄, Id ×R A ⊢ UNIV ×R Id, ⦃Q⦄
unfolding pending.abstract-def pending.interface.simps
by (rule ag.prog.action)
   (use assms in ‹auto simp: stable-def monotone-def le-bool-def split: if-split-asm›)

setup ‹Sign.parent-path›

setup ‹Sign.parent-path›
```

setup ‹*Sign.mandatory-path set*›

record ('*a*, '*s*) *interface* =
  init :: ('*s*, *unit*) *prog*
  ins :: '*a* ⇒ ('*s*, *unit*) *prog*
  mem :: '*a* ⇒ ('*s*, *bool*) *prog*

type-synonym '*a abstract* = '*a list* — model finite sets

definition *abstract* :: ('*a*, '*s* × '*a set.abstract* × '*t*) *set.interface* **where**
  abstract =
    ⦇ *set.interface.init* = *prog.write* (*map-prod id* (*map-prod* ⟨[]⟩ *id*))
    , *set.interface.ins* = λ*x*. *prog.write* (*map-prod id* (*map-prod* ((#) *x*) *id*))
    , *set.interface.mem* = λ*x*. *prog.read* (λ*s*. *x* ∈ *set* (*fst* (*snd s*)))
    ⦈

setup ‹*Sign.mandatory-path ag*›

lemma *init*:
  fixes *Q* :: *unit* ⇒ '*s* × '*a set.abstract* × '*t* ⇒ *bool*
  fixes *A* :: '*s rel*
  assumes *stable* (*A* ×$_R$ *Id* ×$_R$ *B*) (*Q* ())
  shows *prog.p2s* (*set.init set.abstract*) ≤ ⦇λ*s*. *Q* () (*fst s*, [], *snd* (*snd s*))⦈, *A* ×$_R$ *Id* ×$_R$ *B* ⊢ *Id* ×$_R$ *UNIV* ×$_R$ *Id*, ⦇*Q*⦈
using *assms* **by** (*auto intro*: *ag.prog.action simp*: *set.abstract-def stable-def monotone-def*)

lemma *ins*:
  fixes *x* :: '*a*
  fixes *Q* :: *unit* ⇒ '*s* × '*a set.abstract* × '*t* ⇒ *bool*
  fixes *A* :: '*s rel*
  assumes *stable* (*A* ×$_R$ *Id* ×$_R$ *B*) (*Q* ())
  shows *prog.p2s* (*set.ins set.abstract x*) ≤ ⦇λ*s*. *Q* () (*fst s*, *x* # *fst* (*snd s*), *snd* (*snd s*))⦈, *A* ×$_R$ *Id* ×$_R$ *B* ⊢ *Id* ×$_R$ *UNIV* ×$_R$ *Id*, ⦇*Q*⦈
using *assms* **by** (*auto intro*: *ag.prog.action simp*: *set.abstract-def stable-def monotone-def*)

lemma *mem*:
  fixes *Q* :: *bool* ⇒ '*s* × '*a set.abstract* × '*t* ⇒ *bool*
  assumes ⋀*v*. *stable* (*A* ×$_R$ *Id* ×$_R$ *B*) (*Q v*)
  shows *prog.p2s* (*set.mem set.abstract x*) ≤ ⦇λ*s*. *Q* (*x* ∈ *set* (*fst* (*snd s*))) *s*⦈, *A* ×$_R$ *Id* ×$_R$ *B* ⊢ *Id* ×$_R$ *UNIV* ×$_R$ *Id*, ⦇*Q*⦈
using *assms* **by** (*auto intro*: *ag.prog.action simp*: *set.abstract-def stable-def monotone-def*)

setup ‹*Sign.parent-path*›

setup ‹*Sign.parent-path*›

context
  fixes *pending* :: ('*a*, '*p* × '*a set.abstract* × '*s*) *pending.interface*
  fixes *f* :: '*a* ⇒ '*a list*
begin

definition *loop* :: '*a pred* ⇒ ('*p* × '*a set.abstract* × '*s*, '*a option*) *prog* **where**
  loop p =
    prog.while (λ-.
      do { *aopt* ← *pending.deq pending*
        ; *case aopt of*
            None ⇒ *prog.return* (*Inr None*)
          | *Some x* ⇒

```
        if p x
        then prog.return (Inr (Some x))
        else do { prog.app (λy. do { b ← set.mem set.abstract y;
                                     prog.unlessM b (do { set.ins set.abstract y
                                                        ; pending.enq pending y }) })
                           (f x)
             ; prog.return (Inl ())
             }
    }) ()

definition main :: 'a pred ⇒ 'a ⇒ ('p × 'a set.abstract × 's, 'a option) prog where
  main p x =
    do {
      set.init set.abstract
    ; pending.init pending
    ; set.ins set.abstract x
    ; pending.enq pending x
    ; loop p
    }

definition search :: 'a pred ⇒ 'a ⇒ ('s, 'a option) prog where
  search p x = prog.local (prog.local (main p x))

end

abbreviation (input) aloop ≡ loop pending.abstract
abbreviation (input) amain ≡ main pending.abstract
abbreviation (input) asearch ≡ search pending.abstract
abbreviation (input) bfs ≡ search pending.concrete

lemma
  shows pending-g: UNIV ×R Id ⊆ UNIV ×R UNIV ×R Id
    and set-g: Id ×R UNIV ×R Id ⊆ UNIV ×R UNIV ×R Id
by fastforce+

context
  fixes f :: 'a ⇒ 'a list
  fixes P :: 'a pred
  fixes x0 :: 'a
begin

abbreviation (input) step :: 'a rel where
  step ≡ {(x, y). y ∈ set (f x)}

abbreviation (input) path :: 'a rel where
  path ≡ step*

definition aloop-invP :: 'a pending.abstract ⇒ 'a set.abstract ⇒ bool where
  aloop-invP q v ⟷
       q ⊆ set v
     ∧ set v ⊆ path `` {x0}
     ∧ set v ∩ Collect P ⊆ q
     ∧ x0 ∈ set v

definition vclosureP :: 'a ⇒ 'a pending.abstract ⇒ 'a set.abstract ⇒ bool where
  vclosureP x q v ⟷ (x ∈ set v − q ⟶ step `` {x} ⊆ set v)

definition search-postP :: 'a option ⇒ bool where
```

search-postP rv = (*case rv of*
  None ⇒ *finite* (*path* `` {$x_0$}) ∧ (*path* `` {x_0} ∩ *Collect P* = {})
 | *Some y* ⇒ (x_0, *y*) ∈ *path* ∧ *P y*)

abbreviation *aloop-inv s* ≡ *aloop-invP* (*fst s*) (*fst* (*snd s*))
abbreviation *vclosure x s* ≡ *vclosureP x* (*fst s*) (*fst* (*snd s*))
abbreviation *search-post rv* ≡ ⟨*search-postP rv*⟩

lemma *vclosureP-closed*:
 assumes *set v* ⊆ *path* `` {x_0}
 assumes ∀ *y*. *vclosureP y* {} *v*
 assumes x_0 ∈ *set v*
 shows *path* `` {x_0} = *set v*
proof −
 have *y* ∈ *set v* **if** (x_0, *y*) ∈ *path* **for** *y*
  using *that assms*(*2*,*3*) **by** *induct* (*auto simp*: *vclosureP-def*)
 with *assms*(*1*) **show** *?thesis*
  by *fast*
qed

lemma *vclosureP-app*:
 assumes ∀ *y*. *x* ≠ *y* ⟶ *local.vclosureP y q v*
 assumes *set* (*f x*) ⊆ *set v*
 shows *vclosureP y q v*
using *assms* **by** (*fastforce simp*: *vclosureP-def*)

lemma *vclosureP-init*:
 shows *vclosureP x* {x_0} [x_0]
by (*simp add*: *vclosureP-def*)

lemma *vclosureP-step*:
 assumes ∀ *z*. *x* ≠ *z* ⟶ *vclosureP z q v*
 assumes *x* ≠ *z*
 shows *vclosureP z* (*insert y q*) (*y* # *v*)
using *assms* **by** (*fastforce simp*: *vclosureP-def*)

lemma *vclosureP-dequeue*:
 assumes ∀ *z*. *vclosureP z* (*insert x q*) *v*
 assumes *x* ≠ *z*
 shows *vclosureP z q v*
using *assms* **by** (*fastforce simp*: *vclosureP-def*)

lemma *aloop-invPD*:
 assumes *aloop-invP q v*
 assumes *x* ∈ *q*
 shows (x_0, *x*) ∈ *path*
using *assms* **by** (*auto simp*: *aloop-invP-def*)

lemma *aloop-invP-init*:
 shows *aloop-invP* {x_0} [x_0]
by (*simp add*: *aloop-invP-def*)

lemma *aloop-invP-step*:
 assumes *aloop-invP q v*
 assumes (x_0, *x*) ∈ *path*
 assumes *y* ∈ *set* (*f x*) − *set v*
 shows *aloop-invP* (*insert y q*) (*y* # *v*)
using *assms* **by** (*auto simp*: *aloop-invP-def elim*: *rtrancl-into-rtrancl*)

lemma *aloop-invP-dequeue*:
assumes *aloop-invP* (*insert x q*) *v*
assumes ¬ *P x*
shows *aloop-invP q v*
using *assms* **by** (*auto simp*: *aloop-invP-def*)

lemma *search-postcond-None*:
assumes *aloop-invP* {} *v*
assumes ∀ *y*. *vclosureP y* {} *v*
shows *search-postP None*
using *assms* **by** (*auto simp*: *search-postP-def aloop-invP-def dest*: *vclosureP-closed*)

lemma *search-postcond-Some*:
assumes *aloop-invP q v*
assumes *x* ∈ *q*
assumes *P x*
shows *search-postP* (*Some x*)
using *assms* **by** (*auto simp*: *search-postP-def aloop-invP-def*)

lemmas *stable-simps* =
prod.sel
split-def
sum.simps

lemma *ag-aloop*:
shows *prog.p2s* (*aloop f P*) ≤ ⦃*aloop-inv* ∧ (∀ *x*. *vclosure x*)⦄, *Id* ×$_R$ *Id* ×$_R$ *UNIV* ⊢ *UNIV* ×$_R$ *UNIV* ×$_R$ *Id*, ⦃*search-post*⦄
unfolding *loop-def*
apply (*rule ag.prog.while*[*OF - - stable.Id-fst-fst-snd*])
apply (*rule ag.prog.bind*)
apply (*rule ag.prog.case-option*)
apply (*rule ag.prog.return*)
apply ((*simp only*: *stable-simps*)?; (*rule stable.intro*)+; *fail*)
apply (*rule ag.prog.if*)
apply (*rule ag.prog.return*)
apply ((*simp only*: *stable-simps*)?; (*rule stable.intro*)+; *fail*)
apply (*rule ag.prog.bind*)
apply (*rule ag.prog.return*)
apply ((*simp only*: *stable-simps*)?; (*rule stable.intro*)+; *fail*)
apply (*rename-tac x*)
apply (*rule-tac Q*=λ-. *aloop-inv* ∧ (∀ *y*. ⟨*x* ≠ *y*⟩ ⟶ *vclosure y*) ∧ (λ*s*. *set* (*f x*) ⊆ *set* (*fst* (*snd s*)) ∧ (*x*$_0$, *x*) ∈ *path*) **in** *ag.post-imp*)
apply (*force elim*: *vclosureP-app*)
apply (*rule ag.prog.app*)
apply (*rule ag.prog.bind*)
apply (*rule ag.prog.if*)
apply (*rule ag.prog.return*)
apply ((*simp only*: *stable-simps*)?; (*rule stable.intro*)+; *fail*)
apply (*rule ag.prog.bind*)
apply (*rule ag.pre-g*[*OF pending.ag.enq pending-g*])
apply ((*simp only*: *stable-simps*)?; (*rule stable.intro*)+; *fail*)
apply (*rule ag.pre-g*[*OF set.ag.ins set-g*])
apply ((*simp only*: *stable-simps*)?; (*rule stable.intro*)+; *fail*)
apply (*rule ag.pre-pre*[*OF ag.pre-g*[*OF set.ag.mem set-g*]])
apply ((*simp only*: *stable-simps*)?; (*rule stable.intro*)+; *fail*)
apply (*force simp*: *aloop-invP-step vclosureP-step*)
apply ((*simp only*: *stable-simps*)?; (*rule stable.intro*)+; *fail*)

apply (*rule ag.pre-g*[*OF pending.ag.deq pending-g*])
apply ((*simp only*: *stable-simps*)*?*; (*rule stable.intro*)+; *fail*)
apply (*auto elim*: *search-postcond-Some search-postcond-None aloop-invP-dequeue*
aloop-invPD vclosureP-dequeue)
done

lemma *ag-amain*:
shows *prog.p2s* (*amain f P* x_0) ≤ ⦃⟨*True*⟩⦄, *Id* ×$_R$ *Id* ×$_R$ *UNIV* ⊢ *UNIV* ×$_R$ *UNIV* ×$_R$ *Id*, ⦃*search-post*⦄
unfolding *main-def*
apply (*rule ag.pre-pre*)
apply (*rule ag.prog.bind*)+
apply (*rule ag-aloop*)
apply (*rule ag.post-imp*[**where** *Q*=⟨λ(*q*, *v*, -). *q* = {x_0} ∧ *v* = [x_0]⟩])
apply (*clarsimp simp*: *aloop-invP-init vclosureP-init*; *fail*)
apply (*rule ag.pre-g*[*OF pending.ag.enq pending-g*])
apply ((*simp only*: *stable-simps*)*?*; (*rule stable.intro*)+; *fail*)
apply (*rule ag.pre-g*[*OF set.ag.ins set-g*])
apply ((*simp only*: *stable-simps*)*?*; (*rule stable.intro*)+; *fail*)
apply (*rule ag.pre-g*[*OF pending.ag.init pending-g*])
apply ((*simp only*: *stable-simps*)*?*; (*rule stable.intro*)+; *fail*)
apply (*rule ag.pre-g*[*OF set.ag.init set-g*])
apply ((*simp only*: *stable-simps*)*?*; (*rule stable.intro*)+; *fail*)
apply *simp*
done

lemma *ag-asearch*:
shows *prog.p2s* (*asearch f P* x_0 :: ('*s*, '*a option*) *prog*) ≤ ⦃⟨*True*⟩⦄, *UNIV* ⊢ *Id*, ⦃*search-post*⦄
unfolding *search-def* **by** (*rule ag.prog.local ag-amain*)+

Refinement **abbreviation** *A* ≡ *ag.assm* (*Id* ×$_R$ *Id* ×$_R$ *UNIV*)
abbreviation *absfn c* ≡ *prog.sinvmap* (*map-prod pending.absfn id*) *c*
abbreviation *p2s-absfn c* ≡ *prog.p2s* (*absfn c*)

— visited set: reflexive
lemma *ref-set-init*:
shows *prog.p2s* (*set.init set.abstract*) ≤ ⦃λ*s*. *True*⦄, *A* ⊨ *p2s-absfn* (*set.init set.abstract*), ⦃λ*v s*. *True*⦄
by (*auto simp*: *set.abstract-def intro*: *rair.prog.action stable.intro*)

lemma *ref-set-ins*:
shows *prog.p2s* (*set.ins set.abstract x*) ≤ ⦃λ*s*. *True*⦄, *A* ⊨ *p2s-absfn* (*set.ins set.abstract x*), ⦃λ*v s*. *True*⦄
by (*auto simp*: *set.abstract-def intro*: *rair.prog.action stable.intro*)

lemma *ref-set-mem*:
shows *prog.p2s* (*set.mem set.abstract x*) ≤ ⦃λ*s*. *True*⦄, *A* ⊨ *p2s-absfn* (*set.mem set.abstract x*), ⦃λ*v s*. *True*⦄
by (*auto simp*: *set.abstract-def intro*: *rair.prog.action stable.intro*)

— queue
lemma *ref-queue-init*:
shows *prog.p2s* (*pending.init pending.concrete*) ≤ ⦃λ*s*. *True*⦄, *A* ⊨ *p2s-absfn* (*pending.init pending.abstract*), ⦃λ*v s*. *True*⦄
by (*auto simp*: *pending.abstract-def pending.concrete-def pending.absfn-def intro*: *rair.prog.action stable.intro*)

lemma *ref-queue-enq*:
shows *prog.p2s* (*pending.enq pending.concrete x*) ≤ ⦃λ*s*. *True*⦄, *A* ⊨ *p2s-absfn* (*pending.enq pending.abstract x*), ⦃λ*v s*. *True*⦄
by (*auto simp*: *pending.abstract-def pending.concrete-def pending.absfn-def intro*: *rair.prog.action stable.intro*)

lemma *ref-queue-deq*:

shows *prog.p2s* (*pending.deq pending.concrete*) ≤ ⦃λ*s*. *True*⦄, *A* ⊨ *p2s-absfn* (*pending.deq pending.abstract*), ⦃λ*v s*. *True*⦄
by (*auto simp*: *pending.abstract-def pending.concrete-def pending.absfn-def*
intro: *rair.prog.action stable.intro elim*!: *pending.cdeq-update.elims*[*OF sym*])

— program
lemma *ref-bfs-loop*:
shows *prog.p2s* (*loop pending.concrete f P*) ≤ ⦃λ*s*. *True*⦄, *A* ⊨ *p2s-absfn* (*loop pending.abstract f P*), ⦃λ*v s*. *True*⦄
apply (*simp add*: *loop-def*)
apply (*rule rair.prog.while*)
apply (*rule rair.prog.rev-bind*)
apply (*rule ref-queue-deq*)
apply (*rule refinement.pre-pre*[*OF rair.prog.case-option*])
apply (*rule rair.prog.return*)
apply ((*simp only*: *stable-simps*)?; (*rule stable.intro*)+; *fail*)
apply (*rule rair.prog.if*)
apply (*rule rair.prog.return*)
apply ((*simp only*: *stable-simps*)?; (*rule stable.intro*)+; *fail*)
apply (*rule rair.prog.rev-bind*)
apply (*rule rair.prog.app*)
apply (*rule rair.prog.rev-bind*)
apply (*rule ref-set-mem*)
apply (*rule refinement.pre-pre*[*OF rair.prog.if*])
apply (*rule rair.prog.return*)
apply ((*simp only*: *stable-simps*)?; (*rule stable.intro*)+; *fail*)
apply (*rule rair.prog.rev-bind*)
apply (*rule ref-set-ins*)
apply (*rule ref-queue-enq*)
apply (*simp*; *fail*)
apply ((*simp only*: *stable-simps*)?; (*rule stable.intro*)+; *fail*)
apply (*rule refinement.pre-pre*[*OF rair.prog.return*])
apply ((*simp only*: *stable-simps*)?; (*rule stable.intro*)+; *fail*)
apply (*auto intro*: *stable.intro split*: *option.split*)
done

lemma *ref-bfs-main*:
shows *prog.p2s* (*main pending.concrete f P x*) ≤ ⦃⟨*True*⟩⦄, *A* ⊨ *p2s-absfn* (*amain f P x*), ⦃λ*v s*. *True*⦄
apply (*simp add*: *main-def*)
apply (*rule rair.prog.rev-bind*)
apply (*rule ref-set-init*)
apply (*rule rair.prog.rev-bind*)
apply (*rule ref-queue-init*)
apply (*rule rair.prog.rev-bind*)
apply (*rule ref-set-ins*)
apply (*rule rair.prog.rev-bind*)
apply (*rule ref-queue-enq*)
apply (*rule ref-bfs-loop*)
done

theorem *ref-bfs*:
shows *bfs f P x* ≤ *asearch f P x*
unfolding *search-def*
apply (*intro refinement.prog.leI refinement.prog.data*[**where** *sf*=*id*])
apply (*simp add*: *spec.invmap.id spec.localizeA.top*)
apply (*rule refinement.prog.data*[**where** *sf*=*pending.absfn*])
apply (*simp flip*: *prog.p2s.invmap*)
apply (*rule refinement.pre-a*[*OF ref-bfs-main*])

apply (*auto simp*: *spec.localizeA-def spec.invmap.rel*
simp flip: *spec.rel.inf*
intro: *spec.rel.mono*)
done

theorem *bfs-post-le*:
shows *prog.p2s* (*bfs f P* x_0) ≤ *spec.post* (*search-post*)
apply (*strengthen ord-to-strengthen*[*OF ref-bfs*])
apply (*strengthen ord-to-strengthen*(*1*)[*OF ag-asearch*])
apply (*simp add*: *ag-def spec.rel.UNIV flip*: *Sigma-Un-distrib1*)
done

end

24 Observations about safety closure

We demonstrate that *Sup* does not distribute in ('*a*, '*s*, '*v*) *tls* as it does in ('*a*, '*s*, '*v*) *spec*: specifically a *Sup* of a set of safety properties in the former need not be a safety property, whereas in the latter it is (see §8.2).

corec *bnats* :: *nat* ⇒ ('*a* × *nat*, '*v*) *tllist* **where**
bnats i = *TCons* (*undefined*, *i*) (*bnats* (*Suc i*))

definition *bnat* :: ('*a*, *nat*, '*v*) *behavior.t* **where**
bnat = *behavior.B 0* (*bnats 1*)

definition *tnats* :: *nat* ⇒ *nat* ⇒ ('*a* × *nat*) *list* **where**
tnats i j = *map* (*Pair undefined*) (*upt i j*)

definition *tnat* :: *nat* ⇒ ('*a*, *nat*, '*v*) *trace.t* **where**
tnat i = *trace.T 0* (*tnats* (*Suc 0*) (*Suc i*)) *None*

lemma *tnat-simps*[*simp*]:
shows *tnats i i* = []
and *trace.init* (*tnat i*) = *0*
and *trace.rest* (*tnat i*) = *tnats 1* (*Suc i*)
and *length* (*tnats i j*) = *j* − *i*
by (*simp-all add*: *tnats-def tnat-def*)

lemma *take-tnats*:
shows *take i* (*tnats j k*) = *tnats j* (*min* (*i* + *j*) *k*)
by (*simp add*: *tnats-def take-map add.commute split*: *split-min*)

lemma *take-tnat*:
shows *trace.take i* (*tnat j*) = *tnat* (*min i j*)
by (*simp add*: *trace.take-def take-tnats tnat-def*)

lemma *mono-tnat*:
shows *mono tnat*
by (*rule monoI*) (*auto simp*: *trace.less-eq-take-def take-tnat split*: *split-min*)

lemma *final′-tnats*:
shows *trace.final′ i* (*tnats j k*) = (*if j* < *k then k* − *1 else i*)
by (*simp add*: *tnats-def trace.final′-def comp-def*)

lemma *sset-tnat*:
shows *trace.sset* (*tnat i*) = {*j*. *j* ≤ *i*}
by (*force simp*: *tnat-def tnats-def trace.sset.simps*)

lemma *natural′-tnats*:
  shows *trace.natural′ i (tnats (Suc i) (Suc j)) = tnats (Suc i) (Suc j)*
proof −
  have *trace.natural′ i (map (Pair undefined) (upt (Suc i) (Suc j)))*
    = map (Pair undefined) (upt (Suc i) (Suc j)) **for** *j*
    by (*induct j arbitrary*: *i*) (*simp-all add*: *trace.natural′.append*)
  from *this* **show** *?thesis* **unfolding** *tnats-def* **.**
qed

lemma *natural-tnat*:
  shows ♮(*tnat i*) = *tnat i*
by (*simp add*: *tnat-def trace.natural-def natural′-tnats del*: *upt-Suc*)

lemma *ttake-bnats*:
  shows *ttake i* (*bnats j*) = (*tnats j* (*i* + *j*), *None*)
by (*induct i arbitrary*: *j*) (*subst bnats.code*; *simp add*: *tnats-def upt-rec*)+

lemma *take-bnat-tnat*:
  shows *behavior.take i bnat* = *tnat i*
by (*simp add*: *bnat-def tnat-def behavior.take-def ttake-bnats*)

unbundle *tls.extra-syntax*

definition *bnat-approx* :: (*unit*, *nat*, *unit*) *spec set* **where**
  bnat-approx = {⦇*behavior.take i bnat*⦈ |*i*. *True*}

lemma *bnat-approx-alt-def*:
  shows *bnat-approx* = {⦇*tnat i*⦈ |*i*. *True*}
by (*simp add*: *bnat-approx-def take-bnat-tnat*)

lemma *not-tls-from-spec-Sup-distrib*: — *tls.from-spec* is not *Sup*-distributive
  shows ¬*tls.from-spec* (⨆ *bnat-approx*) ≤ ⨆ (*tls.from-spec* ‘ *bnat-approx*) (**is** ¬ *?lhs* ≤ *?rhs*)
proof −
  have ⦇*bnat*⦈$_T$ ≤ *?lhs*
  proof −
    have ∗: ∃*j*. *behavior.take i ω* ∈ *raw.spec.cl* {*behavior.take j bnat*} **if** *bnat* ≃$_T$ *ω*
      for *i* **and** *ω* :: (′*a*, *nat*, ′*b*) *behavior.t*
      by (*rule behavior.stuttering.equiv.takeE*[*OF sym*[*OF that*], **where** *i*=*i*])
        (*force simp*: *raw.spec.cl-def simp flip*: *trace.stuttering.cl.downwards.cl*)
    note *spec.singleton.transfer*[*transfer-rule*]
    show *?thesis*
      unfolding *bnat-approx-def*
      by *transfer*
        (*force dest*: ∗ *simp*: *TLS.raw.singleton-def raw.from-spec-def Safety-Logic.raw.singleton-def*
          simp flip: *ex-simps elim*!: *behavior.stuttering.clE*)
  qed
  moreover
  have ¬(∀*j*. *tnat j* ≤ *tnat i*) **for** *i*
    by (*auto intro*!: *exI*[**where** *x*=*Suc i*] *dest*!: *monoD*[*OF trace.sset.mono*] *simp*: *sset-tnat*)
  then have ¬⦇*bnat*⦈$_T$ ≤ *?rhs*
    by (*fastforce simp*: *bnat-approx-def tls.singleton.le-conv spec.singleton-le-conv take-bnat-tnat natural-tnat*)
  ultimately show *?thesis*
    by (*blast dest*: *order.trans*)
qed

definition *bnat′* :: (*unit*, *nat*, *unit*) *tls set* **where**
  bnat′ = *tls.from-spec* ‘ *bnat-approx*

lemma *not-tls-safety-cl-Sup-distrib*: — *tls.safety.cl* is not *Sup*-distributive
shows $\neg$*tls.safety.cl* ($\bigsqcup$ *bnat'*) $\leq$ $\bigsqcup$ (*tls.safety.cl* ‘ *bnat'*)
proof −
have ($\bigsqcup x{\in}$*bnat-approx*. *tls.to-spec* (*tls.from-spec x*)) = $\bigsqcup$ *bnat-approx* (**is** *?lhs* = *?rhs*)
proof(*rule antisym*)
show *?lhs* $\leq$ *?rhs*
by (*simp add*: *Sup-upper2 tls.safety.lower-upper-contractive*)
have $\exists\omega$ *ia ib*. ($\forall$ *i*. ♮ (*behavior.take i* ω) $\leq$ *tnat ib*) $\wedge$ *tnat i* = *behavior.take ia* ω
for *i*
by (*rule exI*[**where** *x*=*behavior.B 0* (*tshift2* (*tnats* (*Suc 0*) (*Suc i*), *None*) (*trepeat* (*undefined*, *i*)))])
(*force simp*: *behavior.take.tshift ttake-trepeat trace.take.continue take-tnat*
trace.natural.continue trace.natural'.replicate natural-tnat not-le final'-tnats
simp flip: *tnat-def*
split: *split-min*
intro: *monoD*[*OF mono-tnat*])
then show *?rhs* $\leq$ *?lhs*
by (*clarsimp simp*: *bnat-approx-alt-def spec.singleton.le-conv tls.singleton.le-conv*
spec.singleton-le-conv natural-tnat
simp flip: *ex-simps*;
fast)
qed
then show *?thesis*
by (*simp add*: *bnat'-def tls.safety.cl-def tls.safety.upper-lower-upper tls.to-spec.Sup*
not-tls-from-spec-Sup-distrib image-image)
qed

definition *cl-bnat'* :: (*unit*, *nat*, *unit*) *tls set* **where**
cl-bnat' = *tls.safety.cl* ‘ *bnat'*

lemma *not-tls-safety-closed-Sup*:
shows *cl-bnat'* $\subseteq$ *tls.safety.closed*
and $\bigsqcup$ *cl-bnat'* $\notin$ *tls.safety.closed*
unfolding *cl-bnat'-def*
using *not-tls-safety-cl-Sup-distrib*
by (*blast intro*: *tls.safety.expansive complete-lattice-class.Sup-mono*
dest: *tls.safety.least*[*rotated*, **where** *x*=$\bigsqcup$ *bnat'*])+

Negation does not preserve *tls.safety.closed* **notepad**
begin

have *tls.always* (*tls.state-prop id*) $\in$ *tls.safety.closed*
by (*simp add*: *tls.safety.closed.always tls.safety.closed.state-prop*)

have −*tls.always* (*tls.state-prop id*) $\notin$ *tls.safety.closed*
proof −
let *?P* = *behavior.B True* (*trepeat* (*undefined*, *True*)) :: (*'a*, *bool*, *'c*) *behavior.t*
have $\exists\omega'$. *behavior.take i ?P* = *behavior.take i* ω'
$\wedge$ ($\exists$ *j* ω''. *behavior.dropn j* ω' = *Some* ω'' $\wedge$ $\neg$ *behavior.init* ω'')
for *i*
by (*auto simp*: *behavior.dropn.continue behavior.take.continue behavior.take.trepeat*
trace.take.replicate case-tllist-trepeat
exI[**where** *x*=*behavior.take i ?P* @−$_B$ *trepeat* (*undefined*, *False*)]
exI[**where** *x*=*Suc i*])
then have ⦇*?P*⦈$_T$ $\leq$ *tls.safety.cl* (−*tls.always* (*tls.state-prop id*))
by (*clarsimp simp*: *tls.singleton.le-conv*; *fast*)
moreover
have *behavior.init* ω'
if *behavior.dropn i* (*behavior.B True* (*trepeat* (*undefined*, *True*))) = *Some* ω'

for *i* **and** ω' :: (*'a*, *bool*, *'c*) *behavior.t*
using *that*
by (*cases i*) (*auto simp*: *behavior.dropn-alt-def tdropn-trepeat case-tllist-trepeat*)
then have $\neg \langle\!| ?P |\!\rangle_T \leq -tls.always\ (tls.state\text{-}prop\ id)$
by (*auto simp*: *tls.singleton.le-conv*)
ultimately show *?thesis*
using *tls.safety.le-closedE* **by** *blast*
qed

end

24.1 Liveness

Famously arbitrary properties on infinite sequences can be decomposed into *safety* and *liveness* properties. The latter have been identified with the topologically dense sets.
References:

- Alpern and Schneider (1985); Schneider (1987): topological account
- Kindler (1994): overview
- Lynch (1996, §8.5.3)
- Manolios and Trefler (2003): lattice-theoretic account
- Maier (2004): an intuitionistic semantics for LTL (including next/X/◎) over finite and infinite sequences

setup ‹*Sign.mandatory-path raw.safety*›

lemma *dense-alt-def*: — Lynch (1996, §8.5.3 Liveness Property)
shows (*raw.safety.dense* :: (*'a*, *'s*, *'v*) *behavior.t set set*)
$= \{P.\ \forall \sigma.\ \exists xsv.\ \sigma\ @-_B\ xsv \in P\}$ (**is** *?lhs* = *?rhs*)
proof(*rule antisym*)
have $\exists xsv.\ \sigma\ @-_B\ xsv \in P$ **if** *raw.safety.cl P* = *UNIV* **for** *P* **and** σ :: (*'a*, *'s*, *'v*) *trace.t*
using *that*
by (*auto simp*: *behavior.take.continue*
simp flip: *trace.take.Ex-all*
elim!: *raw.safety.cl-altE*[**where** *i*=*Suc* (*length* (*trace.rest* σ))]
dest!: *subsetD*[**where** *c*=$\sigma\ @-_B$ *TNil undefined*, *OF Set.equalityD2*])
(*metis behavior.continue.take-drop-id behavior.continue.tshift2*)
then show *?lhs* ⊆ *?rhs*
by (*clarsimp simp*: *raw.safety.dense-def*)
next
have $\omega \in$ *raw.safety.cl P* **if** $\forall \sigma.\ \exists xsv.\ \sigma\ @-_B\ xsv \in P$ **for** *P* **and** ω :: (*'a*, *'s*, *'v*) *behavior.t*
proof(*rule raw.safety.cl-altI*)
fix *i*
from *spec*[*OF that*, **where** *x*=*behavior.take i* ω]
obtain *xsv* **where** *behavior.take i* $\omega\ @-_B$ *xsv* ∈ *P* **..**
moreover
have *behavior.take i* ω = *behavior.take i* (*behavior.take i* $\omega\ @-_B$ *xsv*)
by (*clarsimp simp*: *behavior.take.continue behavior.take.all-continue*
trace.take.behavior.take length-ttake not-le
split: *enat.split split-min*)
ultimately show $\exists \omega' \in P.$ *behavior.take i* ω = *behavior.take i* ω' **..**
qed
then show *?rhs* ⊆ *?lhs*
by (*auto simp*: *raw.safety.dense-def*)
qed

setup ‹*Sign.parent-path*›

setup ‹*Sign.mandatory-path tls*›

definition *live* :: ('*a*, '*s*, '*v*) *tls set* **where**
  live = *tls.safety.dense*

setup ‹*Sign.mandatory-path live*›

lemma *not-bot*:
  shows ⊥ ∉ *tls.live*
by (*simp add*: *tls.live-def tls.safety.dense-def tls.bot-not-top*)

lemma *top*:
  shows ⊤ ∈ *tls.live*
by (*simp add*: *tls.live-def tls.safety.dense-top*)

lemma *live-le*:
  assumes *P* ∈ *tls.live*
  assumes *P* ≤ *Q*
  shows *Q* ∈ *tls.live*
using *assms* **by** (*simp add*: *tls.live-def tls.safety.dense-le*)

lemma *inf-safety-eq-top*: — Lynch (1996, Theorem 8.8)
  shows *tls.live* ⊓ *tls.safety.closed* = {⊤}
unfolding *tls.live-def* **by** (*rule tls.safety.dense-inf-closed*)

lemma *terminating*:
  shows *tls.eventually tls.terminated* ∈ *tls.live*
by (*simp add*: *tls.live-def tls.safety.dense-def tls.safety.cl.eventually*[*OF tls.terminated.not-bot*])

However this definition of liveness does not endorse traditional *response* properties.

corec *alternating* :: *bool* ⇒ ('*a* × *bool*, '*b*) *tllist* **where**
  alternating b = *TCons* (*undefined*, *b*) (*alternating* (¬*b*))

abbreviation (*input*) *A b* ≡ *behavior.B b* (*tls.live.alternating* (¬*b*))

lemma *dropn-alternating*:
  shows *behavior.dropn i* (*tls.live.A b*) = *Some* (*tls.live.A* (*if even i then b else* ¬*b*))
proof(*induct i arbitrary*: *b*)
  case (*Suc i*) **show** *?case*
    by (*subst tls.live.alternating.code*) (*simp add*: *behavior.dropn.Suc Suc*[*of* ¬*b*, *simplified*])
qed *simp*

notepad
begin

let *?P* = *tls.leads-to* (*tls.state-prop id*) (*tls.state-prop Not*) :: ('*a*, *bool*, *unit*) *tls*
let *?ω* = ⟪*behavior.B True* (*TNil* ())⟫$_T$ :: ('*a*, *bool*, *unit*) *tls*

have ¬*?ω* ≤ *?P*
  by (*auto simp*: *tls.singleton.le-conv split*: *nat.split*)
then have ¬*?ω* ≤ *tls.safety.cl ?P*
  by (*simp add*: *tls.safety.cl.terminated-iff tls.singleton.terminated-le-conv behavior.sset.simps*)
then have *?P* ∉ *tls.live*
  by (*auto simp*: *tls.live-def tls.safety.dense-def*
      dest: *order.trans*[**where** *a*=*?ω*, *OF top-greatest eq-refl*[*OF sym*]])

— non-triviality
let *?ω′ = ⦇tls.live.A True⦈$_T$ :: ('a, bool, unit) tls*
have *?ω′ ≤ ?P*
  by (*clarsimp simp*: *tls.singleton.le-conv tls.live.dropn-alternating*[**where** *b=True*, *simplified*]) *presburger*

— intuition: there's some safety in these response properties
let *?Q = tls.always* (*tls.terminated ⟶$_B$ tls.state-prop Not*) :: ('a, bool, unit) tls
have *?Q ∈ tls.safety.closed*
  by (*simp add*: *tls.safety.closed.always tls.safety.closed.boolean-implication*
      tls.safety.closed.not-terminated tls.safety.closed.state-prop)
moreover have *¬?ω ≤ ?Q*
  by (*auto simp*: *tls.singleton.le-conv behavior.sset.simps split*: *nat.split*)
then have *?Q ≠ ⊤*
  by (*auto dest*: *order.trans*[**where** *a=?ω*, *OF top-greatest eq-refl*[*OF sym*]])
ultimately have *?Q ∉ tls.live*
  using *tls.live.inf-safety-eq-top* **by** *auto*
moreover
  have *tls.terminated ⊓ (tls.state-prop id ⟶$_B$ tls.eventually* (*tls.state-prop Not*)) *≤ tls.state-prop Not*
    by (*simp add*: *boolean-implication.conv-sup inf-sup-distrib tls.state-prop.simps tls.terminated.inf-eventually*)
  then have *?P ≤ ?Q*
    by − (*rule tls.always.mono*;
      simp add: *tls.terminated.inf-always flip*: *boolean-implication.shunt2*)
ultimately have *?P ∉ tls.live*
  by (*blast dest*: *tls.live.live-le*)

end

setup ‹*Sign.parent-path*›

The famous decomposition **definition** *Safe* :: ('a, 's, 'v) *tls* ⇒ ('a, 's, 'v) *tls* **where**
  Safe P = tls.safety.cl P

definition *Live* :: ('a, 's, 'v) *tls* ⇒ ('a, 's, 'v) *tls* **where**
  Live P = P ⊔ −tls.safety.cl P

lemma *decomp*:
  shows *P = tls.Safe P ⊓ tls.Live P*
by (*simp add*: *tls.Safe-def tls.Live-def boolean-algebra.conj-disj-distrib inf.absorb2 tls.safety.expansive*)

setup ‹*Sign.mandatory-path safety.closed*›

lemma *Safe*:
  shows *tls.Safe P ∈ tls.safety.closed*
by (*simp add*: *tls.Safe-def*)

setup ‹*Sign.parent-path*›

setup ‹*Sign.mandatory-path live*›

lemma *Live*:
  shows *tls.Live P ∈ tls.live*
by (*simp add*: *tls.live-def tls.safety.dense-def tls.Live-def sup-shunt tls.safety.cl.sup tls.safety.expansive*)

setup ‹*Sign.parent-path*›

setup ‹*Sign.parent-path*›

24.2 A Haskell-like *Ix* class

We allow arbitrary indexing schemes for user-facing arrays via the *Ix* class, which essentially represents a bijection between a subset of an arbitrary type and an initial segment of the naturals.
Source materials:

- Haskell 2010 report: https://www.haskell.org/onlinereport/haskell2010/haskellch19.html
- GHC implementation: https://hackage.haskell.org/package/base-4.16.0.0/docs/src/GHC.Ix.html
- Haskell pure arrays (just for colour): https://www.haskell.org/onlinereport/haskell2010/haskellch14.html
- SML 2D arrays: https://smlfamily.github.io/Basis/array2.html

Observations:

- follow Haskell convention here: include the bounds
- could alternatively use an array of one-dimensional arrays but those are not necessarily rectangular
- we can't use *enum* as that requires the whole type to be enumerable

Limitations:

- the basic design assumes laziness; we don't ever want to build the list of indices
  - can be improved either by tweaking the code generator setup or changing the constants here
- array indices typically have partial predecessor and successor operations and are totally ordered on their domain
- no guarantee the *interval* is correct (does not prevent off-by-one errors in instances)

class *Ix* =
  fixes *interval* :: $'a \times 'a \Rightarrow 'a\ list$
  fixes *index* :: $'a \times 'a \Rightarrow 'a \Rightarrow nat$
  assumes *index*: $i \in set\ (interval\ b) \Longrightarrow interval\ b\ !\ index\ b\ i = i$
  assumes *interval*: $map\ (index\ b)\ (interval\ b) = [0..<length\ (interval\ b)]$

lemma *index-length*:
  assumes $i \in set\ (interval\ b)$
  shows $index\ b\ i < length\ (interval\ b)$
proof −
  from *assms*[*unfolded in-set-conv-nth*]
  obtain j **where** $j < length\ (interval\ b)$ **and** $interval\ b\ !\ j = i$
    by *blast*
  with *arg-cong*[**where** $f=\lambda x.\ List.nth\ x\ j$, *OF interval*[*of b*]] **show** *?thesis*
    by *simp*
qed

lemma *distinct-interval*:
  shows *distinct* (*interval b*)
by (*metis distinct-map distinct-upt interval*)

lemma *inj-on-index*:
  shows *inj-on* (*index b*) (*set* (*interval b*))
by (*metis distinct-map distinct-upt interval*)

lemma *index-eq-conv*:
  assumes $i \in set\ (interval\ b)$

assumes $j \in set\ (interval\ b)$
shows $index\ b\ i = index\ b\ j \longleftrightarrow i = j$
by (*metis assms index*)

lemma *index-inv-into*:
assumes $i < length\ (interval\ b)$
shows $inv\text{-}into\ (set\ (interval\ b))\ (index\ b)\ i \in set\ (interval\ b)$
by (*metis assms add.left-neutral inv-into-into length-map list.set-map interval nth-mem nth-upt*)

lemma *linear-order-on*:
shows $linear\text{-}order\text{-}on\ (set\ (interval\ b))\ \{(i, j).\ \{i, j\} \subseteq set\ (interval\ b) \wedge index\ b\ i \leq index\ b\ j\}$
by (*force simp*: *linear-order-on-def partial-order-on-def preorder-on-def refl-on-def total-on-def*
intro: *transI antisymI*
dest: *index*)

lemma *interval-map*:
shows $map\ (\lambda i.\ f\ (interval\ b\ !\ i))\ [0..<length\ (interval\ b)] = map\ f\ (interval\ b)$
by (*simp add*: *map-equality-iff*)

lemma *index-forE*:
assumes $i < length\ (interval\ b)$
obtains j **where** $j \in set\ (interval\ b)$ **and** $index\ b\ j = i$
using *assms index index-length nth-eq-iff-index-eq*[*OF distinct-interval*] *nth-mem*[*OF assms*] **by** *blast*

instantiation *unit* :: *Ix*
begin

definition $interval\text{-}unit = (\lambda(x{::}unit,\ y{::}unit).\ [()])$
definition $index\text{-}unit = (\lambda(x{::}unit,\ y{::}unit)\ \text{-}{::}unit.\ 0{::}nat)$

instance by *standard* (*auto simp*: *interval-unit-def index-unit-def*)

end

instantiation *nat* :: *Ix*
begin

definition $interval\text{-}nat = (\lambda(l,\ u{::}nat).\ [l..<Suc\ u])$ — bounds are inclusive
definition $index\text{-}nat = (\lambda(l,\ u{::}nat)\ i{::}nat.\ i - l)$

lemma *upt-minus*:
shows $map\ (\lambda i.\ i - l)\ [l..<u] = [0..<u - l]$
by (*induct u*) (*auto simp*: *Suc-diff-le*)

instance by *standard* (*auto simp*: *interval-nat-def index-nat-def upt-minus nth-append*)

end

instantiation *int* :: *Ix*
begin

definition $interval\text{-}int = (\lambda(l,\ u{::}int).\ [l..u])$ — bounds are inclusive
definition $index\text{-}int = (\lambda(l,\ u{::}int)\ i{::}int.\ nat\ (i - l))$

lemma *upto-minus*:
shows $map\ (\lambda i.\ nat\ (i - l))\ [l..u] = [0..<nat\ (u - l + 1)]$
proof(*induct nat*$(u - l + 1)$ *arbitrary*: *u*)
case (*Suc i*)

from *Suc.hyps*(*1*)[*of* $u - 1$] *Suc.hyps*(*2*) **show** *?case*
by (*simp add*: *upto-rec2 ac-simps Suc-nat-eq-nat-zadd1 flip*: *upt-Suc-append*)
qed *simp*

instance by *standard* (*auto simp*: *interval-int-def index-int-def upto-minus*)

end

type-synonym $('i, 'j)$ *two-dim* $= ('i \times 'j) \times ('i \times 'j)$

instantiation *prod* :: (*Ix*, *Ix*) *Ix*
begin

definition *interval-prod* $= (\lambda((l, l'), (u, u'))$. *List.product* (*interval* (l, u)) (*interval* $(l', u')))$
definition *index-prod* $= (\lambda((l, l'), (u, u'))\ (i, i')$. *index* $(l, u)\ i *$ *length* (*interval* $(l', u')) +$ *index* $(l', u')\ i')$

abbreviation (*input*) *fst-bounds* :: $('a \times 'b) \times ('a \times 'b) \Rightarrow ('a \times 'a)$ **where**
fst-bounds $b \equiv$ (*fst* (*fst* b), *fst* (*snd* b))

abbreviation (*input*) *snd-bounds* :: $('a \times 'b) \times ('a \times 'b) \Rightarrow ('b \times 'b)$ **where**
snd-bounds $b \equiv$ (*snd* (*fst* b), *snd* (*snd* b))

lemma *inj-on-index-prod*:
shows *inj-on* (*index* $((l, l'), (u, u')))$ (*set* (*interval* $((l, l'), (u, u'))))$
by (*clarsimp simp*: *inj-on-def interval-prod-def index-prod-def*)
(*metis index index-length length-pos-if-in-set add-diff-cancel-right'*
div-mult-self-is-m mod-less mod-mult-self3)

instance
proof
show *interval* b ! *index* $b\ i = i$ **if** $i \in$ *set* (*interval* b) **for** b **and** $i :: 'a \times 'b$
proof −
have $*$: $i * n + j < m * n$ **if** $i < m$ **and** $j < n$
for $i\ j\ m\ n :: nat$
using *that* **by** (*metis bot-nat-0.extremum-strict div-less div-less-iff-less-mult div-mult-self3 nat-arith.rule0 not-gr-zero*)
from *that*
have *index* (*fst-bounds* b) (*fst* i) $*$ *length* (*interval* (*snd-bounds* b))
$+$ *index* (*snd-bounds* b) (*snd* i)
$<$ *length* (*interval* (*fst-bounds* b)) $*$ *length* (*interval* (*snd-bounds* b))
by (*clarsimp simp*: *interval-prod-def index-prod-def* $*$ *dest*!: *index-length*)
then show *?thesis*
using *that length-pos-if-in-set*
by (*fastforce simp*: *interval-prod-def index-prod-def List.product-nth index index-length*)
qed
show *map* (*index* b) (*interval* b) $=$ [*0*..<*length* (*interval* b)] **for** $b :: ('a \times 'b) \times ('a \times 'b)$
by (*rule iffD2*[*OF list-eq-iff-nth-eq*])
(*clarsimp simp*: *interval-prod-def index-prod-def split-def product-nth ac-simps*;
metis (*no-types*, *lifting*) *distinct-interval index index-length length-pos-if-in-set nth-mem*
less-mult-imp-div-less mod-div-mult-eq mod-less-divisor mult.commute nth-eq-iff-index-eq)
qed

end

setup ‹*Sign.mandatory-path Ix*›

setup ‹*Sign.mandatory-path prod*›

lemma *interval-conv*:
  shows $(x, y) \in set\ (interval\ b) \longleftrightarrow x \in set\ (interval\ (fst\text{-}bounds\ b)) \wedge y \in set\ (interval\ (snd\text{-}bounds\ b))$
by (*force simp*: *interval-prod-def*)

setup ‹*Sign.parent-path*›

type-synonym $'i\ square = ('i, 'i)\ two\text{-}dim$

definition *square* :: $'i$::*Ix Ix.square* ⇒ *bool* **where**
  $square = (\lambda((l, l'), (u, u')).\ Ix.interval\ (l, u) = Ix.interval\ (l', u'))$

setup ‹*Sign.mandatory-path square*›

lemma *conv*:
  assumes *Ix.square b*
  shows $i \in set\ (Ix.interval\ (fst\text{-}bounds\ b)) \longleftrightarrow i \in set\ (Ix.interval\ (snd\text{-}bounds\ b))$
using *assms* **by** (*clarsimp simp*: *Ix.square-def*)

setup ‹*Sign.parent-path*›

setup ‹*Sign.parent-path*›

hide-const (**open**) *interval index*

25 A polymorphic heap

We model a heap as a partial map from opaque addresses to structured objects.

- we use this extra structure to handle buffered writes (see §27)
- allocation is non-deterministic and partial
- supports explicit deallocation

Limitations:

- does not support polymorphic sum types such as $'a + 'b$ and $'a\ option$ or products or lists
- the class of representable types is too small to represent processes

Source materials:

- $ISABELLE_HOME/src/HOL/Imperative_HOL/Heap.thy
  - that model of heaps includes a *lim* field to support deterministic allocation
  - it uses distinct heaps for arrays and references

setup ‹*Sign.mandatory-path heap*›

type-synonym *addr* = *nat* — untyped heap addresses

datatype *rep* — the concrete representation of heap values
  = *Addr nat heap.addr* — metadata paired with an address
  | *Val nat*

datatype *write* = *Write heap.addr nat heap.rep*

type-synonym *t* = *heap.addr* ⇀ *heap.rep list* — partial map from addresses to structured encoded values

abbreviation *empty* :: *heap.t* **where**
empty ≡ *Map.empty*

primrec *apply-write* :: *heap.write* ⇒ *heap.t* ⇒ *heap.t* **where**
apply-write (*heap.Write addr i x*) *s* = *s*(*addr* ↦ (*the* (*s addr*))[*i* := *x*])

class *rep* = — the class of representable types
assumes *ex-inj*: ∃ *to-heap-rep* :: 'a ⇒ *heap.rep*. *inj to-heap-rep*

setup ‹*Sign.mandatory-path rep*›

lemma *countable-classI*[*intro*!]:
shows *OFCLASS*('a::*countable*, *heap.rep-class*)
by *intro-classes* (*simp add*: *inj-on-def exI*[**where** *x*=*heap.Val* ∘ *to-nat*])

definition *to* :: 'a::*heap.rep* ⇒ *heap.rep* **where**
to = (*SOME f*. *inj f*)

definition *from* :: *heap.rep* ⇒ 'a::*heap.rep* **where**
from = *inv* (*heap.rep.to* :: 'a ⇒ *heap.rep*)

lemmas *inj-to*[*simp*] = *someI-ex*[*OF heap.ex-inj*, *folded heap.rep.to-def*]

lemma *inj-on-to*[*simp*, *intro*]: *inj-on heap.rep.to S*
using *heap.rep.inj-to* **by** (*auto simp*: *inj-on-def*)

lemma *surj-from*[*simp*]: *surj heap.rep.from*
unfolding *heap.rep.from-def* **by** (*simp add*: *inj-imp-surj-inv*)

lemma *to-split*[*simp*]: *heap.rep.to x* = *heap.rep.to y* ⟷ *x* = *y*
using *injD*[*OF heap.rep.inj-to*] **by** *auto*

lemma *from-to*[*simp*]:
shows *heap.rep.from* (*heap.rep.to x*) = *x*
by (*simp add*: *heap.rep.from-def*)

instance *unit* :: *heap.rep* **..**

instance *bool* :: *heap.rep* **..**

instance *nat* :: *heap.rep* **..**

instance *int* :: *heap.rep* **..**

instance *char* :: *heap.rep* **..**

instance *String.literal* :: *heap.rep* **..**

instance *typerep* :: *heap.rep* **..**

setup ‹*Sign.parent-path*›

User-facing heap types typically carry more information than an (untyped) address, such as (phantom) typing and a representation invariant that guarantees the soundness of the encoding (for the given value at the given address only). We abstract over that here and provide some generic operations.

Notes:

- intuitively *addr-of* should be surjective but we do not enforce this
- we use sets here but these are not very flexible: all refs must have the same type
  - this means some intutive facts involving *UNIV* cannot be stated

class *addr-of* =
  fixes *addr-of* :: *'a* ⇒ *heap.addr*
  fixes *rep-val-inv* :: *'a* ⇒ *heap.rep list pred*

definition *obj-at* :: *heap.rep list pred* ⇒ *heap.addr* ⇒ *heap.t pred* **where**
  obj-at P r s = (*case s r of None* ⇒ *False* | *Some v* ⇒ *P v*)

abbreviation (*input*) *present* :: *'a*::*heap.addr-of* ⇒ *heap.t pred* **where**
  present r ≡ *heap.obj-at* ⟨*True*⟩ (*heap.addr-of r*)

abbreviation (*input*) *rep-inv* :: *'a*::*heap.addr-of* ⇒ *heap.t pred* **where**
  rep-inv r ≡ *heap.obj-at* (*heap.rep-val-inv r*) (*heap.addr-of r*)

abbreviation (*input*) *rep-inv-set* :: *'a*::*heap.addr-of* ⇒ *heap.t set* **where**
  rep-inv-set r ≡ *Collect* (*heap.rep-inv r*)

— allows arbitrary transitions provided the *rep-inv* of *r* is respected
abbreviation (*input*) *rep-inv-rel* :: *'a*::*heap.addr-of* ⇒ *heap.t rel* **where**
  rep-inv-rel r ≡ *heap.rep-inv-set r* × *heap.rep-inv-set r*

— totality models the idea that all dereferences are "valid" but only some are reasonable
definition *get* :: *'a*::*heap.addr-of* ⇒ *heap.t* ⇒ *'v*::*heap.rep list* **where**
  get r s = *map heap.rep.from* (*the* (*s* (*heap.addr-of r*)))

definition *set* :: *'a*::*heap.addr-of* ⇒ *'v*::*heap.rep list* ⇒ *heap.t* ⇒ *heap.t* **where**
  set r v s = *s*(*heap.addr-of r* ↦ *map heap.rep.to v*)

definition *dealloc* :: *'a*::*heap.addr-of* ⇒ *heap.t* ⇒ *heap.t* **where**
  dealloc r s = *s* |‘ {*heap.addr-of r*}

— allows no changes to *rs*, asserts the *rep-inv* of *rs* is valid, arbitrary changes to −*rs*
definition *Id-on* :: *'a*::*heap.addr-of set* ⇒ *heap.t rel* (‹*heap.Id*_›) **where**
  heap.Id$_{rs}$ = (⋂ *r*∈*rs*. *heap.rep-inv-rel r* ∩ *Id*$_{\lambda s.\ s\ (heap.addr\text{-}of\ r)}$)

— allows arbitrary changes to *rs* provided the *rep-inv* of *rs* is respected. requires addresses in −*heap.addr-of* ‘ *rs* to be unchanged
definition *modifies* :: *'a*::*heap.addr-of set* ⇒ *heap.t rel* (‹*heap.modifies*_›) **where**
  heap.modifies$_{rs}$ = (⋂ *r*∈*rs*. *heap.rep-inv-rel r*) ∩ {(*s*, *s'*). ∀ *r*∈−*heap.addr-of* ‘ *rs*. *s r* = *s' r*}

setup ‹*Sign.mandatory-path get*›

lemma *cong*:
  assumes *s* (*heap.addr-of r*) = *s'* (*heap.addr-of r'*)
  shows *heap.get r s* = *heap.get r' s'*
by (*simp add*: *assms heap.get-def*)

lemma *Id-on-proj-cong*:
  assumes (*s*, *s'*) ∈ *heap.Id*$_{\{r\}}$
  shows *heap.get r s* = *heap.get r s'*
using *assms* **by** (*simp add*: *heap.Id-on-def heap.get-def*)

lemma *fun-upd*:

shows *heap.get r* (*fun-upd s a* (*Some w*))
= (*if heap.addr-of r* = *a then map heap.rep.from w else heap.get r s*)
by (*simp add*: *heap.get-def*)

lemma *set-eq*:
shows *heap.get r* (*heap.set r v s*) = *v*
by (*simp add*: *heap.get-def heap.set-def comp-def*)

lemma *set-neq*:
assumes *heap.addr-of r* ≠ *heap.addr-of r'*
shows *heap.get r* (*heap.set r' v s*) = *heap.get r s*
by (*simp add*: *heap.get-def heap.set-def assms*)

setup ‹*Sign.parent-path*›

setup ‹*Sign.mandatory-path set*›

lemma *cong*:
assumes *heap.addr-of r* = *heap.addr-of r'*
assumes *v* = *v'*
assumes ⋀*r'*. *r'* ≠ *heap.addr-of r* ⟹ *s r'* = *s' r'*
shows *heap.set r v s* = *heap.set r' v' s'*
by (*simp add*: *assms heap.set-def fun-eq-iff*)

lemma *empty*:
shows *heap.set r v* (*heap.empty*) = [*heap.addr-of r* ↦ *map heap.rep.to v*]
by (*simp add*: *heap.set-def*)

lemma *fun-upd*:
shows *heap.set r v* (*fun-upd s a w*) = (*fun-upd s a w*)(*heap.addr-of r* ↦ *map heap.rep.to v*)
by (*simp add*: *heap.set-def*)

lemma *same*:
shows *heap.set r v* (*heap.set r w s*) = *heap.set r v s*
by (*simp add*: *heap.set-def*)

lemma *twist*:
assumes *heap.addr-of r* ≠ *heap.addr-of r'*
shows *heap.set r v* (*heap.set r' w s*) = *heap.set r' w* (*heap.set r v s*)
using *assms* **by** (*simp add*: *heap.set-def fun-eq-iff*)

setup ‹*Sign.parent-path*›

setup ‹*Sign.mandatory-path obj-at*›

lemma *cong*[*cong*]:
fixes *P* :: *heap.rep list pred*
assumes ⋀*v*. *s r* = *Some v* ⟹ *P v* = *P' v*
assumes *r* = *r'*
assumes *s r* = *s' r'*
shows *heap.obj-at P r s* ⟷ *heap.obj-at P' r' s'*
using *assms* **by** (*simp add*: *heap.obj-at-def cong*: *option.case-cong*)

lemma *split*:
shows *Q* (*heap.obj-at P r s*) ⟷ (*s r* = *None* ⟶ *Q False*) ∧ (∀ *v*. *s r* = *Some v* ⟶ *Q* (*P v*))
by (*simp add*: *heap.obj-at-def split*: *option.splits*)

lemma *split-asm*:

shows Q (*heap.obj-at P r s*) $\longleftrightarrow \neg$ (($s\ r$ = *None* $\wedge \neg Q$ *False*) $\vee$ ($\exists v.\ s\ r$ = *Some* $v \wedge \neg\ Q\ (P\ v)$))
by (*simp add*: *heap.obj-at-def split*: *option.splits*)

lemmas *splits* = *heap.obj-at.split heap.obj-at.split-asm*

lemma *empty*:
shows ¬*heap.obj-at P r heap.empty*
by (*simp add*: *heap.obj-at-def*)

lemma *set*:
shows *heap.obj-at P r* (*heap.set r′ v s*)
$\longleftrightarrow$ (r = *heap.addr-of* $r' \wedge$ *P* (*map heap.rep.to v*)) $\vee$ ($r \neq$ *heap.addr-of* $r' \wedge$ *heap.obj-at P r s*)
by (*simp add*: *comp-def heap.set-def split*: *heap.obj-at.split*)

lemma *fun-upd*:
shows *heap.obj-at P r* (*fun-upd s a* (*Some w*)) = (*if r* = *a then P w else heap.obj-at P r s*)
by (*simp split*: *heap.obj-at.split*)

setup ‹*Sign.parent-path*›

lemmas *simps* = — objective: reduce manifest heaps
heap.get.set-eq
heap.get.fun-upd
heap.set.empty
heap.set.same
heap.set.fun-upd
heap.obj-at.empty
heap.obj-at.fun-upd

setup ‹*Sign.mandatory-path Id-on*›

lemma *empty*[*simp*]:
shows $heap.Id_{\{\}}$ = *UNIV*
by (*simp add*: *heap.Id-on-def*)

lemma *sup*:
shows $heap.Id_{X \cup Y} = heap.Id_X \cap heap.Id_Y$
unfolding *heap.Id-on-def* **by** *blast*

setup ‹*Sign.parent-path*›

setup ‹*Sign.mandatory-path modifies*›

lemma *empty*[*simp*]:
shows $heap.modifies_{\{\}}$ = *Id*
by (*auto simp*: *heap.modifies-def*)

lemma *rep-inv-rel-le*:
shows $heap.modifies_{rs} \subseteq (\bigcap r{\in}rs.\ heap.rep\text{-}inv\text{-}rel\ r)$
by (*simp add*: *heap.modifies-def*)

lemma *rep-inv*:
assumes (s, s') $\in$ $heap.modifies_{\{a\}}$
shows *heap.rep-inv a s*
and *heap.rep-inv a s′*
using *assms* **by** (*simp-all add*: *heap.modifies-def split*: *heap.obj-at.split*)

lemma *Id-conv*:

shows $(s,\ s) \in heap.modifies_{rs} \longleftrightarrow (\forall\, r{\in}rs.\ (s,\ s) \in heap.rep\text{-}inv\text{-}rel\ r)$
by (*simp add*: *heap.modifies-def*)

lemma *eqI*:
assumes $(s,\ s') \in heap.modifies_{rs}$
assumes $\bigwedge r.\ [\![r \in rs;\ heap.rep\text{-}inv\ r\ s;\ heap.rep\text{-}inv\ r\ s']\!] \Longrightarrow s\ (heap.addr\text{-}of\ r) = s'\ (heap.addr\text{-}of\ r)$
shows $s = s'$
using *assms* **by** (*simp add*: *heap.modifies-def*) *blast*

setup ‹*Sign.parent-path*›

setup ‹*Sign.parent-path*›

setup ‹*Sign.mandatory-path stable.heap*›

lemma *Id-on-frame-cong*:
assumes $\bigwedge s\ s'.\ (\bigwedge r.\ r \in rs \Longrightarrow heap.rep\text{-}inv\ r\ s \wedge heap.rep\text{-}inv\ r\ s' \wedge s\ (heap.addr\text{-}of\ r) = s'\ (heap.addr\text{-}of\ r)) \Longrightarrow P\ s \longleftrightarrow P'\ s'$
shows $stable\ heap.Id_{rs}\ P \longleftrightarrow stable\ heap.Id_{rs}\ P'$
using *assms* **by** (*auto 10 0 simp*: *stable-def monotone-def heap.Id-on-def*)

lemma *Id-on-frameI*:
assumes $\bigwedge s\ s'.\ (\bigwedge r.\ r \in rs \Longrightarrow heap.rep\text{-}inv\ r\ s \wedge heap.rep\text{-}inv\ r\ s' \wedge s\ (heap.addr\text{-}of\ r) = s'\ (heap.addr\text{-}of\ r)) \Longrightarrow P\ s \longleftrightarrow P\ s'$
shows $stable\ heap.Id_{rs}\ P$
using *assms* **by** (*auto simp*: *stable-def monotone-def heap.Id-on-def*)

lemma *Id-on-rep-invI*[*stable.intro*]:
assumes $r \in rs$
shows $stable\ heap.Id_{rs}\ (heap.rep\text{-}inv\ r)$
using *assms* **by** (*blast intro*: *stable.heap.Id-on-frameI*)

setup ‹*Sign.parent-path*›

25.1 References

datatype $'a\ ref = Ref\ (addr\text{-}of\text{:}\ heap.addr)$

instantiation *ref* :: (*heap.rep*) *heap.addr-of*
begin

definition *addr-of-ref* :: $'a\ ref \Rightarrow heap.addr$ **where**
addr-of-ref = *ref.addr-of*

definition *rep-val-inv-ref* :: $'a\ ref \Rightarrow heap.rep\ list\ pred$ **where**
rep-val-inv-ref r vs $\longleftrightarrow$ (*case vs of* [*v*] $\Rightarrow$ *heap.rep.to* (*heap.rep.from v* :: $'a$) = *v* | -$\Rightarrow$ *False*)

instance ..

end

instance *ref* :: (*heap.rep*) *heap.rep*
by *standard* (*simp add*: *inj-on-def ref.expand exI*[**where** *x*=*heap.Addr 0* $\circ$ *ref.addr-of*])

setup ‹*Sign.mandatory-path Ref*›

definition *get* :: $'a$::*heap.rep ref* $\Rightarrow$ *heap.t* $\Rightarrow$ $'a$ **where**
get r s = *hd* (*heap.get r s*)

definition *set* :: *'a::heap.rep ref* ⇒ *'a* ⇒ *heap.t* ⇒ *heap.t* **where**
set r v s = *heap.set r* [*v*] *s*

definition *alloc* :: *'a* ⇒ *heap.t* ⇒ (*'a::heap.rep ref* × *heap.t*) *set* **where**
alloc v s = {(*r*, *Ref.set r v s*) |*r*. ¬*heap.present r s*}

lemma *addr-of*:
shows *heap.addr-of* (*Ref r*) = *r*
by (*simp add*: *addr-of-ref-def*)

setup ‹*Sign.mandatory-path get*›

lemma *fun-upd*:
shows *Ref.get r* (*fun-upd s a* (*Some* [*w*]))
= (*if heap.addr-of r* = *a then heap.rep.from w else Ref.get r s*)
by (*simp add*: *Ref.get-def heap.simps*)

lemma *set-eq*:
shows *Ref.get r* (*Ref.set r v s*) = *v*
by (*simp add*: *Ref.get-def Ref.set-def heap.simps*)

lemma *set-neq*:
fixes *r* :: *'a::heap.rep ref*
fixes *r'* :: *'b::heap.rep ref*
assumes *addr-of r* ≠ *addr-of r'*
shows *Ref.get r* (*Ref.set r' v s*) = *Ref.get r s*
using *assms* **by** (*simp add*: *Ref.get-def Ref.set-def addr-of-ref-def heap.get.set-neq*)

setup ‹*Sign.parent-path*›

setup ‹*Sign.mandatory-path set*›

lemma *empty*:
shows *Ref.set r v* (*heap.empty*) = [*heap.addr-of r* ↦ [*heap.rep.to v*]]
by (*simp add*: *Ref.set-def heap.simps*)

lemma *fun-upd*:
shows *Ref.set r v* (*fun-upd s a w*) = (*fun-upd s a w*)(*heap.addr-of r* ↦ [*heap.rep.to v*])
by (*simp add*: *Ref.set-def heap.simps*)

lemma *same*:
shows *Ref.set r v* (*Ref.set r w s*) = *Ref.set r v s*
by (*simp add*: *Ref.set-def heap.set-def*)

lemma *obj-at-conv*:
fixes *a* :: *heap.addr*
fixes *r* :: *'a::heap.rep ref*
fixes *v* :: *'a*
fixes *P* :: *heap.rep list pred*
shows *heap.obj-at P a* (*Ref.set r v s*) ⟷ (*a* = *heap.addr-of r* ∧ *P* [*heap.rep.to v*])
∨ (*a* ≠ *heap.addr-of r* ∧ *heap.obj-at P a s*)
by (*simp add*: *Ref.set-def heap.set-def split*: *heap.obj-at.split*)

setup ‹*Sign.parent-path*›

lemmas *simps*[*simp*] =
Ref.addr-of

Ref.get.set-eq
Ref.get.set-neq
Ref.get.fun-upd
Ref.set.same
Ref.set.empty
Ref.set.fun-upd
Ref.set.obj-at-conv

setup ‹*Sign.parent-path*›

25.2 Arrays

25.2.1 Code generation constants: one-dimensional arrays

We ask that targets of the code generator provide implementations of one-dimensional arrays and the associated operations.
Notes:

- user-facing arrays make use of *Ix*
- due to the lack of bounds there is no *rep-val-inv*

datatype *'a one-dim-array* = *Array* (*addr-of*: *heap.addr*)

instantiation *one-dim-array* :: (*type*) *heap.addr-of*
begin

definition *addr-of-one-dim-array* :: *'a one-dim-array* ⇒ *heap.addr* **where**
addr-of-one-dim-array = *addr-of*

definition *rep-val-inv-one-dim-array* :: *'a one-dim-array* ⇒ *heap.rep list pred* **where**
[*simp*]: *rep-val-inv-one-dim-array a vs* ⟷ *True*

instance ..

end

setup ‹*Sign.mandatory-path ODArray*›

definition *get* :: *'a*::*heap.rep one-dim-array* ⇒ *nat* ⇒ *heap.t* ⇒ *'a* **where**
get a i s = *heap.get a s* ! *i*

definition *set* :: *'a*::*heap.rep one-dim-array* ⇒ *nat* ⇒ *'a* ⇒ *heap.t* ⇒ *heap.t* **where**
set a i v s = *heap.set a* ((*heap.get a s*)[*i*:=*v*]) *s*

definition *alloc* :: *'a list* ⇒ *heap.t* ⇒ (*'a*::*heap.rep one-dim-array* × *heap.t*) *set* **where**
alloc av s = {(*a*, *heap.set a av s*) |*a*. ¬*heap.present a s*}

definition *list-for* :: *'a*::*heap.rep one-dim-array* ⇒ *heap.t* ⇒ *'a list* **where**
list-for a = *heap.get a*

setup ‹*Sign.mandatory-path get*›

lemma *weak-cong*:
assumes *i* = *i'*
assumes *a* = *a'*
assumes *s* (*heap.addr-of a*) = *s'* (*heap.addr-of a'*)
shows *ODArray.get a i s* = *ODArray.get a' i' s'*
using *assms* **by** (*simp add*: *ODArray.get-def cong*: *heap.get.cong*)

lemma *weak-Id-on-proj-cong*:
assumes $i = i'$
assumes $a = a'$
assumes $(s, s') \in heap.Id_{\{a'\}}$
shows *ODArray.get a i s* = *ODArray.get a' i' s'*
using *assms* **by** (*simp add*: *ODArray.get-def cong*: *heap.get.Id-on-proj-cong*)

lemma *set-eq*:
assumes $i <$ *length* (*the* (*s* (*heap.addr-of a*)))
shows *ODArray.get a i* (*ODArray.set a i v s*) = *v*
using *assms* **by** (*simp add*: *ODArray.get-def ODArray.set-def heap.get.set-eq*) (*simp add*: *heap.get-def*)

lemma *set-neq*:
assumes $i \neq j$
shows *ODArray.get a i* (*ODArray.set a j v s*) = *ODArray.get a i s*
using *assms* **by** (*simp add*: *ODArray.get-def ODArray.set-def heap.get.set-eq*)

setup ‹*Sign.parent-path*›

setup ‹*Sign.parent-path*›

25.2.2 User-facing arrays

datatype (*'i*, *'a*) *array* = *Array* (*bounds*: (*'i* × *'i*)) (*arr*: *'a one-dim-array*)

hide-const (**open**) *bounds arr*

instantiation *array* :: (*Ix*, *heap.rep*) *heap.addr-of*
begin

definition *addr-of-array* :: (*'a*, *'b*) *array* ⇒ *heap.addr* **where**
addr-of-array = *addr-of* ∘ *array.arr*

definition *rep-val-inv-array* :: (*'a*, *'b*) *array* ⇒ *heap.rep list pred* **where**
rep-val-inv-array a vs ⟷
length vs = *length* (*Ix.interval* (*array.bounds a*))
∧ (∀ *v*∈*set vs*. *heap.rep.to* (*heap.rep.from v* :: *'b*) = *v*)

instance ..

end

instance *array* :: (*countable*, *type*) *heap.rep*
by *standard*
(*rule exI*[**where** *x*=λ*a*. *heap.Addr* (*to-nat* (*array.bounds a*)) (*addr-of* (*array.arr a*))],
rule injI, *simp add*: *array.expand one-dim-array.expand*)

setup ‹*Sign.mandatory-path Array*›

abbreviation (*input*) *square* :: (*'i*::*Ix* × *'i*, *'a*) *array* ⇒ *bool* **where**
square a ≡ *Ix.square* (*array.bounds a*)

abbreviation (*input*) *index* :: (*'i*::*Ix*, *'a*) *array* ⇒ *'i* ⇒ *nat* **where**
index a ≡ *Ix.index* (*array.bounds a*)

abbreviation (*input*) *interval* :: (*'i*::*Ix*, *'a*) *array* ⇒ *'i list* **where**
interval a ≡ *Ix.interval* (*array.bounds a*)

definition *get* :: ('*i*::*Ix*, '*a*::*heap.rep*) *array* ⇒ '*i* ⇒ *heap.t* ⇒ '*a* **where**
get a i = *ODArray.get* (*array.arr a*) (*Array.index a i*)

definition *set* :: ('*i*::*Ix*, '*a*::*heap.rep*) *array* ⇒ '*i* ⇒ '*a* ⇒ *heap.t* ⇒ *heap.t* **where**
set a i v = *ODArray.set* (*array.arr a*) (*Array.index a i*) *v*

definition *list-for* :: ('*i*::*Ix*, '*a*::*heap.rep*) *array* ⇒ *heap.t* ⇒ '*a list* **where**
list-for a = *ODArray.list-for* (*array.arr a*)

— can coerce any indexing regime into any other provided the contents fit
definition *coerce* :: ('*i*::*Ix*, '*a*::*heap.rep*) *array* ⇒ ('*j* × '*j*) ⇒ ('*j*::*Ix*, '*a*::*heap.rep*) *array option* **where**
coerce a b = (*if length* (*Array.interval a*) = *length* (*Ix.interval b*)
then Some (*Array b* (*array.arr a*))
else None)

definition *Id-on* :: ('*i*::*Ix*, '*a*::*heap.rep*) *array* ⇒ '*i set* ⇒ *heap.t rel* (‹*Array.Id*_, _›) **where**
Array.Id$_{a,\ is}$ = *heap.rep-inv-rel a* ∩ {(*s*, *s'*). ∀ *i*∈*is*. *Array.get a i s* = *Array.get a i s'*}

definition *modifies* :: ('*i*::*Ix*, '*a*::*heap.rep*) *array* ⇒ '*i set* ⇒ *heap.t rel* (‹*Array.modifies*_, _›) **where**
Array.modifies$_{a,\ is}$
= *heap.modifies*$_{\{a\}}$ ∩ {(*s*, *s'*). ∀ *i*∈*set* (*Array.interval a*) − *is*. *Array.get a i s* = *Array.get a i s'*}

lemma *simps*[*simp*]:
shows *heap.addr-of* (*array.arr a*) = *heap.addr-of a*
and *heap.addr-of* ∘ *array.arr* = *heap.addr-of*
by (*simp-all add*: *addr-of-array-def addr-of-one-dim-array-def*)

setup ‹*Sign.mandatory-path get*›

lemma *set-eq*:
assumes *heap.rep-inv a s*
assumes *i* ∈ *set* (*Array.interval a*)
shows *Array.get a i* (*Array.set a i v s*) = *v*
using *assms*
by (*simp add*: *Array.get-def Array.set-def ODArray.get.set-eq index-length rep-val-inv-array-def*
split: *heap.obj-at.split-asm*)

lemma *set-neq*:
assumes *i* ∈ *set* (*Array.interval a*)
assumes *j* ∈ *set* (*Array.interval a*)
assumes *i* ≠ *j*
shows *Array.get a j* (*Array.set a i v s*) = *Array.get a j s*
using *assms* **by** (*simp add*: *Array.get-def Array.set-def ODArray.get.set-neq index-eq-conv*)

lemma *Id-on-proj-cong*:
assumes *a* = *a'*
assumes *i* = *i'*
assumes (*s*, *s'*) ∈ *Array.Id*$_{a',\ \{i'\}}$
assumes *i'* ∈ *set* (*Array.interval a*)
shows *Array.get a i s* = *Array.get a' i' s'*
using *assms* **by** (*simp add*: *Array.get-def Array.Id-on-def*)

lemma *weak-cong*:
assumes *a* = *a'*
assumes *i* = *i'*
assumes *s* (*heap.addr-of a*) = *s'* (*heap.addr-of a'*)
shows *Array.get a i s* = *Array.get a' i' s'*

using *assms* **unfolding** *Array.get-def* **by** (*simp cong*: *ODArray.get.weak-cong*)

lemma *weak-Id-on-proj-cong*:
  assumes $i = i'$
  assumes $a = a'$
  assumes $(s, s') \in heap.Id_{\{a'\}}$
  shows *Array.get a i s* = *Array.get a' i' s'*
using *assms* **unfolding** *Array.get-def*
by (*simp add*: *heap.Id-on-def ODArray.get.weak-Id-on-proj-cong split*: *heap.obj-at.splits*)

lemma *ext*:
  assumes *heap.rep-inv a s*
  assumes *heap.rep-inv a s'*
  assumes $\forall i \in set$ (*Ix-class.interval* (*array.bounds a*)). *Array.get a i s* = *Array.get a i s'*
  shows *s* (*heap.addr-of a*) = *s'* (*heap.addr-of a*)
using *assms*
by (*simp add*: *Array.get-def ODArray.get-def heap.get-def rep-val-inv-array-def*
      split: *heap.obj-at.splits*)
   (*rule nth-equalityI*, *simp*, *metis index-forE nth-map nth-mem*)

setup ‹*Sign.parent-path*›

setup ‹*Sign.mandatory-path set*›

lemma *cong-deref*:
  assumes $a = a'$
  assumes $i = i'$
  assumes $v = v'$
  assumes *s r* = *s' r'*
  assumes $r = r'$
  shows *Array.set a i v s r* = *Array.set a' i' v' s' r'*
using *assms* **by** (*clarsimp simp*: *Array.set-def ODArray.set-def heap.set-def heap.get-def*)

lemma *same*:
  shows *Array.set a i v* (*Array.set a i v' s*) = *Array.set a i v s*
by (*simp add*: *Array.set-def ODArray.set-def heap.simps*)

setup ‹*Sign.parent-path*›

setup ‹*Sign.mandatory-path coerce*›

lemma *ex-bij-betw*:
  fixes *a* :: (′*i*::*Ix*, ′*a*::*heap.rep*) *array*
  fixes *b* :: ′*j*::*Ix* × ′*j*
  assumes *Array.coerce a b* = *Some a'*
  obtains *f* **where** *map f* (*Array.interval a*) = *Ix.interval b*
using *assms* **unfolding** *Array.coerce-def* **by** (*metis interval map-map map-nth not-None-eq*)

lemma *ex-bij-betw2*:
  fixes *a* :: (′*i*::*Ix*, ′*a*::*heap.rep*) *array*
  fixes *b* :: ′*j*::*Ix* × ′*j*
  assumes *Array.coerce a b* = *Some a'*
  obtains *f* **where** *map f* (*Ix.interval b*) = *Array.interval a*
using *assms* **by** (*metis Array.coerce-def Array.coerce.ex-bij-betw array.sel*(*1*) *option.distinct*(*1*))

setup ‹*Sign.parent-path*›

setup ‹*Sign.mandatory-path rep-inv*›

lemma *set*:
assumes *heap.rep-inv a s*
shows *heap.rep-inv a* (*Array.set a i v s*)
using *assms*
by (*simp add*: *Array.set-def ODArray.set-def rep-val-inv-array-def heap.set-def heap.get-def*
split: *heap.obj-at.splits*)

setup ‹*Sign.parent-path*›

setup ‹*Sign.mandatory-path modifies*›

lemma *heap-modifies-le*:
shows $Array.modifies_{a,\ is} \subseteq heap.modifies_{\{a\}}$
by (*simp add*: *Array.modifies-def*)

lemma *heap-rep-inv-rel-le*:
shows $Array.modifies_{a,\ is} \subseteq heap.rep\text{-}inv\text{-}rel\ a$
using *heap.modifies.rep-inv-rel-le*[**where** *rs*={*a*}] **by** (*auto simp*: *Array.modifies-def*)

lemma *empty*:
shows $Array.modifies_{a,\ \{\}} = Id \cap heap.rep\text{-}inv\text{-}rel\ a$ (**is** *?lhs* = *?rhs*)
by (*auto simp*: *Array.modifies-def heap.modifies.Id-conv heap.modifies.rep-inv*
elim: *heap.modifies.eqI Array.get.ext*)

lemma *mono*:
assumes $is \subseteq js$
shows $Array.modifies_{a,\ is} \subseteq Array.modifies_{a,\ js}$
using *assms* **by** (*auto simp*: *Array.modifies-def*)

lemma *INTER*:
shows $Array.modifies_{a,\ \bigcap x\in X.\ f\ x} = (\bigcap x\in X.\ Array.modifies_{a,\ f\ x}) \cap heap.modifies_{\{a\}}$
by (*auto simp*: *Array.modifies-def*)

lemma *Inter*:
shows $Array.modifies_{a,\ \bigcap X} = (\bigcap x\in X.\ Array.modifies_{a,\ x}) \cap heap.modifies_{\{a\}}$
by (*auto simp*: *Array.modifies-def*)

lemma *inter*:
shows $Array.modifies_{a,\ is} \cap Array.modifies_{a,\ js} = Array.modifies_{a,\ is \cap js}$
by (*auto simp*: *Array.modifies-def*)

lemma *UNION-subseteq*:
shows $(\bigcup x\in X.\ Array.modifies_{a,\ I\ x}) \subseteq Array.modifies_{a,\ (\bigcup x\in X.\ I\ x)}$
by (*simp add*: *Array.modifies.mono Sup-upper UN-least*)

lemma *union-subseteq*:
shows $Array.modifies_{a,\ is} \cup Array.modifies_{a,\ js} \subseteq Array.modifies_{a,\ is \cup js}$
by (*simp add*: *Array.modifies.mono*)

lemma *Diag-subseteq*:
assumes $\bigwedge s.\ P\ s \Longrightarrow heap.rep\text{-}inv\ a\ s$
shows $Diag\ P \subseteq Array.modifies_{a,\ is}$
using *assms* **by** (*auto simp*: *Array.modifies-def heap.modifies-def Diag-def*)

lemma *get*:
assumes $(s,\ s') \in Array.modifies_{a,\ is}$
assumes $i \in set\ (Array.interval\ a) - is$

shows *Array.get a i s′ = Array.get a i s*
using *assms* **by** (*simp add*: *Array.modifies-def*)

lemma *set*:
assumes *heap.rep-inv a s*
shows (*s*, *Array.set a i v s*) ∈ *heap.modifies*$_{\{a\}}$
using *assms*
by (*simp add*: *heap.modifies-def Array.set-def ODArray.set-def heap.set-def heap.get-def rep-val-inv-array-def*
split: *heap.obj-at.splits*)

lemma *Array-set*:
assumes *heap.rep-inv a s*
assumes *i* ∈ *set* (*Array.interval a*) ∩ *is*
shows (*s*, *Array.set a i v s*) ∈ *Array.modifies*$_{a,\ is}$
using *assms*
by (*auto simp*: *Array.modifies-def Array.rep-inv.set Array.modifies.set*
intro: *Array.get.set-neq*[*symmetric*])

lemma *Array-set-conv*:
assumes *i* ∈ *set* (*Array.interval a*) ∩ *is*
shows (*s*, *Array.set a i v s*) ∈ *Array.modifies*$_{a,\ is}$ ⟷ *heap.rep-inv a s* (**is** *?lhs* ⟷ *?rhs*)
proof(*rule iffI*)
show *?lhs* ⟹ *?rhs*
using *heap.modifies.rep-inv-rel-le*[*of* {*a*}, *simplified*] **by** (*auto simp*: *Array.modifies-def*)
from *assms* **show** *?rhs* ⟹ *?lhs*
by (*simp add*: *Array.modifies.Array-set*)
qed

setup ‹*Sign.parent-path*›

lemmas *simps′* =
Array.rep-inv.set
Array.get.set-eq

setup ‹*Sign.parent-path*›

setup ‹*Sign.mandatory-path heap.Id-on.Array*›

lemma *Id-on-le*:
shows *heap.Id*$_{\{a\}}$ ⊆ *Array.Id*$_{a,\ is}$
by (*auto simp*: *Array.Id-on-def heap.Id-on-def Array.get-def ODArray.get-def heap.get-def*)

setup ‹*Sign.parent-path*›

setup ‹*Sign.mandatory-path Array.Id-on*›

lemma *empty*:
shows *Array.Id*$_{a,\ \{\}}$ = *heap.rep-inv-rel a*
by (*simp add*: *Array.Id-on-def*)

lemma *mono*:
assumes *is* ⊆ *js*
shows *Array.Id*$_{a,\ js}$ ⊆ *Array.Id*$_{a,\ is}$
using *assms* **by** (*auto simp*: *Array.Id-on-def*)

lemma *insert*:
shows *Array.Id*$_{a,\ insert\ i\ is}$ = *Array.Id*$_{a,\ \{i\}}$ ∩ *Array.Id*$_{a,\ is}$
by (*fastforce simp*: *Array.Id-on-def*)

lemma *union*:
  shows $Array.Id_{a,\ is \cup js} = Array.Id_{a,\ is} \cap Array.Id_{a,\ js}$
by (*fastforce simp*: *Array.Id-on-def*)

lemma *rep-inv-rel*:
  shows $Array.Id_{a,\ is} \subseteq heap.rep\text{-}inv\text{-}rel\ a$
by (*simp add*: *Array.Id-on-def*)

lemma *eq-heap-Id-on*:
  assumes *set* (*Array.interval a*) $\subseteq$ *is*
  shows $Array.Id_{a,\ is} = heap.Id_{\{a\}}$
by (*rule antisym*[*OF - heap.Id-on.Array.Id-on-le*])
   (*use assms* **in** ‹*force simp*: *Array.Id-on-def heap.Id-on-def elim*: *Array.get.ext*›)

setup ‹*Sign.parent-path*›

25.2.3 Stability

setup ‹*Sign.mandatory-path stable.heap.Id-on.Array*›

lemma *get*[*stable.intro*]:
  assumes $a \in as$
  shows *stable* $heap.Id_{as}$ ($\lambda s.\ P$ (*Array.get a i s*))
using *assms* **by** (*auto simp*: *stable-def monotone-def heap.Id-on-def cong*: *Array.get.weak-cong*)

lemma *get-chain*: — difficult to apply
  assumes $\bigwedge v.$ *stable* $heap.Id_{as}$ ($P\ v$)
  assumes $a \in as$
  shows *stable* $heap.Id_{as}$ ($\lambda s.\ P$ (*Array.get a i s*) *s*)
using *assms* **by** (*auto simp*: *stable-def monotone-def heap.Id-on-def cong*: *Array.get.weak-cong*)

setup ‹*Sign.parent-path*›

setup ‹*Sign.mandatory-path stable.Array.Id-on.Array*›

lemma *get*[*stable.intro*]:
  assumes $i \in is$
  shows *stable* $Array.Id_{a,\ is}$ ($\lambda s.\ P$ (*Array.get a i s*))
using *assms* **by** (*auto simp*: *stable-def monotone-def Array.Id-on-def*)

lemma *get-chain*: — difficult to apply
  assumes $\bigwedge v.$ *stable* $Array.Id_{a,\ is}$ ($P\ v$)
  assumes $i \in is$
  shows *stable* $Array.Id_{a,\ is}$ ($\lambda s.\ P$ (*Array.get a i s*) *s*)
using *assms* **by** (*auto simp*: *stable-def monotone-def Array.Id-on-def*)

setup ‹*Sign.parent-path*›

setup ‹*Sign.mandatory-path stable.heap.Array.Id-on.heap*›

lemma *rep-inv*[*stable.intro*]:
  shows *stable* $Array.Id_{a,\ is}$ (*heap.rep-inv a*)
by (*simp add*: *stable-def monotone-def Array.Id-on-def*)

setup ‹*Sign.parent-path*›

setup ‹*Sign.mandatory-path stable.heap.Array.modifies.heap*›

lemma *rep-inv*[*stable.intro*]:
shows *stable Array.modifies*$_{a,\ is}$ (*heap.rep-inv a*)
by (*simp add*: *stable-def monotone-def Array.modifies-def heap.modifies-def*)

setup ‹*Sign.parent-path*›

setup ‹*Sign.mandatory-path stable.heap.Array.modifies.Array*›

lemma *get*[*stable.intro*]:
assumes $i \in$ *set* (*Array.interval a*) − *is*
shows *stable Array.modifies*$_{a,\ is}$ (λ*s*. *P* (*Array.get a i s*))
using *assms* **by** (*simp add*: *stable-def monotone-def Array.modifies-def*)

lemma *get-chain*: — difficult to apply
assumes ⋀*v*. *stable Array.modifies*$_{a,\ is}$ (*P v*)
assumes $i \in$ *set* (*Array.interval a*) − *is*
shows *stable Array.modifies*$_{a,\ is}$ (λ*s*. *P* (*Array.get a i s*) *s*)
using *assms* **by** (*simp add*: *stable-def monotone-def Array.modifies-def*)

setup ‹*Sign.parent-path*›

26 A concurrent variant of Imperative HOL

We model programs operating on sequentially-consistent memory with the type (*heap.t*, *'v*) *prog*.
Source materials:

- $ISABELLE_HOME/src/HOL/Imperative_HOL/Heap_Monad.thy
- $ISABELLE_HOME/src/HOL/Imperative_HOL/Array.thy
- $ISABELLE_HOME/src/HOL/Imperative_HOL/Ref.thy
  - note that ImperativeHOL is deterministic and sequential

type-synonym *'v imp* = (*heap.t*, *'v*) *prog*

setup ‹*Sign.mandatory-path prog*›

definition *raise* :: *String.literal* ⇒ *'a imp* **where** — the literal is just decoration
raise s = ⊥

definition *assert* :: *bool* ⇒ *unit imp* **where**
assert P = (*if P then prog.return* () *else prog.raise STR ''assert''*)

setup ‹*Sign.mandatory-path Ref*›

definition *ref* :: *'a*::*heap.rep* ⇒ *'a ref imp* **where**
ref v = *prog.action* {(*r*, *s*, *s'*). (*r*, *s'*) ∈ *Ref.alloc v s*}

definition *lookup* :: *'a*::*heap.rep ref* ⇒ *'a imp* (‹!-› *61*) **where**
lookup r = *prog.read* (*Ref.get r*)

definition *update* :: *'a ref* ⇒ *'a*::*heap.rep* ⇒ *unit imp* (‹- := -› *62*) **where**
update r v = *prog.write* (*Ref.set r v*)

setup ‹*Sign.parent-path*›

setup ‹*Sign.mandatory-path Array*›

definition *new* :: ('*i* × '*i*) ⇒ '*a* ⇒ ('*i*::*Ix*, '*a*::*heap.rep*) *array imp* **where**
new b v = *prog.action* {(*Array b a*, *s*, *s'*) |*a s s'*. (*a*, *s'*) ∈ *ODArray.alloc* (*replicate* (*length* (*Ix.interval b*)) *v*) *s*}

definition *make* :: ('*i* × '*i*) ⇒ ('*i* ⇒ '*a*) ⇒ ('*i*::*Ix*, '*a*::*heap.rep*) *array imp* **where**
make b f = *prog.action* {(*Array b a*, *s*, *s'*) |*a s s'*. (*a*, *s'*) ∈ *ODArray.alloc* (*map f* (*Ix.interval b*)) *s*}

— Approximately Haskell's *listArray*: "Construct an array from a pair of bounds and a list of values in index order."
definition *of-list* :: ('*i* × '*i*) ⇒ '*a list* ⇒ ('*i*::*Ix*, '*a*::*heap.rep*) *array imp* **where**
of-list b xs = *prog.action* {(*Array b a*, *s*, *s'*) |*a s s'*. *length* (*Ix.interval b*) ≤ *length xs* ∧ (*a*, *s'*) ∈ *ODArray.alloc xs s*}

definition *nth* :: ('*i*::*Ix*, '*a*::*heap.rep*) *array* ⇒ '*i* ⇒ '*a imp* **where**
nth a i = *prog.read* (λ*s*. *Array.get a i s*)

definition *upd* :: ('*i*::*Ix*, '*a*::*heap.rep*) *array* ⇒ '*i* ⇒ '*a* ⇒ *unit imp* **where**
upd a i v = *prog.write* (*Array.set a i v*)

— derived operations; observe the lack of atomicity

definition *freeze* :: ('*i*::*Ix*, '*a*::*heap.rep*) *array* ⇒ '*a list imp* **where**
freeze a = *prog.fold-mapM* (*prog.Array.nth a*) (*Array.interval a*)

definition *swap* :: ('*i*::*Ix*, '*a*::*heap.rep*) *array* ⇒ '*i* ⇒ '*i* ⇒ *unit imp*
where
swap a i j =
do {
x ← *prog.Array.nth a i*;
y ← *prog.Array.nth a j*;
prog.Array.upd a i y;
prog.Array.upd a j x;
prog.return ()
}

declare *prog.raise-def*[*code del*]
declare *prog.Ref.ref-def*[*code del*]
declare *prog.Ref.lookup-def*[*code del*]
declare *prog.Ref.update-def*[*code del*]
declare *prog.Array.new-def*[*code del*]
declare *prog.Array.make-def*[*code del*]
declare *prog.Array.of-list-def*[*code del*]
declare *prog.Array.nth-def*[*code del*]
declare *prog.Array.upd-def*[*code del*]
declare *prog.Array.freeze-def*[*code del*]

Operations on two-dimensional arrays **definition** *fst-app-chaotic* :: ('*a*::*Ix*, '*b*::*Ix*) *two-dim* ⇒ ('*a* ⇒ ('*s*, *unit*) *prog*) ⇒ ('*s*, *unit*) *prog* **where**
fst-app-chaotic b f = *prog.set-app f* (*set* (*Ix.interval* (*fst-bounds b*)))

definition *fst-app* :: ('*a*::*Ix*, '*b*::*Ix*) *two-dim* ⇒ ('*a* ⇒ ('*s*, *unit*) *prog*) ⇒ ('*s*, *unit*) *prog* **where**
fst-app b f = *prog.app f* (*Ix.interval* (*fst-bounds b*))

lemma *fst-app-fst-app-chaotic-le*:
shows *prog.Array.fst-app b f* ≤ *prog.Array.fst-app-chaotic b f*
unfolding *prog.Array.fst-app-chaotic-def prog.Array.fst-app-def*
by (*strengthen ord-to-strengthen*(*1*)[*OF prog.app.set-app-le*]) (*auto simp*: *distinct-interval*)

setup ‹*Sign.parent-path*›

setup ‹*Sign.parent-path*›

setup ‹*Sign.mandatory-path ag.prog*›

lemmas *fst-app-chaotic* =
ag.prog.app-set[**where** *X*=*set* (*Ix.interval* (*fst-bounds b*)) **for** *b*, *folded prog.Array.fst-app-chaotic-def*]
lemmas *fst-app* =
ag.prog.app[**where** *xs*=*Ix.interval* (*fst-bounds b*) **for** *b*, *folded prog.Array.fst-app-def*]

setup ‹*Sign.parent-path*›

26.1 Code generator setup

26.1.1 Haskell

code-printing code-module *Heap* ⇀ (*Haskell*)
‹
−− Sequentially−consistent primitives
−− Arrays:
−− https://hackage.haskell.org/package/array−0.5.4.0/docs/Data−Array−IO.html
−− https://hackage.haskell.org/package/array−0.5.4.0/docs/src/Data.Array.Base.html
module Heap (
  Prog
, *Ref*, *newIORef*, *readIORef*, *writeIORef*
, *Array*, *newArray*, *newListArray*, *newFunArray*, *readArray*, *writeArray*
, *parallel*
) *where*

import Control.Concurrent (*forkIO*)
import qualified Control.Concurrent.MVar as MVar
import qualified Data.Array.IO as Array
import Data.IORef (*IORef*, *newIORef*, *readIORef*, *atomicWriteIORef*)
import Data.List (*genericLength*)

type Prog a b = IO b
type Array a = Array.IOArray Integer a
type Ref a = Data.IORef.IORef a

writeIORef :: IORef a −> a −> IO ()
writeIORef = atomicWriteIORef −− could use the strict variant

newArray :: Integer −> a −> IO (*Array a*)
newArray k = Array.newArray (*0*, *k − 1*)

newFunArray :: Integer −> (Integer −> a) −> IO (*Array a*)
newFunArray k f = Array.newListArray (*0*, *k − 1*) (*map f* [*0..k−1*])

newListArray :: Integer −> [a] −> IO (*Array a*)
newListArray k xs = Array.newListArray (*0*, *k*) *xs*

readArray :: Array a −> Integer −> IO a
readArray = Array.readArray

writeArray :: Array a −> Integer −> a −> IO ()
writeArray = Array.writeArray −− probably should be the WMM atomic op

```
{−
−− 'forkIO' is reputedly cheap, but other papers imply the use of worker threads, perhaps for other reasons
−− note we don't want forkFinally as we don't model exceptions
parallel' :: IO a −> IO b −> IO (a, b)
parallel' p q = do
  mvar <− MVar.newEmptyMVar
  forkIO (p >>= MVar.putMVar mvar) −− note putMVar is lazy
  b <− q
  a <− MVar.takeMVar mvar
  return (a, b)
−}

parallel :: IO () −> IO () −> IO ()
parallel p q = do
  mvar <− MVar.newEmptyMVar
  forkIO (p >> MVar.putMVar mvar ()) −− note putMVar is lazy
  b <− q
  a <− MVar.takeMVar mvar
  return ()
›
```

code-reserved (*Haskell*) *Ix*

code-printing type-constructor *prog* ⇀ (*Haskell*) *Heap.Prog* - -
code-monad *prog.bind Haskell*
code-printing constant *prog.return* ⇀ (*Haskell*) *return*
code-printing constant *prog.raise* ⇀ (*Haskell*) *error*
code-printing constant *prog.parallel* ⇀ (*Haskell*) *Heap.parallel*

Intermediate operation avoids invariance problem in *Scala* (similar to value restriction)

setup ‹*Sign.mandatory-path Ref*›

definition *ref'* **where**
  [*code del*]: *ref'* = *prog.Ref.ref*

lemma [*code*]:
  prog.Ref.ref x = *Ref.ref' x*
by (*simp add*: *Ref.ref'-def*)

setup ‹*Sign.parent-path*›

code-printing type-constructor *ref* ⇀ (*Haskell*) *Heap.Ref* -
code-printing constant *Ref* ⇀ (*Haskell*) *error*/ *bare Ref*
code-printing constant *Ref.ref'* ⇀ (*Haskell*) *Heap.newIORef*
code-printing constant *prog.Ref.lookup* ⇀ (*Haskell*) *Heap.readIORef*
code-printing constant *prog.Ref.update* ⇀ (*Haskell*) *Heap.writeIORef*
code-printing constant *HOL.equal* :: 'a *ref* ⇒ 'a *ref* ⇒ *bool* ⇀ (*Haskell*) **infix** 4 ==
code-printing class-instance *ref* :: *HOL.equal* ⇀ (*Haskell*) −

The target language only has to provide one-dimensional arrays indexed by *integer*.

setup ‹*Sign.mandatory-path prog.Array*›

definition *new'* :: *integer* ⇒ 'a ⇒ 'a::*heap.rep one-dim-array imp* **where**
  new' k v = *prog.action* {(*a*, *s*, *s'*) |*a s s'*. (*a*, *s'*) ∈ *ODArray.alloc* (*replicate* (*nat-of-integer k*) *v*) *s*}

declare *prog.Array.new'-def*[*code del*]

lemma *new-new'*[*code*]:

shows *prog.Array.new b v = prog.Array.new' (of-nat (length (Ix.interval b))) v ⤜ prog.return ∘ Array b*
by (*force simp*: *prog.Array.new-def prog.Array.new'-def prog.vmap.action*
simp flip: *prog.vmap.eq-return*
intro: *arg-cong*[**where** *f*=*prog.action*])

definition *make'* :: *integer* ⇒ (*integer* ⇒ *'a*) ⇒ *'a*::*heap.rep one-dim-array imp* **where**
make' k f = prog.action {(*a*, *s*, *s'*) |*a s s'*. (*a*, *s'*) ∈ *ODArray.alloc* (*map* (*f* ∘ *of-nat*) [*0..<nat-of-integer k*]) *s*}

declare *prog.Array.make'-def*[*code del*]

lemma *make-make'*[*code*]:
shows *prog.Array.make b f*
= prog.Array.make' (of-nat (length (Ix.interval b))) (λi. f (Ix.interval b ! nat-of-integer i))
⤜ prog.return ∘ Array b
by (*force simp*: *interval-map prog.Array.make-def prog.Array.make'-def prog.vmap.action comp-def*
simp flip: *prog.vmap.eq-return*
intro: *arg-cong*[**where** *f*=*prog.action*])

definition *of-list'* :: *integer* ⇒ *'a list* ⇒ *'a*::*heap.rep one-dim-array imp* **where**
of-list' k xs = prog.action {(*a*, *s*, *s'*) |*a s s'*. *nat-of-integer k* ≤ *length xs* ∧ (*a*, *s'*) ∈ *ODArray.alloc xs s*}

declare *prog.Array.of-list'-def*[*code del*]

lemma *of-list-of-list'*[*code*]:
shows *prog.Array.of-list b xs*
= prog.Array.of-list' (of-nat (length (Ix.interval b))) xs ⤜ prog.return ∘ Array b
by (*force simp*: *prog.Array.of-list-def prog.Array.of-list'-def prog.vmap.action*
simp flip: *prog.vmap.eq-return*
intro: *arg-cong*[**where** *f*=*prog.action*])

definition *nth'* :: *'a*::*heap.rep one-dim-array* ⇒ *integer* ⇒ *'a imp* **where**
nth' a i = prog.read (ODArray.get a (nat-of-integer i))

declare *prog.Array.nth'-def*[*code del*]

lemma *nth-nth'*[*code*]:
shows *prog.Array.nth a i = prog.Array.nth' (array.arr a) (of-nat (Array.index a i))*
by (*simp add*: *prog.Array.nth-def prog.Array.nth'-def Array.get-def*)

definition *upd'* :: *'a*::*heap.rep one-dim-array* ⇒ *integer* ⇒ *'a*::*heap.rep* ⇒ *unit imp* **where**
upd' a i v = prog.write (ODArray.set a (nat-of-integer i) v)

declare *prog.Array.upd'-def*[*code del*]

lemma *upd-upd'*[*code*]:
shows *prog.Array.upd a i v = prog.Array.upd' (array.arr a) (of-nat (Array.index a i)) v*
by (*simp add*: *prog.Array.upd-def prog.Array.upd'-def Array.set-def*)

setup ‹*Sign.parent-path*›

code-printing type-constructor *one-dim-array* ⇀ (*Haskell*) *Heap.Array/ -*
code-printing constant *one-dim-array.Array* ⇀ (*Haskell*) *error/ bare Array*
code-printing constant *prog.Array.new'* ⇀ (*Haskell*) *Heap.newArray*
code-printing constant *prog.Array.make'* ⇀ (*Haskell*) *Heap.newFunArray*
code-printing constant *prog.Array.of-list'* ⇀ (*Haskell*) *Heap.newListArray*
code-printing constant *prog.Array.nth'* ⇀ (*Haskell*) *Heap.readArray*
code-printing constant *prog.Array.upd'* ⇀ (*Haskell*) *Heap.writeArray*
code-printing constant *HOL.equal* :: (*'i*, *'a*) *array* ⇒ (*'i*, *'a*) *array* ⇒ *bool* ⇀ (*Haskell*) **infix** *4 ==*

code-printing class-instance *array* :: *HOL.equal* ⇀ (*Haskell*) −

26.2 Value-returning parallel

definition *parallelP′* :: *'a*::*heap.rep imp* ⇒ *'b*::*heap.rep imp* ⇒ (*'a* × *'b*) *imp* **where**
parallelP′ P$_1$ *P*$_2$ = *do* {
r$_1$ ← *prog.Ref.ref undefined*
; *r*$_2$ ← *prog.Ref.ref undefined*
; ((*P*$_1$ ⤜ *prog.Ref.update r*$_1$) ‖ (*P*$_2$ ⤜ *prog.Ref.update r*$_2$))
; *v*$_1$ ← *prog.Ref.lookup r*$_1$
; *v*$_2$ ← *prog.Ref.lookup r*$_2$
; *prog.return* (*v*$_1$, *v*$_2$)
}

27 Total store order (TSO)

The total store order (TSO) memory model (Owens, Sarkar, and Sewell (2009); valid on multicore x86) can be modelled as a closure as demonstrated by Jagadeesan, Petri, and Riely (2012, p182). Essentially this is done by incorporating a write buffer into each thread's local state and adding buffer draining opportunities before and after every command. The only subtlety is that the all threads involved in a parallel composition need to start and end with empty write buffers (see §27).

We configure the code generator in §27.3.

Comparison with Jagadeesan et al. (2012):

- We ignore mumbling-related issues and it doesn't make any difference
  - in our model we commit writes one at a time; mumbling allows several to be committed at once (p182) which we model as an uninterrupted sequence of individual writes
  - if we allowed *commit-writes* to commit multiple writes in a single step then *tso-closure* would not be idempotent
- their semantics is for terminating computations only; ours is for safety only
- their language is deterministic, ours is non-deterministic
- They do not provide many general laws for TSO
- Their claims that the semantics allows them to prove things (§5) is not substantiated

type-synonym *write-buffer* = *heap.write list*

definition *apply-writes* :: *write-buffer* ⇒ *heap.t* ⇒ *heap.t* **where**
apply-writes ws = *fold* (λ*w*. (∘) (*heap.apply-write w*)) *ws id*

lemma *apply-write-present*:
assumes *heap.present r s*
shows *heap.present r* (*heap.apply-write w s*)
using *assms* **by** (*cases w*) (*simp split*: *heap.obj-at.splits*)

lemma *apply-writes-present*:
assumes *heap.present r s*
shows *heap.present r* (*apply-writes wb s*)
using *assms* **by** (*induct wb arbitrary*: *s*) (*simp-all add*: *apply-writes-def fold-fun apply-write-present*)

setup ‹*Sign.mandatory-path raw*›

type-synonym *'v tso* = *write-buffer* ⇒ (*heap.t*, *'v* × *write-buffer*) *prog*

definition *bind* :: *'a raw.tso* ⇒ (*'a* ⇒ *'b raw.tso*) ⇒ *'b raw.tso* **where**
bind f g = (λ*wb*. *f wb* ⤜ *uncurry g*)

adhoc-overloading
Monad-Syntax.bind ⇌ *raw.bind*

definition *prim-return* :: *'a* ⇒ *'a raw.tso* **where**
prim-return v = (λ*wb*. *prog.return* (*v*, *wb*))

setup ‹*Sign.mandatory-path bind*›

lemma *mono*:
assumes *f* ≤ *f'*
assumes ⋀*x*. *g x* ≤ *g' x*
shows *raw.bind f g* ≤ *raw.bind f' g'*
using *assms* **by** (*fastforce simp*: *raw.bind-def prog.bind.mono le-fun-def intro*: *prog.bind.mono*)

lemma *strengthen*[*strg*]:
assumes *st-ord F f f'*
assumes ⋀*x*. *st-ord F* (*g x*) (*g' x*)
shows *st-ord F* (*raw.bind f g*) (*raw.bind f' g'*)
using *assms* **by** (*cases F*; *clarsimp intro*!: *raw.bind.mono*)

lemma *mono2mono*[*cont-intro*, *partial-function-mono*]:
assumes *monotone orda* (≤) *F*
assumes ⋀*x*. *monotone orda* (≤) (λ*y*. *G y x*)
shows *monotone orda* (≤) (λ*f*. *raw.bind* (*F f*) (*G f*))
using *assms* **unfolding** *monotone-def* **by** (*meson raw.bind.mono*)

lemma *botL*:
shows *raw.bind* ⊥ *g* = ⊥
by (*simp add*: *raw.bind-def fun-eq-iff prog.bind.botL*)

lemma *bind*:
fixes *f* :: *- raw.tso*
shows *f* ⤜ *g* ⤜ *h* = *f* ⤜ (λ*x*. *g x* ⤜ *h*)
by (*simp add*: *raw.bind-def fun-eq-iff split-def prog.bind.bind*)

lemma *prim-return*:
shows *prim-returnL*: *raw.bind* (*raw.prim-return v*) = (λ*g*. *g v*)
and *prim-returnR*: *f* ⤜ *raw.prim-return* = *f*
by (*simp-all add*: *fun-eq-iff raw.prim-return-def raw.bind-def split-def prog.bind.return*)

lemma *supL*:
fixes *g* :: *- ⇒ - raw.tso*
shows *f*$_1$ ⊔ *f*$_2$ ⤜ *g* = (*f*$_1$ ⤜ *g*) ⊔ (*f*$_2$ ⤜ *g*)
by (*simp add*: *raw.bind-def fun-eq-iff prog.bind.supL*)

lemma *supR*:
fixes *f* :: *- raw.tso*
shows *f* ⤜ (λ*v*. *g*$_1$ *v* ⊔ *g*$_2$ *v*) = (*f* ⤜ *g*$_1$) ⊔ (*f* ⤜ *g*$_2$)
by (*simp add*: *raw.bind-def fun-eq-iff split-def prog.bind.supR*)

lemma *SUPL*:
fixes *X* :: *- set*
fixes *f* :: *- ⇒ - raw.tso*
shows (⨆*x*∈*X*. *f x*) ⤜ *g* = (⨆*x*∈*X*. *f x* ⤜ *g*)
by (*simp add*: *raw.bind-def fun-eq-iff prog.bind.SUPL image-image*)

lemma *SUPR*:
  fixes X :: - *set*
  fixes f :: - *raw.tso*
  shows $f \gg\!= (\lambda v.\ \bigsqcup x{\in}X.\ g\ x\ v) = (\bigsqcup x{\in}X.\ f \gg\!= g\ x) \sqcup (f \gg\!= \bot)$
by (*simp add*: *raw.bind-def split-def fun-eq-iff image-image bot-fun-def prog.bind.SUPR*)

lemma *SUPR-not-empty*:
  fixes f :: - *raw.tso*
  assumes $X \neq \{\}$
  shows $f \gg\!= (\lambda v.\ \bigsqcup x{\in}X.\ g\ x\ v) = (\bigsqcup x{\in}X.\ f \gg\!= g\ x)$
by (*simp add*: *raw.bind-def split-def fun-eq-iff image-image prog.bind.SUPR-not-empty*[*OF assms*])

lemma *mcont2mcont*[*cont-intro*]:
  assumes *mcont luba orda Sup* ($\leq$) f
  assumes $\bigwedge v.$ *mcont luba orda Sup* ($\leq$) ($\lambda x.\ g\ x\ v$)
  shows *mcont luba orda Sup* ($\leq$) ($\lambda x.$ *raw.bind* ($f\ x$) ($g\ x$))
proof(*rule ccpo.mcont2mcont'*[*OF complete-lattice-ccpo - - assms*(*1*)])
  show *mcont Sup* ($\leq$) *Sup* ($\leq$) ($\lambda f.$ *raw.bind* f ($g\ x$)) **for** x
    by (*intro mcontI contI monotoneI*) (*simp-all add*: *raw.bind.mono flip*: *raw.bind.SUPL*)
  show *mcont luba orda Sup* ($\leq$) ($\lambda x.$ *raw.bind* f ($g\ x$)) **for** f
    by (*intro mcontI monotoneI contI*)
        (*simp-all add*: *mcont-monoD*[*OF assms*(*2*)] *raw.bind.mono flip*: *raw.bind.SUPR-not-empty contD*[*OF mcont-cont*[*OF assms*(*2*)]])
qed

setup ‹*Sign.parent-path*›

interpretation *kleene*: *kleene raw.prim-return* () $\lambda x\ y.$ *raw.bind* x $\langle y \rangle$
by *standard* (*simp-all add*: *raw.bind.prim-return raw.bind.botL raw.bind.bind raw.bind.supL raw.bind.supR*)

primrec *commit-write* :: *unit raw.tso* **where**
  commit-write [] = *prog.return* ((), [])
| *commit-write* (w # wb) = *prog.action* {((), wb), h, *heap.apply-write* w h) |h. *True*}

definition *commit-writes* :: *unit raw.tso* **where**
  commit-writes = *raw.kleene.star raw.commit-write*

setup ‹*Sign.mandatory-path tso*›

definition *cl* :: $'v$ *raw.tso* $\Rightarrow$ $'v$ *raw.tso* **where**
  cl P = *raw.commit-writes* $\gg$ P $\gg\!=$ ($\lambda v.$ *raw.commit-writes* $\gg$ *raw.prim-return* v)

setup ‹*Sign.parent-path*›

definition *action* :: (*write-buffer* $\Rightarrow$ ($'v$ $\times$ *write-buffer* $\times$ *heap.t* $\times$ *heap.t*) *set*) $\Rightarrow$ $'v$ *raw.tso* **where**
  action F = *raw.tso.cl* ($\lambda wb.$ *prog.action* {((v, wb @ ws), ss') |v ss' ws. (v, ws, ss') $\in$ F wb})

definition *return* :: $'v \Rightarrow$ $'v$ *raw.tso* **where**
  return v = *raw.action* $\langle \{v\} \times \{[]\} \times Id \rangle$

definition *guard* :: (*write-buffer* $\Rightarrow$ *heap.t pred*) $\Rightarrow$ *unit raw.tso* **where**
  guard g = *raw.action* ($\lambda wb.$ {()} $\times$ {[]} $\times$ *Diag* (g wb))

definition *MFENCE* :: *unit raw.tso* **where**
  MFENCE = *raw.guard* (λwb s. wb = [])

definition *vmap* :: ($'v \Rightarrow$ $'w$) $\Rightarrow$ $'v$ *raw.tso* $\Rightarrow$ $'w$ *raw.tso* **where**

vmap vf P = (λ*wb*. *prog.vmap* (*map-prod vf id*) (*P wb*))

— Parallel composition
definition *t2p* :: '*v raw.tso* ⇒ (*heap.t*, '*v*) *prog* **where**
t2p P = *P* [] ⤜ (λ(*v*, *wb*). *raw.MFENCE wb* ≫ *prog.return v*)

— Jagadeesan et al. (2012, p184 rule PAR-CMD): perform MFENCE before fork
definition *parallel* :: *unit raw.tso* ⇒ *unit raw.tso* ⇒ *unit raw.tso* **where**
parallel P Q = *raw.MFENCE* ≫ ⟨(*raw.t2p P* ∥ *raw.t2p Q*) ≫ *prog.return* ((), [])⟩

lemma *return-alt-def*:
shows *raw.return* = (λ*v*. *raw.tso.cl* (*raw.prim-return v*))
by (*fastforce simp*: *raw.return-def raw.action-def raw.prim-return-def prog.return-def*
intro: *arg-cong*[**where** *f*=λ*P*. *raw.tso.cl P wb* **for** *wb*] *arg-cong*[**where** *f*=*prog.action*])

setup ‹*Sign.mandatory-path commit-writes*›

lemma *return-le*:
shows *raw.prim-return* () ≤ *raw.commit-writes*
unfolding *raw.commit-writes-def* **by** (*subst raw.kleene.star.simps*) *simp*

lemma *return-le'*:
shows *prog.return* ((), *wb*) ≤ *raw.commit-writes wb*
using *raw.commit-writes.return-le* **by** (*simp add*: *raw.prim-return-def le-fun-def*)

lemma *commit-writes*:
shows *raw.commit-writes* ≫ *raw.commit-writes* = *raw.commit-writes*
by (*simp add*: *raw.commit-writes-def raw.kleene.star-comp-star*)

lemma *Nil*:
shows *raw.commit-writes* [] = *prog.return* ((), []) (**is** *?lhs* = *?rhs*)
proof(*rule antisym*)
show *?lhs* ≤ *?rhs*
unfolding *raw.commit-writes-def*
by (*induct rule*: *raw.kleene.star.fixp-induct*)
(*simp-all add*: *raw.bind-def raw.prim-return-def prog.bind.returnL prog.p2s.bot spec.bind.mono*)
show *?rhs* ≤ *?lhs*
unfolding *raw.commit-writes-def*
by (*subst raw.kleene.star.simps*) (*simp add*: *raw.bind-def raw.prim-return-def*)
qed

lemma *Cons*:
shows *raw.commit-writes* (*w* # *wb*)
= (*raw.commit-write* [*w*] ≫ *raw.commit-writes wb*) ⊔ *raw.prim-return* () (*w* # *wb*)
apply (*simp add*: *raw.commit-writes-def*)
apply (*subst* (*1*) *raw.kleene.star.simps*)
apply (*subst* (*1*) *raw.bind-def*)
apply *simp*
apply (*subst prog.action.return-const*[**where** *F*={(*s*, *heap.apply-write w s*) |*s*. *True*} **and** *V*={((), *wb*)} **and** *W*={((), [])}, *simplified Pair-image*[*symmetric*] *image-def*, *simplified*])
apply (*simp add*: *prog.bind.bind prog.bind.returnL*)
done

lemma *Cons-le*:
shows *raw.commit-write* [*w*] ≫ *raw.commit-writes wb* ≤ *raw.commit-writes* (*w* # *wb*)
by (*simp add*: *raw.commit-writes.Cons*)

setup ‹*Sign.parent-path*›

setup ‹*Sign.parent-path*›

setup ‹*Sign.mandatory-path spec.singleton.raw*›

lemma *prim-return-Nil-le*:
  shows ⟪*s*, [], *Some* ((), *wb*)⟫ ≤ *prog.p2s* (*raw.prim-return* () *wb*)
by (*simp add*: *raw.prim-return-def prog.p2s.return spec.interference.cl.return*
        spec.bind.continueI[**where** *xs*=[], *simplified*] *spec.singleton.le-conv*)

setup ‹*Sign.parent-path*›

setup ‹*Sign.mandatory-path spec.singleton.raw.commit-writes*›

lemma *noop-le*:
  shows ⟪*s*, [], *Some* ((), *wb*)⟫ ≤ *prog.p2s* (*raw.commit-writes wb*)
unfolding *raw.commit-writes-def*
by (*rule order.trans*[*OF spec.singleton.raw.prim-return-Nil-le*
                  prog.p2s.mono[*OF le-funD*[*OF raw.kleene.epsilon-star-le*]]])

lemma *wb-suffix*:
  assumes ⟪*s*, *xs*, *Some* ((), *wb*′)⟫ ≤ *prog.p2s* (*raw.commit-writes wb*)
  shows *suffix wb*′ *wb*
using *assms*
by (*induct wb arbitrary*: *s xs*)
   (*auto simp*: *raw.commit-writes.Nil raw.commit-writes.Cons raw.prim-return-def*
        prog.p2s.simps prog.p2s.return spec.interference.cl.return
        trace.split-all spec.singleton.le-conv
        suffix-ConsI
    elim!: *spec.singleton.bind-le*)

setup ‹*Sign.parent-path*›

setup ‹*Sign.mandatory-path raw*›

setup ‹*Sign.mandatory-path tso.cl*›

lemma *bind-commit-writes-absorbL*:
  fixes *P* :: ′*v raw.tso*
  shows *raw.commit-writes* ≫ *raw.tso.cl P* = *raw.tso.cl P*
by (*simp add*: *raw.tso.cl-def raw.commit-writes.commit-writes flip*: *raw.bind.bind*)

lemma *bind-commit-writes-absorb-unitR*:
  fixes *P* :: *unit raw.tso*
  shows *raw.tso.cl P* ≫ *raw.commit-writes* = *raw.tso.cl P*
by (*simp add*: *raw.tso.cl-def raw.bind.bind raw.commit-writes.commit-writes raw.bind.prim-returnR*)

lemma *bind-commit-writes-absorbR*:
  fixes *P* :: ′*v raw.tso*
  shows *raw.tso.cl P* ≫= (λ*v*. *raw.commit-writes* ≫ *raw.prim-return v*) = *raw.tso.cl P*
by (*simp add*: *raw.tso.cl-def raw.bind.bind raw.commit-writes.commit-writes raw.bind.prim-returnL*)
   (*simp add*: *raw.commit-writes.commit-writes flip*: *raw.bind.bind*)

lemma *bot*:
  shows *raw.tso.cl* ⊥ = *raw.commit-writes* ≫= ⊥
by (*simp add*: *raw.tso.cl-def raw.bind.bind raw.bind.botL flip*: *bot-fun-def*)

lemma *prim-return*:

shows *raw.tso.cl* (*raw.prim-return v*) = *raw.commit-writes* ≫ *raw.prim-return v*
by (*simp add*: *raw.tso.cl-def raw.bind.bind raw.bind.prim-returnL*)
(*simp add*: *raw.commit-writes.commit-writes flip*: *raw.bind.bind*)

lemma *Nil*:
shows *raw.tso.cl P* [] = *P* [] ≫= (λ*v*. *raw.commit-writes* (*snd v*) ≫= (λ*w*. *prog.return* (*fst v*, *snd w*)))
by (*simp add*: *raw.tso.cl-def raw.bind-def raw.prim-return-def raw.commit-writes.Nil prog.bind.returnL split-def*)

lemma *commit*:
fixes *wb* :: *write-buffer*
shows *raw.commit-write* [*w*] ≫ *f wb* ≤ *raw.tso.cl f* (*w* # *wb*)
apply (*simp add*: *raw.tso.cl-def raw.bind-def raw.prim-return-def split-def*)
apply (*strengthen ord-to-strengthen*[*OF raw.commit-writes.Cons-le*])
apply (*simp add*: *prog.bind.bind*)
apply (*rule prog.bind.mono*[*OF order.refl*])
apply (*strengthen ord-to-strengthen*[*OF raw.commit-writes.return-le'*])
apply (*simp add*: *prog.bind.returnL prog.bind.returnR*)
done

setup ‹*Sign.parent-path*›

interpretation *tso*: *closure-complete-distrib-lattice-distributive-class raw.tso.cl*
proof *standard*
show (*x* ≤ *raw.tso.cl y*) = (*raw.tso.cl x* ≤ *raw.tso.cl y*) **for** *x y* :: 'a *raw.tso*
proof(*intro iffD2*[*OF order-class.order.closure-axioms-alt-def*[*unfolded closure-axioms-def*], *rule-format*, *simplified conj-explode*] *allI*)
show *P* ≤ *raw.tso.cl P* **for** *P* :: 'a *raw.tso*
unfolding *raw.tso.cl-def*
by (*strengthen ord-to-strengthen*[*OF raw.commit-writes.return-le*])
(*simp add*: *raw.bind.prim-returnL raw.bind.prim-returnR*)
show *mono raw.tso.cl*
proof(*rule monotoneI*)
fix *P P'* :: 'v *raw.tso*
assume *P* ≤ *P'* **show** *raw.tso.cl P* ≤ *raw.tso.cl P'*
unfolding *raw.tso.cl-def* **by** (*strengthen ord-to-strengthen*(*1*)[*OF* ‹*P* ≤ *P'*›]) *simp*
qed
show *raw.tso.cl* (*raw.tso.cl P*) = *raw.tso.cl P* **for** *P* :: 'a *raw.tso*
by (*simp add*: *raw.tso.cl-def raw.bind.bind raw.commit-writes.commit-writes raw.bind.prim-returnL*)
(*simp add*: *raw.commit-writes.commit-writes flip*: *raw.bind.bind*)
qed
show *raw.tso.cl* (⨆ *X*) ≤ ⨆ (*raw.tso.cl* ‘ *X*) ⊔ *raw.tso.cl* ⊥ **for** *X* :: 'a *raw.tso set*
by (*simp add*: *raw.tso.cl-def raw.bind.bind raw.bind.botL flip*: *bot-fun-def raw.bind.SUPR raw.bind.SUPL*)
qed

setup ‹*Sign.mandatory-path tso.cl*›

lemma *bind*:
fixes *f* :: 'v *raw.tso*
assumes *f* ∈ *raw.tso.closed*
shows *raw.tso.cl* (*f* ≫= *g*) = *f* ≫= (λ*v*. *raw.tso.cl* (*g v*))
apply (*simp add*: *raw.tso.cl-def raw.bind.bind*)
apply (*subst* (*1 2*) *raw.tso.closed-conv*[*OF assms*(*1*)])
apply (*simp add*: *raw.tso.cl.bind-commit-writes-absorbL flip*: *raw.bind.bind*)
apply (*subst* (*1*) *raw.tso.cl.bind-commit-writes-absorbR*[*symmetric*])
apply (*simp add*: *raw.bind.bind raw.bind.prim-returnL*)
done

setup ‹*Sign.parent-path*›

setup ‹*Sign.mandatory-path bind*›

lemma *commit-writes-absorbL*:
  assumes $f \in$ *raw.tso.closed*
  shows *raw.commit-writes* $\gg f = f$
by (*metis assms raw.tso.closed-conv raw.tso.cl.bind-commit-writes-absorbL*)

lemma *commit-writes-absorb-unitR*:
  assumes $f \in$ *raw.tso.closed*
  shows $f \gg$ *raw.commit-writes* $= f$
by (*metis assms raw.tso.closed-conv raw.tso.cl.bind-commit-writes-absorb-unitR*)

lemma *returnL*:
  assumes $g\ v \in$ *raw.tso.closed*
  shows *raw.return* $v \gg\!= g = g\ v$
by (*simp add*: *assms raw.return-alt-def raw.bind.commit-writes-absorbL*
        raw.tso.cl.prim-return raw.bind.bind raw.bind.prim-returnL)

lemma *returnR*:
  assumes $f \in$ *raw.tso.closed*
  shows $f \gg\!=$ *raw.return* $= f$
by (*simp add*: *raw.return-alt-def raw.tso.cl.prim-return*
   raw.tso.cl.bind-commit-writes-absorbR[*of f*, *simplified raw.tso.closed-conv*[*OF assms*, *symmetric*]])

setup ‹*Sign.parent-path*›

setup ‹*Sign.mandatory-path tso.closed*›

lemma *commit-writes*:
  shows *raw.commit-writes* $\in$ *raw.tso.closed*
by (*rule raw.tso.closed-clI*)
   (*simp add*: *raw.tso.cl-def raw.commit-writes.commit-writes raw.bind.prim-returnR*
       flip: *raw.bind.bind*)

lemma *bind*[*intro*]:
  fixes $f :: 'v$ *raw.tso*
  fixes $g :: 'v \Rightarrow 'w$ *raw.tso*
  assumes $f \in$ *raw.tso.closed*
  assumes $\bigwedge x.\ g\ x \in$ *raw.tso.closed*
  shows $f \gg\!= g \in$ *raw.tso.closed*
by (*simp add*: *assms raw.tso.closed-clI raw.tso.cl.bind flip*: *raw.tso.closed-conv*)

lemma *action*[*intro*]:
  shows *raw.action F* $\in$ *raw.tso.closed*
by (*simp add*: *raw.action-def*)

lemma *guard*[*intro*]:
  shows *raw.guard g* $\in$ *raw.tso.closed*
by (*simp add*: *raw.guard-def raw.tso.closed.action*)

lemma *MFENCE*[*intro*]:
  shows *raw.MFENCE* $\in$ *raw.tso.closed*
by (*simp add*: *raw.MFENCE-def raw.tso.closed.guard*)

lemma *parallel*[*intro*]:
  assumes $P \in$ *raw.tso.closed*
  assumes $Q \in$ *raw.tso.closed*

shows *raw.parallel P Q ∈ raw.tso.closed*
apply (*rule raw.tso.closed-clI*)
apply (*clarsimp simp*: *raw.parallel-def raw.tso.cl-def raw.bind.prim-returnR le-fun-def*)
apply (*subst* (*2*) *raw.bind.commit-writes-absorbL*[*OF raw.tso.closed.MFENCE*, *symmetric*])
apply (*simp add*: *raw.bind-def split-def prog.bind.bind prog.bind.mono*[*OF order.refl*]
prog.bind.returnL raw.commit-writes.Nil)
done

lemma *vmap*[*intro*]:
assumes *P ∈ raw.tso.closed*
shows *raw.vmap vf P ∈ raw.tso.closed*
proof(*rule raw.tso.closed-clI*)
have *raw.tso.cl* (*raw.vmap vf P*) ≤ *raw.vmap vf* (*raw.tso.cl P*)
by (*simp add*: *le-funI raw.tso.cl-def raw.vmap-def raw.bind-def raw.prim-return-def split-def comp-def*
prog.vmap.eq-return prog.bind.bind prog.bind.returnL)
then show *raw.tso.cl* (*raw.vmap vf P*) ≤ *raw.vmap vf P*
by (*simp flip*: *raw.tso.closed-conv*[*OF assms*])
qed

setup ‹*Sign.parent-path*›

setup ‹*Sign.mandatory-path action*›

lemma *bot*:
shows *raw.action* ⊥ = *raw.tso.cl* ⊥
by (*simp add*: *raw.action-def prog.action.empty bot-fun-def*)

lemma *monotone*:
shows *mono raw.action*
unfolding *raw.action-def*
by (*fastforce simp*: *le-fun-def intro*: *monoI prog.action.mono raw.tso.mono-cl*)

lemmas *strengthen*[*strg*] = *st-monotone*[*OF raw.action.monotone*]
lemmas *mono* = *monotoneD*[*OF raw.action.monotone*]

lemma *Sup*:
shows *raw.action* (⨆ *Fs*) = ⨆ (*raw.action* ‘ *Fs*) ⊔ *raw.tso.cl* ⊥ (**is** *?lhs* = *?rhs*)
proof −
have *?rhs* = ⨆ (*raw.tso.cl* ‘ (λ*F wb*. *prog.action* {((*v*, *wb* @ *ws*), *s*, *s′*) |*v s s′ ws*. (*v*, *ws*, *s*, *s′*) ∈ *F wb*}) ‘ *Fs*)
⊔ *raw.tso.cl* ⊥
by (*simp add*: *raw.action-def image-comp*)
also have ... = *raw.tso.cl* (⨆ *F*∈*Fs*. (λ*wb*. *prog.action* {((*v*, *wb* @ *ws*), *s*, *s′*) |*v s s′ ws*. (*v*, *ws*, *s*, *s′*) ∈ *F wb*}))
by (*simp add*: *raw.tso.cl-Sup*)
also have ... = *raw.tso.cl* (λ*wb*. ⨆ (*prog.action* ‘ (λ*F*. {((*v*, *wb* @ *ws*), *s*, *s′*) |*v s s′ ws*. (*v*, *ws*, *s*, *s′*) ∈ *F wb*})
‘ *Fs*))
by (*simp add*: *Sup-fun-def image-comp*)
also have ... = *?lhs*
by (*force simp*: *raw.action-def*
simp flip: *prog.action.Sup*
intro: *arg-cong*[**where** *f*=*raw.tso.cl*] *arg-cong*[**where** *f*=*prog.action*])
finally show *?thesis* **..**
qed

lemma *sup*:
shows *raw.action* (*F* ⊔ *G*) = *raw.action F* ⊔ *raw.action G*
using *raw.action.Sup*[**where** *Fs*={*F*, *G*}]
by (*simp add*: *sup.absorb1 le-supI1 raw.action.mono flip*: *raw.action.bot*)

setup ‹*Sign.parent-path*›

setup ‹*Sign.mandatory-path guard*›

lemma *return-le*:
  shows *raw.guard g* ≤ *raw.return* ()
by (*fastforce simp add*: *raw.guard-def raw.return-def intro*: *le-funI raw.action.mono*)

lemma *monotone*:
  shows *mono* (*raw.guard* :: (*write-buffer* ⇒ *heap.t pred*) ⇒ -)
proof(*rule monoI*)
  show *raw.guard g* ≤ *raw.guard h* **if** *g* ≤ *h* **for** *g h* :: *write-buffer* ⇒ *heap.t pred*
    unfolding *raw.guard-def Diag-def*
    by (*blast intro*: *raw.action.mono le-funI predicate2D*[*OF that*])
qed

lemmas *strengthen*[*strg*] = *st-monotone*[*OF raw.guard.monotone*]
lemmas *mono* = *monotoneD*[*OF raw.guard.monotone*]

lemma *less*: — Non-triviality; essentially replay *prog.guard.less*
  assumes *g* < *g'*
  shows *raw.guard g* < *raw.guard g'*
proof(*rule le-neq-trans*)
  show *raw.guard g* ≤ *raw.guard g'*
    by (*strengthen ord-to-strengthen*(*1*)[*OF order-less-imp-le*[*OF assms*]]) *simp*
  from *assms* **obtain** *wb s* **where** *g' wb s* ¬*g wb s* **by** (*metis leD predicate2I*)
  from ‹¬*g wb s*› **have** ¬⦇*trace.T s* [] (*Some* ((), *wb*))⦈ ≤ *prog.p2s* (*raw.guard g wb*)
    by (*auto simp*: *raw.guard-def raw.action-def raw.tso.cl-def raw.bind-def raw.prim-return-def*
                split-def trace.split-all
                prog.p2s.simps prog.p2s.action prog.p2s.return
                spec.interference.cl.action spec.interference.cl.return
                spec.singleton.le-conv spec.singleton.action-le-conv trace.steps'.step-conv
                suffix-order.antisym-conv
          elim!: *spec.singleton.bind-le*
          dest!: *spec.singleton.raw.commit-writes.wb-suffix*)
  moreover
  from ‹*g' wb s*› **have** ⦇*trace.T s* [] (*Some* ((), *wb*))⦈ ≤ *prog.p2s* (*raw.guard g' wb*)
    by (*force simp*: *raw.guard-def raw.action-def raw.prim-return-def raw.tso.cl-def raw.bind-def*
                spec.bind.bind spec.singleton.le-conv
                prog.p2s.bind prog.p2s.action prog.p2s.return
                spec.interference.cl.action spec.interference.cl.return
           intro: *spec.bind.continueI*[**where** *xs*=[], *simplified*] *spec.action.stutterI*
                spec.singleton.raw.commit-writes.noop-le)
  ultimately show *raw.guard g* ≠ *raw.guard g'* **by** *metis*
qed

setup ‹*Sign.parent-path*›

lemma *MFENCE-alt-def*:
  shows *raw.MFENCE* = *raw.commit-writes* ≫ (λ*wb*. *prog.action* ({((), *wb*)} × *Diag* ⟨*wb* = []⟩))
proof −
  have ∗: *prog.action* {*x*. (∃ *a*. *x* = (((), *wb*), *a*, *a*)) ∧ *wb* = []} ≫= (λ*p*. *raw.commit-writes* (*snd p*))
        = *prog.action* ({((), *wb*)} × *Diag* (λ*s*. *wb* = [])) ≫= *prog.return*
   for *wb*
  proof(*induct rule*: *refinement.prog.eqI*[*case-names l2r r2l*])
    case *l2r* **show** *?case*
      apply (*rule refinement.prog.rev-bind*)
       apply (*rule refinement.prog.action*[**where** *Q*=λ*v s*. *snd v* = []];

```
        simp add: stable-def monotone-def; fail)
    apply (rule refinement.gen-asm; clarsimp simp: raw.commit-writes.Nil)
    apply (rule refinement.sort-of-refl)
     apply (subst refinement.top, simp; fail)
    apply (simp add: spec.idle.p2s-le; fail)
    done
  next
    case r2l show ?case
    apply (rule refinement.prog.rev-bind)
     apply (rule refinement.prog.action[where Q=λv s. snd v = []];
        simp add: stable-def monotone-def; fail)
    apply (rule refinement.gen-asm; clarsimp simp: raw.commit-writes.Nil)
    apply (rule refinement.sort-of-refl)
     apply (subst refinement.top, simp; fail)
    apply (simp add: spec.idle.p2s-le; fail)
    done
  qed
  show ?thesis
    by (simp add: raw.MFENCE-def raw.guard-def raw.action-def raw.tso.cl-def
          raw.bind.bind raw.bind.prim-returnR)
      (simp add: ∗ raw.bind-def raw.prim-return-def split-def prog.bind.return)
qed

setup ‹Sign.mandatory-path MFENCE›

lemma Nil:
  shows raw.MFENCE [] = prog.return ((), [])
by (simp add: raw.MFENCE-alt-def raw.bind-def raw.commit-writes.Nil prog.bind.returnL
      flip: Id-def prog.return-def)

setup ‹Sign.parent-path›

setup ‹Sign.parent-path›

setup ‹Sign.mandatory-path refinement.raw›

lemma MFENCE:
  shows prog.p2s (raw.MFENCE wb) ≤ {|P|}, A ⊨ prog.p2s (raw.MFENCE wb), {|λv s. snd v = []|}
apply (simp add: raw.MFENCE-alt-def raw.bind-def split-def)
apply (rule refinement.prog.rev-bind)
 apply (rule refinement.sort-of-refl)
 apply (subst refinement.top, simp; fail)
apply (rule refinement.prog.action; simp add: stable-def monotone-def)
done

setup ‹Sign.parent-path›

setup ‹Sign.mandatory-path raw›

setup ‹Sign.mandatory-path bind›

lemma MFENCEL:
  shows raw.MFENCE wb ≫= g = raw.MFENCE wb ≫ g ((), []) (is ?lhs = ?rhs)
proof(induct rule: refinement.prog.eqI[case-names l2r r2l])
  case l2r show ?case
    apply (rule refinement.prog.rev-bind)
     apply (rule refinement.raw.MFENCE)
    apply (rule refinement.gen-asm; clarsimp)
```

apply (*rule refinement.sort-of-refl*)
apply (*subst refinement.top*, *simp*; *fail*)
apply (*rule spec.idle.p2s-le*)
done
case *r2l* **show** *?case*
apply (*rule refinement.prog.rev-bind*)
apply (*rule refinement.raw.MFENCE*)
apply (*rule refinement.gen-asm*; *clarsimp*)
apply (*rule refinement.sort-of-refl*)
apply (*subst refinement.top*, *simp*; *fail*)
apply (*rule spec.idle.p2s-le*)
done
qed

lemma *MFENCE-return*:
shows *raw.MFENCE wb* ≫ *prog.return* ((), []) = *raw.MFENCE wb*
by (*simp add*: *prog.bind.returnR flip*: *raw.bind.MFENCEL*)

lemma *MFENCE-MFENCE*:
shows *raw.MFENCE* ≫ *raw.MFENCE* = *raw.MFENCE*
by (*simp add*: *raw.bind-def raw.prim-return-def raw.MFENCE.Nil raw.bind.MFENCE-return*
raw.bind.MFENCEL[**where** *g*=(λ(-, *y*). *raw.MFENCE y*)])

setup ‹*Sign.parent-path*›

setup ‹*Sign.mandatory-path t2p*›

lemma *bot*:
shows *raw.t2p* ⊥ = ⊥
by (*simp add*: *raw.t2p-def prog.bind.botL*)

lemma *cl-bot*:
shows *raw.t2p* (*raw.tso.cl* ⊥) = ⊥
by (*simp add*: *raw.t2p-def raw.tso.cl.bot raw.bind-def raw.commit-writes.Nil*
prog.bind.bind prog.bind.botL prog.bind.returnL)

lemma *monotone*:
shows *mono raw.t2p*
by (*rule monotoneI*) (*simp add*: *raw.t2p-def le-fun-def prog.bind.mono*)

lemmas *strengthen*[*strg*] = *st-monotone*[*OF raw.t2p.monotone*]
lemmas *mono* = *monotoneD*[*OF raw.t2p.monotone*]
lemmas *mono2mono*[*cont-intro*, *partial-function-mono*] = *monotone2monotone*[*OF raw.t2p.monotone*, *simplified*]

lemma *Sup*:
shows *raw.t2p* (⨆ *X*) = ⨆ (*raw.t2p* ‘ *X*)
by (*simp add*: *raw.t2p-def prog.bind.SUPL*)

lemma *sup*:
shows *raw.t2p* (*P* ⊔ *Q*) = *raw.t2p P* ⊔ *raw.t2p Q*
using *raw.t2p.Sup*[**where** *X*={*P*, *Q*}] **by** *simp*

lemma *mcont2mcont*[*cont-intro*]:
fixes *P* :: - ⇒ - *raw.tso*
assumes *mcont luba orda Sup* (≤) *F*
shows *mcont luba orda Sup* (≤) (λ*x*. *raw.t2p* (*F x*))
proof −
from *assms* **have** *mcont luba orda Sup* (≤) (λ*x*. *F x* [])

by (*fastforce intro*!: *mcontI contI monotoneI*
dest: *mcont-contD mcont-monoD*
simp: *le-funD*
simp flip: *SUP-apply*)
then show *?thesis*
by (*simp add*: *raw.t2p-def split-def*)
qed

lemma *return*:
shows *raw.t2p* (*raw.return v*) = *prog.return v*
by (*simp add*: *raw.t2p-def raw.return-alt-def raw.tso.cl-def raw.bind-def raw.prim-return-def*
prog.p2s.simps prog.p2s.return
spec.interference.cl.action spec.interference.cl.return
spec.bind.bind spec.bind.return
raw.commit-writes.Nil raw.MFENCE.Nil
flip: *prog.p2s-inject*)
(*simp add*: *spec.rel.wind-bind flip*: *spec.bind.bind*)

lemma *MFENCE-bind*:
shows *raw.t2p* (*raw.MFENCE* ⤜ *P*) = *raw.t2p* (*P* ())
by (*simp add*: *raw.t2p-def raw.bind-def split-def prog.bind.returnL raw.MFENCE.Nil*)

lemma *bind-return-unit*:
shows *raw.t2p* (λ*wb*. *prog.bind P* (λ-::*unit*. *prog.return* ((), []))) = *P*
by (*simp add*: *raw.t2p-def raw.bind-def split-def*
prog.bind.bind prog.bind.returnL prog.bind.returnR raw.MFENCE.Nil)

setup ‹*Sign.parent-path*›

setup ‹*Sign.mandatory-path parallel*›

lemma *commute*: — Jagadeesan et al. (2012, §5 (3))
shows *raw.parallel P Q* = *raw.parallel Q P*
by (*simp add*: *raw.parallel-def prog.parallel.commute*)

lemma *assoc*: — Jagadeesan et al. (2012, §5 (4))
shows *raw.parallel P* (*raw.parallel Q R*) = *raw.parallel* (*raw.parallel P Q*) *R*
by (*simp add*: *raw.parallel-def raw.t2p.MFENCE-bind raw.t2p.bind-return-unit prog.parallel.assoc*)

lemma *mono*:
assumes $P \leq P'$
assumes $Q \leq Q'$
shows *raw.parallel P Q* ≤ *raw.parallel P' Q'*
unfolding *raw.parallel-def*
apply (*strengthen ord-to-strengthen*(*1*)[*OF assms*(*1*)])
apply (*strengthen ord-to-strengthen*(*1*)[*OF assms*(*2*)])
apply (*rule order.refl*)
done

lemma *botL*:
shows *raw.parallel* (*raw.tso.cl* ⊥) *f* = *raw.MFENCE* ≫ *f* ≫ *raw.MFENCE* ≫ *raw.tso.cl* ⊥
apply (*simp add*: *raw.parallel-def raw.t2p.cl-bot prog.parallel.bot prog.bind.bind prog.bind.botL*)
apply (*simp add*: *raw.t2p-def split-def raw.bind-def prog.bind.bind prog.bind.returnL*)
apply (*subst* (*3 4*) *raw.bind.MFENCEL*)
apply (*simp add*: *raw.tso.cl.Nil prog.bind.botL*)
done

lemma *returnL*:

shows *raw.parallel* (*raw.return* ()) *P* = *raw.MFENCE* ⤜ (λ-. *P*) ⤜ (λ-. *raw.MFENCE*)
apply (*simp add*: *raw.parallel-def raw.t2p.return prog.parallel.return*)
apply (*simp add*: *raw.t2p-def split-def raw.bind-def prog.bind.bind prog.bind.returnL raw.bind.MFENCE-return*)
apply (*subst* (*2*) *raw.bind.MFENCEL*)
apply *simp*
done

lemma *SupL-not-empty*:
assumes ∀ *x*∈*X*. *x* ∈ *raw.tso.closed*
assumes *Q* ∈ *raw.tso.closed*
assumes *X* ≠ {}
shows *raw.parallel* (⨆ *X* ⊔ *raw.tso.cl* ⊥) *Q* = (⨆ *P*∈*X*. *raw.parallel P Q*) ⊔ *raw.tso.cl* ⊥
proof −
from ‹*X* ≠ {}›
have *raw.parallel* (⨆ *X*) *Q* = (⨆ *P*∈*X*. *raw.parallel P Q*)
by (*simp add*: *raw.parallel-def raw.t2p.Sup raw.t2p.sup*
prog.parallel.SupL-not-empty prog.parallel.supL prog.bind.SUPL prog.bind.supL)
(*simp add*: *raw.bind-def split-def fun-eq-iff prog.bind.SUPR-not-empty image-image*)
moreover
from *assms* **have** *raw.tso.cl* ⊥ ≤ (⨆ *P*∈*X*. *raw.parallel P Q*)
by (*force intro*: *less-eq-Sup raw.tso.least*[*OF* - *raw.tso.closed.parallel*])
moreover note *assms*
ultimately show *?thesis*
by (*simp add*: *sup.absorb1 less-eq-Sup raw.tso.least*)
qed

setup ‹*Sign.parent-path*›

setup ‹*Sign.parent-path*›

typedef *'v tso* = *raw.tso.closed* :: *'v raw.tso set*
morphisms *t2p'* *Abs-tso*
by *blast*

setup-lifting *type-definition-tso*

instantiation *tso* :: (*type*) *complete-distrib-lattice*
begin

lift-definition *bot-tso* :: *'v tso* **is** *raw.tso.cl* ⊥ **..**
lift-definition *top-tso* :: *'v tso* **is** ⊤ **..**
lift-definition *sup-tso* :: *'v tso* ⇒ *'v tso* ⇒ *'v tso* **is** *sup* **..**
lift-definition *inf-tso* :: *'v tso* ⇒ *'v tso* ⇒ *'v tso* **is** *inf* **..**
lift-definition *less-eq-tso* :: *'v tso* ⇒ *'v tso* ⇒ *bool* **is** *less-eq* **.**
lift-definition *less-tso* :: *'v tso* ⇒ *'v tso* ⇒ *bool* **is** *less* **.**
lift-definition *Inf-tso* :: *'v tso set* ⇒ *'v tso* **is** *Inf* **..**
lift-definition *Sup-tso* :: *'v tso set* ⇒ *'v tso* **is** λ*X*. *Sup X* ⊔ *raw.tso.cl* ⊥ **..**

instance by (*standard*; *transfer*; *auto simp*: *InfI Inf-lower le-supI1 SupI SupE raw.tso.least*)

end

setup ‹*Sign.mandatory-path tso*›

lift-definition *bind* :: *'v tso* ⇒ (*'v* ⇒ *'w tso*) ⇒ *'w tso* **is** *raw.bind* **..**
lift-definition *action* :: (*write-buffer* ⇒ (*'v* × *write-buffer* × *heap.t* × *heap.t*) *set*) ⇒ *'v tso* **is** *raw.action* **..**
lift-definition *MFENCE* :: *unit tso* **is** *raw.MFENCE* **..**
lift-definition *parallel* :: *unit tso* ⇒ *unit tso* ⇒ *unit tso* **is** *raw.parallel* **..**

lift-definition *vmap* :: $('v \Rightarrow 'w) \Rightarrow 'v\ tso \Rightarrow 'w\ tso$ **is** *raw.vmap* **..**

lift-definition *t2p* :: $'v\ tso \Rightarrow (heap.t, 'v)\ prog$ **is** *raw.t2p* **.**

adhoc-overloading
Monad-Syntax.bind $\rightleftharpoons$ *tso.bind*
adhoc-overloading
parallel $\rightleftharpoons$ *tso.parallel*

definition *return* :: $'v \Rightarrow 'v\ tso$ **where**
return v = *tso.action* $\langle\{v\} \times \{[]\} \times Id\rangle$

definition *guard* :: $(write\text{-}buffer \Rightarrow heap.t\ pred) \Rightarrow unit\ tso$ **where**
guard g = *tso.action* $(\lambda wb.\ \{()\} \times \{[]\} \times Diag\ (g\ wb))$

abbreviation (*input*) *read* :: $(heap.t \Rightarrow 'v) \Rightarrow 'v\ tso$ **where**
read f $\equiv$ *tso.action* $(\lambda wb.\ \{(f\ (apply\text{-}writes\ wb\ s),\ [],\ s,\ s)\ |s.\ True\})$

abbreviation (*input*) *write* :: $(heap.t \Rightarrow heap.write) \Rightarrow unit\ tso$ **where**
write f $\equiv$ *tso.action* $\langle\{((),\ [f\ s],\ s,\ s)\ |s.\ True\}\rangle$

lemma *return-alt-def*:
shows *tso.return v* = *tso.read* $\langle v\rangle$
by (*auto simp*: *tso.return-def intro*: *arg-cong*[**where** *f*=*tso.action*])

declare *tso.bind-def*[*code del*]
declare *tso.action-def*[*code del*]
declare *tso.return-def*[*code del*]
declare *tso.MFENCE-def*[*code del*]
declare *tso.parallel-def*[*code del*]
declare *tso.vmap-def*[*code del*]

setup ‹*Sign.mandatory-path return*›

lemma *transfer*[*transfer-rule*]:
shows *rel-fun* (=) *cr-tso raw.return tso.return*
unfolding *raw.return-def tso.return-def* **by** *transfer-prover*

setup ‹*Sign.parent-path*›

setup ‹*Sign.mandatory-path action*›

lemma *empty*:
shows *bot*: *tso.action* $\bot = \bot$
and *tso.action* $(\lambda\text{-}.\ \{\}) = \bot$
by (*simp-all add*: *raw.action.bot*[*transferred*, *unfolded bot-fun-def*] *bot-fun-def*)

lemmas *monotone* = *raw.action.monotone*[*transferred*]
lemmas *strengthen*[*strg*] = *st-monotone*[*OF tso.action.monotone*]
lemmas *mono* = *monotoneD*[*OF tso.action.monotone*]
lemmas *mono2mono*[*cont-intro*, *partial-function-mono*] = *monotone2monotone*[*OF tso.action.monotone*, *simplified*]

lemma *Sup*:
shows *tso.action* $(\bigsqcup Fs) = \bigsqcup (tso.action\ `\ Fs)$
by *transfer* (*simp add*: *raw.action.Sup*)

lemmas *sup* = *tso.action.Sup*[**where** *Fs*=$\{F,\ G\}$ **for** *F G*, *simplified*]

setup ‹*Sign.parent-path*›

setup ‹*Sign.mandatory-path bind*›

lemmas *if-distrL* = *if-distrib*[**where** *f*=λ*f*. *tso.bind f g* **for** *g*] — Jagadeesan et al. (2012, §5 (5))

lemmas *mono* = *raw.bind.mono*[*transferred*]

lemma *strengthen*[*strg*]:
  assumes *st-ord F f f′*
  assumes ⋀*x*. *st-ord F* (*g x*) (*g′ x*)
  shows *st-ord F* (*tso.bind f g*) (*tso.bind f′ g′*)
using *assms* **by** (*cases F*; *clarsimp intro*!: *tso.bind.mono*)

lemmas *mono2mono*[*cont-intro*, *partial-function-mono*] = *raw.bind.mono2mono*[*transferred*]

lemma *bind*: — Jagadeesan et al. (2012, §5 (2))
  shows *f* ⤜ *g* ⤜ *h* = *tso.bind f* (λ*x*. *g x* ⤜ *h*)
by *transfer* (*simp add*: *raw.bind.bind*)

lemma *return*: — Jagadeesan et al. (2012, §5 (1))
  shows *returnL*: *tso.return v* ⤜ *g* = *g v*
    and *returnR*: *f* ⤜ *tso.return* = *f*
by (*transfer*; *simp add*: *raw.bind.returnL raw.bind.returnR*)+

lemma *botL*:
  shows *tso.bind* ⊥ *g* = ⊥
by *transfer* (*simp add*: *raw.tso.cl.bot raw.bind.bind raw.bind.botL flip*: *bot-fun-def*)

lemma *botR-le*:
  shows *tso.bind f* ⟨⊥⟩ ≤ *f* (**is** *?thesis1*)
    and *tso.bind f* ⊥ ≤ *f* (**is** *?thesis2*)
proof −
  show *?thesis1*
    by (*metis bot.extremum dual-order.refl tso.bind.mono tso.bind.returnR*)
  then show *?thesis2*
    by (*simp add*: *bot-fun-def*)
qed

lemma
  fixes *f* :: - *tso*
  fixes *f*$_1$ :: - *tso*
  shows *supL*: (*f*$_1$ ⊔ *f*$_2$) ⤜ *g* = (*f*$_1$ ⤜ *g*) ⊔ (*f*$_2$ ⤜ *g*)
    and *supR*: *f* ⤜ (λ*x*. *g*$_1$ *x* ⊔ *g*$_2$ *x*) = (*f* ⤜ *g*$_1$) ⊔ (*f* ⤜ *g*$_2$)
by (*transfer*; *blast intro*: *raw.bind.supL raw.bind.supR*)+

lemma *SUPL*:
  fixes *X* :: - *set*
  fixes *f* :: - ⇒ - *tso*
  shows (⨆*x*∈*X*. *f x*) ⤜ *g* = (⨆*x*∈*X*. *f x* ⤜ *g*)
by *transfer*
  (*simp add*: *raw.bind.supL raw.bind.SUPL raw.tso.cl.bot raw.bind.bind raw.bind.botL*
      flip: *bot-fun-def*)

lemma *SUPR*:
  fixes *X* :: - *set*
  fixes *f* :: - *tso*

shows $f \gg\!= (\lambda v.\ \bigsqcup x{\in}X.\ g\ x\ v) = (\bigsqcup x{\in}X.\ f \gg\!= g\ x) \sqcup (f \gg\!= \bot)$
unfolding *bot-fun-def*
by *transfer*
(*simp add*: *raw.bind.supR raw.bind.SUPR ac-simps*
sup.absorb2 le-supI1 raw.bind.mono raw.tso.closed.bind raw.tso.least)

lemma *SupR*:
fixes X :: - *set*
fixes f :: - *tso*
shows $f \gg (\bigsqcup X) = (\bigsqcup x{\in}X.\ f \gg x) \sqcup (f \gg\!= \bot)$
by (*simp add*: *tso.bind.SUPR*[**where** $g{=}\lambda x\ v.\ x$, *simplified*])

lemma *SUPR-not-empty*:
fixes f :: - *tso*
assumes $X \neq \{\}$
shows $f \gg\!= (\lambda v.\ \bigsqcup x{\in}X.\ g\ x\ v) = (\bigsqcup x{\in}X.\ f \gg\!= g\ x)$
using *iffD2*[*OF ex-in-conv assms*]
by (*subst trans*[*OF tso.bind.SUPR sup.absorb1*]; *force intro*: *SUPI tso.bind.mono*)

lemma *mcont2mcont*[*cont-intro*]:
assumes *mcont luba orda Sup* $(\leq)$ f
assumes $\bigwedge v.$ *mcont luba orda Sup* $(\leq)$ $(\lambda x.\ g\ x\ v)$
shows *mcont luba orda Sup* $(\leq)$ $(\lambda x.$ *tso.bind* $(f\ x)\ (g\ x))$
proof(*rule ccpo.mcont2mcont'*[*OF complete-lattice-ccpo - - assms*(*1*)])
show *mcont Sup* $(\leq)$ *Sup* $(\leq)$ $(\lambda f.$ *tso.bind* $f\ (g\ x))$ **for** x
by (*intro mcontI contI monotoneI*) (*simp-all add*: *tso.bind.mono flip*: *tso.bind.SUPL*)
show *mcont luba orda Sup* $(\leq)$ $(\lambda x.$ *tso.bind* $f\ (g\ x))$ **for** f
by (*intro mcontI monotoneI contI*)
(*simp-all add*: *mcont-monoD*[*OF assms*(*2*)] *tso.bind.mono*
flip: *tso.bind.SUPR-not-empty contD*[*OF mcont-cont*[*OF assms*(*2*)]])
qed

setup ‹*Sign.parent-path*›

setup ‹*Sign.mandatory-path guard*›

lemma *transfer*[*transfer-rule*]:
shows *rel-fun* (=) *cr-tso raw.guard tso.guard*
unfolding *raw.guard-def tso.guard-def* **by** *transfer-prover*

lemma *bot*:
shows *tso.guard* $\bot = \bot$
and *tso.guard* $(\lambda$- - .*False*$) = \bot$
by (*simp-all add*: *tso.guard-def tso.action.empty*)

lemma *top*:
shows *tso.guard* $\top$ = *tso.return* () (**is** *?thesis1*)
and *tso.guard* $(\lambda$-. $\top) =$ *tso.return* () (**is** *?thesis2*)
and *tso.guard* $(\lambda$- -. *True*$) =$ *tso.return* () (**is** *?thesis3*)
proof −
show *?thesis1*
by (*simp add*: *tso.guard-def tso.return-def flip*: *Id-def*)
then show *?thesis2* **and** *?thesis3*
by (*simp-all add*: *top-fun-def*)
qed

lemma *return-le*:
shows *tso.guard* $g \leq$ *tso.return* ()

by *transfer* (*rule raw.guard.return-le*)

lemma *monotone*:
  shows *mono tso.guard*
by *transfer* (*rule raw.guard.monotone*)

lemmas *strengthen*[*strg*] = *st-monotone*[*OF tso.guard.monotone*]
lemmas *mono* = *monotoneD*[*OF tso.guard.monotone*]
lemmas *mono2mono*[*cont-intro*, *partial-function-mono*] = *monotone2monotone*[*OF tso.guard.monotone*, *simplified*]

lemma *less*: — Non-triviality
  assumes $g < g'$
  shows *tso.guard g* < *tso.guard g'*
using *assms* **by** *transfer* (*rule raw.guard.less*)

setup ‹*Sign.parent-path*›

setup ‹*Sign.mandatory-path parallel*›

lemma *commute*: — Jagadeesan et al. (2012, §5 (3))
  shows *tso.parallel P Q* = *tso.parallel Q P*
by *transfer* (*rule raw.parallel.commute*)

lemma *assoc*: — Jagadeesan et al. (2012, §5 (4))
  shows *tso.parallel P* (*tso.parallel Q R*) = *tso.parallel* (*tso.parallel P Q*) *R*
by *transfer* (*rule raw.parallel.assoc*)

lemmas *mono* = *raw.parallel.mono*[*transferred*]

lemma *strengthen*[*strg*]:
  assumes *st-ord F P P'*
  assumes *st-ord F Q Q'*
  shows *st-ord F* (*tso.parallel P Q*) (*tso.parallel P' Q'*)
using *assms* **by** (*cases F*; *simp add*: *tso.parallel.mono*)

lemma *mono2mono*[*cont-intro*, *partial-function-mono*]:
  assumes *monotone orda* (≤) *F*
  assumes *monotone orda* (≤) *G*
  shows *monotone orda* (≤) (λ*f*. *tso.parallel* (*F f*) (*G f*))
using *assms* **by** (*simp add*: *monotone-def tso.parallel.mono*)

lemma *bot*:
  shows *parallel-botL*: *tso.parallel* ⊥ *f* = *tso.MFENCE* ≫ *f* ≫ *tso.MFENCE* ≫= ⊥ (**is** *?thesis1*)
    and *parallel-botR*: *tso.parallel f* ⊥ = *tso.MFENCE* ≫ *f* ≫ *tso.MFENCE* ≫= ⊥ (**is** *?thesis2*)
proof −
  show *?thesis1*
    unfolding *bot-fun-def* **by** *transfer* (*simp add*: *raw.parallel.botL raw.bind.bind*)
  then show *?thesis2*
    by (*simp add*: *tso.parallel.commute*)
qed

lemma *return*: — Jagadeesan et al. (2012, unnumbered)
  shows *returnL*: *tso.return* () ∥ *P* = *tso.MFENCE* ≫ *P* ≫ *tso.MFENCE* (**is** *?thesis1*)
    and *returnR*: *P* ∥ *tso.return* () = *tso.MFENCE* ≫ *P* ≫ *tso.MFENCE* (**is** *?thesis2*)
proof −
  show *?thesis1*
    by *transfer* (*rule raw.parallel.returnL*)

then show *?thesis2*
by (*simp add*: *tso.parallel.commute*)
qed

lemma *Sup-not-empty*:
fixes X :: *unit tso set*
assumes $X \neq \{\}$
shows *SupL-not-empty*: $\bigsqcup X \parallel Q = (\bigsqcup P{\in}X.\ P \parallel Q)$ (**is** *?thesis1 Q*)
and *SupR-not-empty*: $P \parallel \bigsqcup X = (\bigsqcup Q{\in}X.\ P \parallel Q)$ (**is** *?thesis2*)
proof −
from *assms* **show** *?thesis1 Q* **for** Q
by *transfer* (*rule raw.parallel.SupL-not-empty*)
then show *?thesis2*
by (*simp add*: *tso.parallel.commute*)
qed

lemma *sup*:
fixes P :: *unit tso*
shows *supL*: $P \sqcup Q \parallel R = (P \parallel R) \sqcup (Q \parallel R)$
and *supR*: $P \parallel Q \sqcup R = (P \parallel Q) \sqcup (P \parallel R)$
using *tso.parallel.SupL-not-empty*[**where** $X{=}\{P,\ Q\}$] *tso.parallel.SupR-not-empty*[**where** $X{=}\{Q,\ R\}$]
by *simp-all*

lemma *mcont2mcont*[*cont-intro*]:
assumes *mcont luba orda Sup* $(\leq)$ P
assumes *mcont luba orda Sup* $(\leq)$ Q
shows *mcont luba orda Sup* $(\leq)$ $(\lambda x.\ tso.parallel\ (P\ x)\ (Q\ x))$
proof(*rule ccpo.mcont2mcont'*[*OF complete-lattice-ccpo - - assms*(*1*)])
show *mcont Sup* $(\leq)$ *Sup* $(\leq)$ $(\lambda y.\ tso.parallel\ y\ (Q\ x))$ **for** x
by (*intro mcontI contI monotoneI*) (*simp-all add*: *tso.parallel.mono tso.parallel.SupL-not-empty*)
show *mcont luba orda Sup* $(\leq)$ $(\lambda x.\ tso.parallel\ y\ (Q\ x))$ **for** y
by (*simp add*: *mcontI monotoneI contI mcont-monoD*[*OF assms*(*2*)]
spec.parallel.mono mcont-contD[*OF assms*(*2*)] *tso.parallel.SupR-not-empty image-image*)
qed

setup ‹*Sign.parent-path*›

setup ‹*Sign.mandatory-path bind*›

lemmas *MFENCE-MFENCE* = *raw.bind.MFENCE-MFENCE*[*transferred*]

setup ‹*Sign.parent-path*›

setup ‹*Sign.mandatory-path t2p'*›

lemma *monotone*:
shows *mono* $(\lambda t.\ t2p'\ t\ wb)$
by (*simp add*: *le-fun-def less-eq-tso.rep-eq monotone-def*)

lemmas *strengthen*[*strg*] = *st-monotone*[*OF tso.t2p'.monotone*]
lemmas *mono* = *monotoneD*[*OF tso.t2p'.monotone*]

lemmas *action* = *tso.action.rep-eq*
lemma *return*:
shows $t2p'\ (tso.return\ v) = raw.return\ v$
by *transfer simp*

setup ‹*Sign.parent-path*›

setup ‹*Sign.parent-path*›

Combinators **setup** ‹*Sign.mandatory-path tso*›

abbreviation *guardM* :: *bool* ⇒ *unit tso* **where**
guardM b ≡ *if b then* ⊥ *else tso.return* ()

abbreviation *unlessM* :: *bool* ⇒ *unit tso* ⇒ *unit tso* **where**
unlessM b c ≡ *if b then tso.return* () *else c*

abbreviation *whenM* :: *bool* ⇒ *unit tso* ⇒ *unit tso* **where**
whenM b c ≡ *if b then c else tso.return* ()

definition *app* :: (*'a* ⇒ *unit tso*) ⇒ *'a list* ⇒ *unit tso* **where** — Haskell's *mapM-*
app f xs = *foldr* (λ*x m*. *f x* ≫ *m*) *xs* (*tso.return* ())

primrec *fold-mapM* :: (*'a* ⇒ *'b tso*) ⇒ *'a list* ⇒ *'b list tso* **where**
fold-mapM f [] = *tso.return* []
| *fold-mapM f* (*x* # *xs*) = *do* {
y ← *f x*;
ys ← *fold-mapM f xs*;
tso.return (*y* # *ys*)
}

— Jagadeesan et al. (2012, §5 (6) is *tso.while.simps*)
partial-function (*lfp*) *while* :: (*'k* ⇒ (*'k* + *'v*) *tso*) ⇒ *'k* ⇒ *'v tso* **where**
while c k = *c k* ≫= (λ*rv*. *case rv of Inl k'* ⇒ *while c k'* | *Inr v* ⇒ *tso.return v*)

abbreviation (*input*) *while'* :: ((*unit* + *'v*) *tso*) ⇒ *'v tso* **where**
while' c ≡ *tso.while* ⟨*c*⟩ ()

definition *raise* :: *String.literal* ⇒ *'v tso* **where**
raise s = ⊥

definition *assert* :: *bool* ⇒ *unit tso* **where**
assert P = (*if P then tso.return* () *else tso.raise STR ''assert''*)

declare *tso.raise-def*[*code del*]

setup ‹*Sign.mandatory-path fold-mapM*›

lemma *bot*:
shows *tso.fold-mapM* ⊥ = (λ*xs*. *case xs of* [] ⇒ *tso.return* [] | - ⇒ ⊥)
by (*simp add*: *fun-eq-iff tso.bind.botL split*: *list.split*)

lemma *append*:
shows *tso.fold-mapM f* (*xs* @ *ys*) = *tso.fold-mapM f xs* ≫= (λ*xs*. *tso.fold-mapM f ys* ≫= (λ*ys*. *tso.return* (*xs* @ *ys*)))
by (*induct xs*) (*simp-all add*: *tso.bind.bind tso.bind.returnL tso.bind.returnR*)

setup ‹*Sign.parent-path*›

setup ‹*Sign.mandatory-path app*›

lemma *bot*:
shows *tso.app* ⊥ = (λ*xs*. *case xs of* [] ⇒ *tso.return* () | - ⇒ ⊥)
and *tso.app* (λ-. ⊥) = (λ*xs*. *case xs of* [] ⇒ *tso.return* () | - ⇒ ⊥)

by (*simp-all add*: *fun-eq-iff tso.app-def tso.bind.botL split*: *list.split*)

lemma *Nil*:
  shows *tso.app f* [] = *tso.return* ()
by (*simp add*: *tso.app-def*)

lemma *Cons*:
  shows *tso.app f* (*x* # *xs*) = *f x* ≫ *tso.app f xs*
by (*simp add*: *tso.app-def*)

lemmas *simps* =
  tso.app.bot
  tso.app.Nil
  tso.app.Cons

lemma *append*:
  shows *tso.app f* (*xs* @ *ys*) = *tso.app f xs* ≫ *tso.app f ys*
by (*induct xs arbitrary*: *ys*) (*simp-all add*: *tso.app.simps tso.bind.returnL tso.bind.bind*)

lemma *monotone*:
  shows *mono* (λ*f*. *tso.app f xs*)
by (*induct xs*) (*simp-all add*: *tso.app.simps le-fun-def monotone-on-def tso.bind.mono*)

lemmas *strengthen*[*strg*] = *st-monotone*[*OF tso.app.monotone*]
lemmas *mono* = *monotoneD*[*OF tso.app.monotone*]
lemmas *mono2mono*[*cont-intro*, *partial-function-mono*] = *monotone2monotone*[*OF tso.app.monotone*, *simplified*, *of orda P* **for** *orda P*]

lemma *Sup-le*:
  shows (⨆*f*∈*X*. *tso.app f xs*) ≤ *tso.app* (⨆ *X*) *xs*
by (*simp add*: *SUP-le-iff SupI tso.app.mono*)

setup ‹*Sign.parent-path*›

setup ‹*Sign.parent-path*›

27.1 References

Observe that allocation is global in this model. We allow the memory location to have an arbitrary value and enqueue the initialising write in the TSO buffer.

setup ‹*Sign.mandatory-path tso.Ref*›

definition *ref* :: *'a*::*heap.rep* ⇒ *'a ref tso* **where**
  ref v = *tso.action* (λ*wb*. {(*r*, [*heap.Write* (*ref.addr-of r*) *0* (*heap.rep.to v*)], *s*, *s'*)
                    |*r s s' v'*. (*r*, *s'*) ∈ *Ref.alloc v' s*})

definition *lookup* :: *'a*::*heap.rep ref* ⇒ *'a tso* (‹!-› *61*) **where**
  lookup r = *tso.read* (*Ref.get r*)

definition *update* :: *'a ref* ⇒ *'a*::*heap.rep* ⇒ *unit tso* (‹- := -› *62*) **where**
  update r v = *tso.write* ⟨*heap.Write* (*ref.addr-of r*) *0* (*heap.rep.to v*)⟩

declare *tso.Ref.ref-def*[*code del*]
declare *tso.Ref.lookup-def*[*code del*]
declare *tso.Ref.update-def*[*code del*]

setup ‹*Sign.parent-path*›

27.2 Inhabitation

In order to obtain compositional rules we need to make the write buffer explicit.

setup ‹*Sign.mandatory-path tso*›

definition *t2s* :: *write-buffer* ⇒ *'v tso* ⇒ (*sequential*, *heap.t*, *'v* × *write-buffer*) *spec* **where**
t2s wb P = *prog.p2s* (*tso.t2p'* *P wb*)

setup ‹*Sign.parent-path*›

setup ‹*Sign.mandatory-path spec.singleton.tso*›

lemma *t2s-commit*:
assumes ⟪*heap.apply-write w s*, *xs*, *v*⟫ ≤ *tso.t2s wb f*
shows ⟪*s*, (*self*, *heap.apply-write w s*) # *xs*, *v*⟫ ≤ *tso.t2s* (*w* # *wb*) *f*
unfolding *tso.t2s-def*
by (*subst raw.tso.closed-conv*[*OF tso.t2p'*])
(*fastforce simp*: *prog.p2s.action simp add*: *prog.p2s.simps simp flip*: *tso.t2s-def*
intro: *order.trans*[*OF - prog.p2s.mono*[*OF raw.tso.cl.commit*]]
spec.bind.continueI[**where** *xs*=[(*self*, *heap.apply-write w s*)] **and** *v*=((), []), *simplified*, *OF - assms*]
order.trans[*OF spec.action.stepI spec.interference.expansive*])

setup ‹*Sign.parent-path*›

setup ‹*Sign.mandatory-path spec.idle.tso*›

lemma *t2s-le*:
shows *spec.idle* ≤ *tso.t2s wb P*
by (*simp add*: *tso.t2s-def spec.idle.p2s-le*)

setup ‹*Sign.parent-path*›

setup ‹*Sign.mandatory-path spec.t2s*›

lemmas *minimal*[*iff*] = *order.trans*[*OF spec.idle.minimal-le spec.idle.tso.t2s-le*]

setup ‹*Sign.parent-path*›

setup ‹*Sign.mandatory-path spec.interference.tso*›

lemma *t2s-le*:
shows *spec.rel* ({*env*} × *UNIV*) ≫= (λ-::*unit*. *tso.t2s wb P*) ≤ *tso.t2s wb P*
by (*simp add*: *tso.t2s-def prog.p2s.interference-wind-bind*)

setup ‹*Sign.parent-path*›

setup ‹*Sign.mandatory-path prog.p2s*›

lemma *t2p*[*prog.p2s.simps*]:
shows *prog.p2s* (*tso.t2p P*)
= *tso.t2s* [] *P* ≫= (λ*vwb*. *prog.p2s* (*raw.MFENCE* (*snd vwb*) ≫ *prog.return* (*fst vwb*)))
by *transfer* (*simp add*: *tso.t2p-def tso.t2s-def raw.t2p-def prog.p2s.simps split-def*)

setup ‹*Sign.parent-path*›

setup ‹*Sign.mandatory-path tso.t2s*›

lemma *bind*:

shows *tso.t2s wb* (*f* ⤜ *g*) = *tso.t2s wb f* ⤜ (λ*x*. *tso.t2s* (*snd x*) (*g* (*fst x*)))
unfolding *tso.t2s-def* **by** *transfer* (*simp add*: *raw.bind-def split-def prog.p2s.simps*)

lemma *parallel*:
shows *tso.t2s* [] (*P* ∥ *Q*) = *prog.p2s* ((*tso.t2p P* ∥ *tso.t2p Q*) ⟫ *prog.return* ((), []))
unfolding *tso.t2s-def*
by *transfer* (*simp add*: *raw.parallel-def raw.bind-def raw.MFENCE.Nil prog.bind.returnL*)

lemma *return*:
shows *tso.t2s* [] (*tso.return v*) = *prog.p2s* (*prog.return* (*v*, []))
unfolding *tso.t2s-def*
by *transfer*
(*simp add*: *raw.return-alt-def raw.tso.cl.Nil raw.prim-return-def prog.bind.returnL raw.commit-writes.Nil*)

setup ‹*Sign.parent-path*›

Inhabitation rules. **setup** ‹*Sign.mandatory-path inhabits.tso*›

lemma *bind*:
assumes *tso.t2s wb f* −*s*, *xs*→ *tso.t2s wb*′ *f*′
shows *tso.t2s wb* (*f* ⤜ *g*) −*s*, *xs*→ *tso.t2s wb*′ (*f*′ ⤜ *g*)
by (*simp add*: *tso.t2s.bind inhabits.spec.bind assms*)

lemma *commit*:
shows *tso.t2s* (*w* # *wb*) *f* −*s*, [(*self*, *heap.apply-write w s*)]→ *tso.t2s wb f*
by (*clarsimp simp*: *inhabits-def spec.bind.singletonL spec.term.none.singleton trace.split-all*
spec.singleton.tso.t2s-commit)

setup ‹*Sign.mandatory-path Ref*›

lemma *ref*:
fixes *r* :: ′*a*::*heap.rep ref*
fixes *s* :: *heap.t*
fixes *v* :: ′*a*
fixes *v*′ :: ′*a*
assumes ¬*heap.present r s*
shows *tso.t2s wb* (*tso.Ref.ref v*)
−*s*, [(*self*, *Ref.set r v*′ *s*)]→
tso.t2s (*wb* @ [*heap.Write* (*ref.addr-of r*) *0* (*heap.rep.to v*)]) (*tso.return r*) (**is** *?lhs* −*s*, *?step*→ *?rhs*)
proof −
have *rhs*: *?rhs* = *prog.p2s* (*raw.commit-writes* (*wb* @ [*heap.Write* (*ref.addr-of r*) *0* (*heap.rep.to v*)])
⤜ (λ*v*. *raw.prim-return r* (*snd v*)))
apply (*simp add*: *tso.t2s-def tso.t2p*′*.return raw.return-def raw.action-def*)
apply (*subst* (*1*) *prog.return.cong*)
apply (*simp-all add*: *image-iff split-def Sup-fst fst-image raw.tso.cl.prim-return raw.bind-def*
flip: *raw.prim-return-def*)
done
note ∗ = *order.trans*[*OF* - *spec.bind.mono*[*OF prog.p2s.mono*[*OF*
raw.commit-writes.return-le[*unfolded le-fun-def raw.prim-return-def*, *rule-format*]]
order.refl]]
note ∗∗ = *spec.bind.mono*[*OF spec.action.stepI*[**where** *a*=*self* **and** *s*=*s*
and *v*=(*r*, *wb* @ [*heap.Write* (*ref.addr-of r*) *0* (*heap.rep.to v*)])
and *s*′=*Ref.set r v*′ *s*
and *w*=*Some* (*r*, *wb* @ [*heap.Write* (*ref.addr-of r*) *0* (*heap.rep.to v*)])]
order.refl]
have *lhs*: ⦇*s*, [(*self*, *Ref.set r v*′ *s*)], *Some* (*r*, *wb* @ [*heap.Write* (*ref.addr-of r*) *0* (*heap.rep.to v*)])⦈
⤜ (λ*v*. *prog.p2s* (*raw.commit-writes* (*snd v*))
⤜ (λ*x*. *prog.p2s* (*raw.prim-return* (*fst v*) (*snd x*))))

```
      ≤ ?lhs
    apply (simp add: tso.Ref.ref-def tso.t2s-def split-def tso.t2p'.action
                     raw.action-def raw.tso.cl-def raw.bind-def prog.p2s.bind prog.bind.bind)
    apply (rule * )
    apply (simp flip: prog.p2s.bind)
    apply (force simp: assms Ref.alloc-def prog.p2s.simps prog.p2s.action prog.bind.returnL
                intro: ** order.trans[OF - spec.bind.mono[OF spec.interference.expansive order.refl]])
    done
  show ?thesis
    unfolding inhabits-def
    by (rule order.trans[OF - lhs])
       (simp add: rhs prog.p2s.simps spec.bind.singletonL spec.term.none.singleton)
qed

lemma lookup:
  fixes r :: 'a::heap.rep ref
  shows tso.t2s wb (!r) −s, []→ tso.t2s wb (tso.return (Ref.get r (apply-writes wb s)))
apply (clarsimp simp: tso.Ref.lookup-def  inhabits-def trace.split-all
                      tso.t2s-def tso.t2p'.action tso.t2p'.return
                      raw.action-def raw.return-alt-def raw.tso.cl.prim-return
                      spec.bind.singletonL spec.term.none.singleton)
apply (clarsimp simp: raw.tso.cl-def raw.bind-def split-def prog.bind.bind)
apply (rule order.trans[OF - prog.p2s.mono[OF
  prog.bind.mono[OF raw.commit-writes.return-le[unfolded raw.prim-return-def le-fun-def, rule-format]
                    order.refl]]])
apply (force simp: prog.bind.return prog.p2s.bind prog.p2s.action
     intro: order.trans[OF spec.bind.continueI[where xs=[], simplified, OF spec.action.stutterI]
  spec.bind.mono[OF spec.interference.expansive order.refl]])
done

lemma update:
  fixes r :: 'a::heap.rep ref
  shows tso.t2s wb (r := v)
        −s, []→
          tso.t2s (wb @ [heap.Write (ref.addr-of r) 0 (heap.rep.to v)]) (tso.return ())
proof −
  have *: (λp. raw.prim-return () (snd p)) = prog.return
    by (simp add: raw.prim-return-def fun-eq-iff)
  have raw.tso.cl (λwb. prog.return ((), wb)) (wb @ [heap.Write (ref.addr-of r) 0 (heap.rep.to v)])
    ≤ raw.tso.cl (λwb. prog.action {((((), wb @ [heap.Write (ref.addr-of r) 0 (heap.rep.to v)]), s, s) |s. True}) wb
    — LHS
    apply (simp add: raw.tso.cl-def raw.bind.prim-returnR raw.commit-writes.commit-writes
                flip: raw.prim-return-def
                cong: order.assms-cong)
    — RHS
    apply (simp add: * raw.tso.cl-def raw.bind-def split-def prog.bind.bind)
    apply (rule order.trans[OF -
           prog.bind.mono[OF raw.commit-writes.return-le[unfolded raw.prim-return-def le-fun-def, rule-format]
                             order.refl]])
    apply (simp add: prog.bind.returnL flip: prog.p2s.bind)
    apply (subst (1) prog.return.cong, force, force)
    apply (simp add: split-def Sup-fst fst-image prog.bind.return)
    done
  from prog.p2s.mono[OF this] show ?thesis
    by (fastforce simp: tso.Ref.update-def raw.action-def tso.t2s-def inhabits-def
                        tso.t2p'.return tso.t2p'.action
                        raw.return-alt-def raw.prim-return-def
                        spec.bind.singletonL spec.term.none.singleton)
```

qed

setup ‹*Sign.parent-path*›

lemmas *bind′* = *inhabits.trans*[*OF inhabits.tso.bind*]
lemmas *commit′* = *inhabits.trans*[*OF inhabits.tso.commit*]

setup ‹*Sign.parent-path*›

27.3 Code generator setup for TSO

The following is only sound if the generated code runs on a machine with a TSO memory model such as:

- x86
- x86 code running on macOS under Rosetta 2 (ask Google)

Notes:

- Haskell: GHC exposes unfenced operations for references and some kinds of arrays
  - GHC has a zoo of arrays; for now we use the general but inefficient boxed array type
- SML: Poly/ML appears to have committed to release/acquire (see email with subject "Git master update: ARM64, PIE and new bootstrap process")
  - on x86 this is TSO
- Scala: beyond the scope of this work

TODO:

- support a CAS-like operation
  - Haskell: https://stackoverflow.com/questions/10102881/haskell-how-does-atomicmodifyioref-work

27.3.1 Haskell

Adaption layer

code-printing code-module *TSOHeap* ⇀ (*Haskell*)
‹
module TSOHeap (
    TSO
  , *IORef*, *newIORef*, *readIORef*, *writeIORef*
  , *Array*, *newArray*, *newListArray*, *newFunArray*, *lengthArray*, *readArray*, *writeArray*
  , *parallel*
) *where*

import Control.Concurrent (*forkIO*)
import qualified Control.Concurrent.MVar as MVar
import qualified Data.Array.IO as Array −− FIXME boxed, contemplate the menagerie of other arrays; perhaps type families might help here
import Data.IORef (*IORef*, *newIORef*, *readIORef*, *writeIORef*)
import Data.List (*genericLength*)

type TSO a = IO a
type Array a = Array.IOArray Integer a
type Ref a = Data.IORef.IORef a

```
writeIORef :: IORef a -> a -> IO ()
writeIORef = writeIORef -- FIXME strict variant?

newArray :: Integer -> a -> IO (Array a)
newArray k = Array.newArray (0, k - 1)

newListArray :: [a] -> IO (Array a)
newListArray xs = Array.newListArray (0, genericLength xs - 1) xs

newFunArray :: Integer -> (Integer -> a) -> IO (Array a)
newFunArray k f = Array.newListArray (0, k - 1) (map f [0..k-1])

lengthArray :: Array a -> IO Integer
lengthArray a = Array.getBounds a >>= return . (\(-, l) -> l + 1)

readArray :: Array a -> Integer -> IO a
readArray = Array.readArray

writeArray :: Array a -> Integer -> a -> IO ()
writeArray = Array.writeArray

-- note we don't want forkFinally as we don't model exceptions
parallel :: IO () -> IO () -> IO ()
parallel p q = do
  mvar <- MVar.newEmptyMVar
  forkIO (p >> MVar.putMVar mvar ()) -- FIXME putMVar is lazy
  b <- q
  a <- MVar.takeMVar mvar
  return ()
›
```

code-reserved (*Haskell*) *TSOHeap*

Monad

code-printing type-constructor *tso* ⇀ (*Haskell*) *TSOHeap.TSO* -
code-monad *tso.bind Haskell*
code-printing constant *tso.return* ⇀ (*Haskell*) *return*
code-printing constant *tso.raise* ⇀ (*Haskell*) *error*
code-printing constant *tso.parallel* ⇀ (*Haskell*) *TSOHeap.parallel*

Intermediate operation avoids invariance problem in *Scala* (similar to value restriction)

setup ‹*Sign.mandatory-path tso.Ref*›

definition *ref'* **where**
  [*code del*]: *ref' = tso.Ref.ref*

lemma [*code*]:
  tso.Ref.ref x = tso.Ref.ref' x
  by (*simp add*: *tso.Ref.ref'-def*)

setup ‹*Sign.parent-path*›

Haskell

code-printing type-constructor *ref* ⇀ (*Haskell*) *TSOHeap.Ref* -
code-printing constant *Ref* ⇀ (*Haskell*) *error/ bare Ref*
code-printing constant *tso.Ref.ref'* ⇀ (*Haskell*) *TSOHeap.newIORef*
code-printing constant *tso.Ref.lookup* ⇀ (*Haskell*) *TSOHeap.readIORef*

code-printing constant *tso.Ref.update* ⇀ (*Haskell*) *TSOHeap.writeIORef*
code-printing constant *HOL.equal* :: *'a ref* ⇒ *'a ref* ⇒ *bool* ⇀ (*Haskell*) **infix** *4* ==
code-printing class-instance *ref* :: *HOL.equal* ⇀ (*Haskell*) –

27.4 A TSO litmus test

The classic TSO litmus test Owens et al. (2009, §1): write buffering allows both threads to read zero, which is impossible under sequential consistency.

definition *iwp2-3-a* :: (*nat* × *nat*) *tso* **where**
iwp2-3-a = *do* {
x ← *tso.Ref.ref 0*
; *y* ← *tso.Ref.ref 0*
; *xvr* ← *tso.Ref.ref 0*
; *yvr* ← *tso.Ref.ref 0*
; ((*do* { *x* := *1* ; *yv* ← !*y* ; *yvr* := *yv* })
|| (*do* { *y* := *1* ; *xv* ← !*x* ; *xvr* := *xv* }))
; *xv* <− !*xvr*
; *yv* <− !*yvr*
; *tso.return* (*xv*, *yv*)
}

code-thms *iwp2-3-a*
export-code *iwp2-3-a* **in** *Haskell*

schematic-goal *iwp2-3-a*: — "Can terminate with both threads reading 0"
shows ⦇*heap.empty*, *?xs*, *Some* (*0*, *0*)⦈ ≤ *prog.p2s* (*tso.t2p iwp2-3-a*)
supply *heap.simps*[*simp*]
apply (*rule inhabits.I*)
unfolding *iwp2-3-a-def*
apply (*simp add*: *prog.p2s.t2p*)
apply (*rule inhabits.spec.bind'*)
apply (*rule inhabits.tso.bind'*)
apply (*rule inhabits.tso.Ref.ref*[**where** *r*=*Ref 0*], *simp*; *fail*)
apply (*simp add*: *tso.bind.returnL*)
apply (*rule inhabits.tso.bind'*)
apply (*rule inhabits.tso.Ref.ref*[**where** *r*=*Ref 1*], *simp*)
apply (*simp add*: *tso.bind.returnL*)
apply (*rule inhabits.tso.bind'*)
apply (*rule inhabits.tso.Ref.ref*[**where** *r*=*Ref 2*], *simp*)
apply (*simp add*: *tso.bind.returnL*)
apply (*rule inhabits.tso.bind'*)
apply (*rule inhabits.tso.Ref.ref*[**where** *r*=*Ref 3*], *simp*)
apply (*simp add*: *tso.bind.returnL*)
apply (*rule inhabits.tso.commit'*)+
apply *simp*

apply (*rule inhabits.tso.bind'*)
apply (*simp add*: *tso.t2s.parallel prog.p2s.bind prog.p2s.parallel prog.p2s.t2p*)
apply (*rule inhabits.spec.bind'*)

apply (*rule inhabits.spec.parallelL'*)
apply (*rule inhabits.spec.bind'*)
apply (*rule inhabits.tso.bind'*)
apply (*rule inhabits.tso.Ref.update*)
apply (*simp add*: *tso.bind.returnL*)
apply (*rule inhabits.tso.bind'*)
apply (*rule inhabits.tso.Ref.lookup*)

apply (*simp add*: *tso.bind.returnL*)
apply (*rule inhabits.tso.Ref.update*)
apply (*rule inhabits.tau*)
apply (*simp add*: *spec.idle.bind-le-conv spec.idle.tso.t2s-le*; *fail*)
apply (*simp add*: *spec.bind.mono spec.interference.tso.t2s-le flip*: *spec.bind.bind*; *fail*)

apply (*rule inhabits.spec.parallelR'*)
apply (*rule inhabits.spec.bind'*)
apply (*rule inhabits.tso.bind'*)
apply (*rule inhabits.tso.Ref.update*)
apply (*simp add*: *tso.bind.returnL*)
apply (*rule inhabits.tso.bind'*)
apply (*rule inhabits.tso.Ref.lookup*)
apply (*simp add*: *tso.bind.returnL*)
apply (*rule inhabits.tso.Ref.update*)
apply (*rule inhabits.tau*)
apply (*simp add*: *spec.idle.bind-le-conv spec.idle.tso.t2s-le*; *fail*)
apply (*simp add*: *spec.bind.mono spec.interference.tso.t2s-le flip*: *spec.bind.bind*; *fail*)
apply *clarsimp*

apply (*rule inhabits.spec.parallelL'*)
apply (*rule inhabits.spec.bind'*[*OF inhabits.tso.commit*])+
apply (*simp add*: *tso.t2s.return split-def prog.bind.returnL raw.MFENCE.Nil flip*: *prog.p2s.bind*)
apply (*rule inhabits.tau*)
apply (*simp add*: *spec.idle.p2s-le*; *fail*)
apply (*simp add*: *spec.bind.mono spec.interference.tso.t2s-le flip*: *spec.bind.bind*; *fail*)

apply (*rule inhabits.spec.parallelR'*)
apply (*rule inhabits.spec.bind'*[*OF inhabits.tso.commit*])+
apply (*simp add*: *tso.t2s.return split-def prog.bind.returnL raw.MFENCE.Nil flip*: *prog.p2s.bind*)
apply (*rule inhabits.tau*)
apply (*simp add*: *spec.idle.p2s-le*; *fail*)
apply (*simp add*: *prog.p2s.interference-wind-bind*; *fail*)

apply (*simp add*: *prog.parallel.return flip*: *prog.p2s.parallel*)
apply (*rule inhabits.tau*)
apply (*simp add*: *spec.idle.p2s-le*; *fail*)
apply (*simp add*: *prog.bind.returnL flip*: *prog.p2s.bind*)
apply (*subst tso.t2s.return*[*symmetric*])
apply (*rule inhabits.tau*)
apply (*simp add*: *spec.idle.tso.t2s-le*; *fail*)

apply (*simp add*: *tso.bind.return*)
apply (*rule inhabits.tso.bind'*)
apply (*rule inhabits.tso.Ref.lookup*)
apply (*simp add*: *Ref.get-def apply-writes-def tso.bind.return*)
apply (*rule inhabits.tso.bind'*)
apply (*rule inhabits.tso.Ref.lookup*)
apply (*simp add*: *Ref.get-def apply-writes-def tso.bind.return tso.t2s.return*)
apply (*rule inhabits.prog.return*)
apply (*simp add*: *spec.bind.returnL spec.idle.p2s-le raw.MFENCE.Nil prog.bind.returnL*)
apply (*rule inhabits.prog.return*)
done

thm *iwp2-3-a*[*simplified apply-writes-def*, *simplified*]

28 Floyd-Warshall all-pairs shortest paths

The Floyd-Warshall algorithm computes the lengths of the shortest paths between all pairs of nodes by updating an adjacency (square) matrix that represents the edge weights. Our goal here is to present it at a very abstract level to exhibit the data dependencies.

Source materials:

- https://en.wikipedia.org/wiki/Floyd%E2%80%93Warshall_algorithm
- $AFP/Floyd_Warshall/Floyd_Warshall.thy
  - a proof by refinement yielding a thorough correctness result including negative weights but not the absence of edges
- Dingel (2002, §6.2)
  - Overly parallelised, which is not practically useful but does reveal the data dependencies
  - the refinement is pretty much the same as the direct partial correctness proof here
  - the equivalent to *fw-update* is a single expression

We are not very ambitious here. This theory:

- does not track the actual shortest paths here but it is easy to add another array to do so
- ignores numeric concerns
- assumes the graph is complete

A further step would be to refine the parallel program to the classic three-loop presentation.

definition *fw-update* :: (*'i*::*Ix* × *'i*, *nat*) *array* ⇒ *'i* × *'i* ⇒ *'i* ⇒ *unit imp* **where**
fw-update = (λ*a* (*i*, *j*) *k*. *do* {
ij ← *prog.Array.nth a* (*i*, *j*);
ik ← *prog.Array.nth a* (*i*, *k*);
kj ← *prog.Array.nth a* (*k*, *j*);
prog.whenM (*ik* + *kj* < *ij*) (*prog.Array.upd a* (*i*, *j*) (*ik* + *kj*))
})

— top-level specification: we can process the nodes in an arbitrary order
definition *fw-chaotic* :: (*'i*::*Ix* × *'i*, *nat*) *array* ⇒ *unit imp* **where**
fw-chaotic a =
(*let b* = *array.bounds a in*
prog.Array.fst-app-chaotic b (λ*k*. ||(*i*, *j*)∈*set* (*Ix.interval b*). *fw-update a* (*i*, *j*) *k*))

— executable version
definition *fw* :: (*'i*::*Ix* × *'i*, *nat*) *array* ⇒ *unit imp* **where**
fw a =
(*let b* = *array.bounds a in*
prog.Array.fst-app b (λ*k*. ||(*i*, *j*)∈*set* (*Ix.interval b*). *fw-update a* (*i*, *j*) *k*))

lemma *fw-fw-chaotic-le*: — the executable program refines the specification
shows *fw a* ≤ *fw-chaotic a*
unfolding *fw-chaotic-def fw-def*
by (*strengthen ord-to-strengthen*(*1*)[*OF prog.Array.fst-app-fst-app-chaotic-le*]) *simp*

Safety proof **type-synonym** *'i matrix = 'i × 'i ⇒ nat*

— The weight of the given path
fun *path-weight :: 'i matrix ⇒ 'i × 'i ⇒ 'i list ⇒ nat* **where**
path-weight m ij [] = m ij
| *path-weight m ij (k # xs) = m (fst ij, k) + path-weight m (k, snd ij) xs*

— The set of acyclic paths from *i* to *j* using the nodes *ks*
definition *paths :: 'i × 'i ⇒ 'i set ⇒ 'i list set* **where**
paths ij ks = {p. set p ⊆ ks ∧ fst ij ∉ set p ∧ snd ij ∉ set p ∧ distinct p}

— The minimum weight of a path from *i* to *j* using the nodes *ks*. See $AFP/Floyd_Warshall/Floyd_Warshall.thy for proof that these are minimal amongst all paths.
definition *min-path-weight :: 'i matrix ⇒ 'i × 'i ⇒ 'i set ⇒ nat* **where**
min-path-weight m ij ks = Min (path-weight m ij ‘ paths ij ks)

context
fixes *a :: ('i::Ix × 'i, nat) array*
fixes *m :: 'i matrix*
begin

definition *fw-p-inv :: 'i × 'i ⇒ 'i set ⇒ heap.t pred* **where** — process invariant
fw-p-inv ij ks = (heap.rep-inv a ∧ Array.get a ij = ⟨min-path-weight m ij ks⟩)

definition *fw-inv :: 'i set ⇒ heap.t pred* **where** — loop invariant
fw-inv ks = (∀ ij. ⟨ij∈set (Array.interval a)⟩ ⟶ fw-p-inv ij ks)

definition *fw-pre :: heap.t pred* **where** — overall precondition
fw-pre = (⟨Array.square a⟩ ∧ heap.rep-inv a
∧ (∀ ij. ⟨ij∈set (Array.interval a)⟩ ⟶ Array.get a ij = ⟨m ij⟩))

definition *fw-post :: unit ⇒ heap.t pred* **where** — overall postcondition
fw-post - = fw-inv (set (Ix.interval (fst-bounds (array.bounds a))))

end

setup ‹*Sign.mandatory-path paths*›

lemma *I*:
assumes *set p ⊆ ks*
assumes *i ∉ set p*
assumes *j ∉ set p*
assumes *distinct p*
shows *p ∈ paths (i, j) ks*
using *assms* **by** (*simp add*: *paths-def*)

lemma *Nil*:
shows *[] ∈ paths ij ks*
by (*simp add*: *paths-def*)

lemma *empty*:
shows *paths ij {} = {[]}*
by (*fastforce simp*: *paths-def*)

lemma *not-empty*:
shows *paths ij ks ≠ {}*
by (*metis empty-iff paths.Nil*)

lemma *monotone*:
  shows *mono* (*paths ij*)
by (*rule monoI*) (*auto simp add*: *paths-def*)

lemmas *mono* = *monoD*[*OF paths.monotone*]
lemmas *strengthen*[*strg*] = *st-monotone*[*OF paths.monotone*]

lemma *finite*:
  assumes *finite ks*
  shows *finite* (*paths ij ks*)
unfolding *paths-def* **by** (*rule finite-subset*[*OF - iffD1*[*OF finite-distinct-conv assms*]]) *auto*

lemma *unused*:
  assumes $p \in$ *paths ij* (*insert k ks*)
  assumes $k \notin$ *set p*
  shows $p \in$ *paths ij ks*
using *assms* **unfolding** *paths-def* **by** *blast*

lemma *decompE*:
  assumes $p \in$ *paths* (i, j) (*insert k ks*)
  assumes $k \in$ *set p*
  obtains *r s*
    where $p = r$ @ k # s
      and $r \in$ *paths* (i, k) *ks* **and** $s \in$ *paths* (k, j) *ks*
      and *distinct* (r @ s) **and** $i \notin$ *set* (r @ k # s) **and** $j \notin$ *set* (r @ k # s)
using *assms* **by** (*fastforce simp*: *paths-def dest*: *split-list*)

setup ‹*Sign.parent-path*›

setup ‹*Sign.mandatory-path path-weight*›

lemma *append*:
  shows *path-weight m ij* (*xs* @ *y* # *ys*) = *path-weight m* (*fst ij*, *y*) *xs* + *path-weight m* (*y*, *snd ij*) *ys*
by (*induct xs arbitrary*: *ij*) *simp-all*

setup ‹*Sign.parent-path*›

lemmas *min-path-weightI* = *trans*[*OF min-path-weight-def Min-eqI*]

setup ‹*Sign.mandatory-path min-path-weight*›

lemma *fw-update*:
  assumes *m*: *min-path-weight m* (i, k) *ks* + *min-path-weight m* (k, j) *ks* < *min-path-weight m* (i, j) *ks*
  assumes *finite ks*
  shows *min-path-weight m* (i, j) (*insert k ks*)
      = *min-path-weight m* (i, k) *ks* + *min-path-weight m* (k, j) *ks* (**is** *?lhs* = *?rhs*)
proof(*rule min-path-weightI*)
  from ‹*finite ks*› **show** *finite* (*path-weight m* (i, j) ‘ *paths* (i, j) (*insert k ks*))
    by (*simp add*: *paths.finite*)
next
  fix *w*
  assume *w*: *w* ∈ *path-weight m* (i, j) ‘ *paths* (i, j) (*insert k ks*)
  then obtain *p* **where** *p*: *w* = *path-weight m* (i, j) *p p* ∈ *paths* (i, j) (*insert k ks*) **..**
  show *?rhs* ≤ *w*
  proof(*cases k* ∈ *set p*)
    case *True* **with** *m* ‹*finite ks*› *w p* **show** *?thesis*
      by (*clarsimp simp*: *min-path-weight-def path-weight.append elim*!: *paths.decompE*)
        (*auto simp*: *Min-plus paths.finite paths.not-empty finite-image-set2 intro*!: *Min-le*)

```
  next
    case False with m ‹finite ks› w p show ?thesis
      unfolding min-path-weight-def
      by (fastforce simp: paths.finite paths.not-empty dest: paths.unused)
  qed
next
  from ‹finite ks› obtain pik
    where pik: pik ∈ paths (i, k) ks
      and mpik: Min (path-weight m (i, k) ‘ paths (i, k) ks) = path-weight m (i, k) pik
    by (meson finite-set Min-in finite-imageI paths.finite image-iff image-is-empty paths.not-empty)
  from ‹finite ks› obtain pkj
    where pkj: pkj ∈ paths (k, j) ks
      and mpkj: Min (path-weight m (k, j) ‘ paths (k, j) ks) = path-weight m (k, j) pkj
    by (meson finite-set Min-in finite-imageI paths.finite image-iff image-is-empty paths.not-empty)
  let ?p = pik @ k # pkj
  have ?p ∈ paths (i, j) (insert k ks)
  proof(rule paths.I)
    from pik pkj
    show set ?p ⊆ insert k ks by (auto simp: paths-def)
    show i ∉ set ?p
    proof(rule notI)
      assume i ∈ set ?p
      with m pik have i ∈ set pkj by (fastforce simp: paths-def)
      then obtain p' zs where *: pkj = zs @ i # p' by (meson split-list)
      moreover from pkj * have p' ∈ paths (i, j) ks by (simp add: paths-def)
      moreover note m ‹finite ks› mpkj
     ultimately show False by (simp add: paths.finite leD min-path-weight-def path-weight.append trans-le-add2)
    qed
    show j ∉ set ?p
    proof(rule notI)
      assume j ∈ set ?p
      with m pkj have j ∈ set pik by (fastforce simp: paths-def)
      then obtain p' zs where *: pik = p' @ j # zs by (meson split-list)
      moreover from pik * have p' ∈ paths (i, j) ks by (simp add: paths-def)
      moreover note m ‹finite ks› mpik
      ultimately show False
        by (fastforce simp: min-path-weight-def path-weight.append paths.finite paths.not-empty)
    qed
    show distinct ?p
    proof(rule ccontr)
      let ?p1 = takeWhile (λx. x ∉ set pkj) pik
      let ?l = hd (drop (length ?p1) pik)
      let ?p2 = tl (dropWhile (λx. x ≠ ?l) pkj)
      let ?p' = ?p1 @ ?l # ?p2
      assume ¬distinct (pik @ k # pkj)
      from pik pkj ‹¬distinct (pik @ k # pkj)› have strict-prefix ?p1 pik
        by (auto simp: paths-def strict-prefix-def takeWhile-is-prefix)
      from pik pkj ‹¬distinct (pik @ k # pkj)› ‹strict-prefix ?p1 pik› have strict-suffix ?p2 pkj
        by (fastforce simp: dropWhile-eq-drop tl-drop
                      intro: drop-strict-suffix[OF strict-suffix-tl]
                       dest: prefix-length-less nth-length-takeWhile)
      from ‹strict-prefix ?p1 pik› have ?l ∈ set pkj
        by (fastforce simp: hd-drop-conv-nth dest: prefix-length-less nth-length-takeWhile)
      have ?p' ∈ paths (i, j) ks
      proof(rule paths.I)
        from pik pkj ‹strict-prefix ?p1 pik› ‹strict-suffix ?p2 pkj› ‹?l ∈ set pkj› show set ?p' ⊆ ks
          by (force dest: set-takeWhileD strict-suffix-set-subset simp: paths-def)
        from ‹i ∉ set ?p› ‹strict-suffix ?p2 pkj› ‹?l ∈ set pkj› show i ∉ set ?p'
```

```
      by (auto dest: set-takeWhileD strict-suffix-set-subset)
    from ‹j ∉ set ?p› ‹strict-suffix ?p2 pkj› ‹?l ∈ set pkj› show j ∉ set ?p'
      by (auto dest: set-takeWhileD strict-suffix-set-subset)
    from pik pkj ‹strict-suffix ?p2 pkj› ‹?l ∈ set pkj› show distinct ?p'
      by (auto simp: paths-def distinct-tl dest!: set-takeWhileD strict-suffix-set-subset
          simp flip: arg-cong[where f=set, OF takeWhile-neq-rev, simplified])
  qed
  have path-weight m (i, j) ?p' ≤ path-weight m (i, k) pik + path-weight m (k, j) pkj
    unfolding path-weight.append
  proof(induct rule: add-le-mono[case-names l r])
    case l from ‹strict-prefix ?p1 pik› show ?case
      by (metis append.right-neutral append-take-drop-id fst-conv linorder-le-less-linear
            list.collapse not-add-less1 path-weight.append prefix-order.less-le takeWhile-eq-take)
  next
    case r from ‹?l ∈ set pkj› show ?case
      by (smt (verit) append.right-neutral hd-dropWhile le-add2 list.collapse path-weight.append
            set-takeWhileD snd-conv takeWhile-dropWhile-id)
  qed
  with m ‹finite ks› mpik mpkj ‹?p' ∈ paths (i, j) ks› show False
    by (fastforce simp: min-path-weight-def paths.finite paths.not-empty)
 qed
qed
with m mpik mpkj
show ?rhs ∈ path-weight m (i, j) ‘ paths (i, j) (insert k ks)
 by (force simp: min-path-weight-def path-weight.append)
qed

lemma return:
 assumes m: ¬(min-path-weight m (i, k) ks + min-path-weight m (k, j) ks < min-path-weight m (i, j) ks)
 assumes finite ks
 shows min-path-weight m (i, j) (insert k ks) = min-path-weight m (i, j) ks
unfolding min-path-weight-def
proof(rule Min-eqI)
 from ‹finite ks› show finite (path-weight m (i, j) ‘ paths (i, j) (insert k ks))
  by (simp add: paths.finite)
next
 fix w
 assume w: w ∈ path-weight m (i, j) ‘ paths (i, j) (insert k ks)
 then obtain p where p: w = path-weight m (i, j) p p ∈ paths (i, j) (insert k ks) ..
 with m ‹finite ks› show Min (path-weight m (i, j) ‘ paths (i, j) ks) ≤ w
 proof(cases k ∈ set p)
  case True with m ‹finite ks› w p show ?thesis
   by (auto simp: not-less min-path-weight-def path-weight.append paths.finite
        intro: order.trans[OF add-mono[OF Min-le Min-le]]
        elim!: order.trans paths.decompE)
 next
  case False with m ‹finite ks› w p show ?thesis
   by (meson Min-le finite-imageI paths.finite image-eqI paths.unused)
 qed
next
 from ‹finite ks›
 show Min (path-weight m (i, j) ‘ paths (i, j) ks) ∈ path-weight m (i, j) ‘ paths (i, j) (insert k ks)
  by (fastforce simp: paths.finite paths.not-empty intro: subsetD[OF - Min-in] subsetD[OF paths.mono])
qed

setup ‹Sign.parent-path›

setup ‹Sign.mandatory-path stable›
```

lemma *Id-on-fw-inv*:
shows *stable heap.Id*$_{\{a\}}$ (*fw-inv a m ys*)
by (*auto simp*: *fw-inv-def fw-p-inv-def intro*!: *stable.intro stable.impliesI*)

lemma *Id-on-fw-p-inv*:
shows *stable heap.Id*$_{\{a\}}$ (*fw-p-inv a m ij ks*)
by (*auto simp*: *fw-p-inv-def intro*: *stable.intro*)

lemma *modifies-fw-p-inv*:
assumes *ij* ∈ *set* (*Array.interval a*) − *is*
shows *stable Array.modifies*$_{a,\ is}$ (*fw-p-inv a m ij ks*)
using *assms* **by** (*auto simp*: *fw-p-inv-def intro*: *stable.intro*)

setup ‹*Sign.parent-path*›

lemma *fw-p-inv-cong*:
assumes *a* = *a*′
assumes *m* = *m*′
assumes *ij* = *ij*′
assumes *ks* = *ks*′
assumes *s* (*heap.addr-of a*) = *s*′ (*heap.addr-of a*′)
shows *fw-p-inv a m ij ks s* = *fw-p-inv a*′ *m*′ *ij*′ *ks*′ *s*′
using *assms* **by** (*simp add*: *fw-p-inv-def cong*: *heap.obj-at.cong Array.get.weak-cong*)

lemma *fw-p-invD*:
assumes *fw-p-inv a m ij ks s*
shows *heap.rep-inv a s*
and *Array.get a ij s* = *min-path-weight m ij ks*
using *assms* **unfolding** *fw-p-inv-def* **by** *blast*+

lemma *fw-p-inv-fw-update*:
assumes *finite ks*
assumes *ij* ∈ *set* (*Array.interval a*)
assumes *fw-p-inv a m ij ks s*
assumes *min-path-weight m* (*fst ij*, *k*) *ks* + *min-path-weight m* (*k*, *snd ij*) *ks* < *min-path-weight m ij ks*
shows *fw-p-inv a m ij* (*insert k ks*) (*Array.set a ij* (*min-path-weight m* (*fst ij*, *k*) *ks* + *min-path-weight m* (*k*, *snd ij*) *ks*) *s*)
using *assms* **by** (*cases ij*) (*simp add*: *fw-p-inv-def Array.simps*′ *min-path-weight.fw-update*)

lemma *fw-p-inv-return*:
assumes *finite ks*
assumes *fw-p-inv a m ij ks s*
assumes ¬(*min-path-weight m* (*fst ij*, *k*) *ks* + *min-path-weight m* (*k*, *snd ij*) *ks* < *min-path-weight m ij ks*)
shows *fw-p-inv a m ij* (*insert k ks*) *s*
using *assms* **by** (*cases ij*) (*simp add*: *fw-p-inv-def min-path-weight.return*)

setup ‹*Sign.mandatory-path ag*›

Dingel (2000, p109) key intuition: when processing index k, neither $a[i, k]$ and $a[k, j]$ change.

- his argument is bogus: it is enough to observe that shortest paths never get shorter by adding edges
- he unnecessarily assumes that $\delta(i, i) = 0$ for all i

lemma *fw-update*:
assumes *insert k ks* ⊆ *set* (*Ix.interval* (*fst-bounds* (*array.bounds a*)))
assumes *Array.square a*
assumes *ij*: *ij* ∈ *set* (*Array.interval a*)

defines $\bigwedge ij.\ G\ ij \equiv Array.modifies_{a,\ \{ij\ |\text{-::}unit.\ k \notin \{fst\ ij,\ snd\ ij\}\}}$
defines $A \equiv heap.Id_{\{a\}} \cup \bigcup\ (G\ `\ (set\ (Array.interval\ a) - \{ij\}))$
shows *prog.p2s* (*fw-update a ij k*)
$\leq \{\!|fw\text{-}p\text{-}inv\ a\ m\ ij\ ks \wedge fw\text{-}p\text{-}inv\ a\ m\ (fst\ ij,\ k)\ ks \wedge fw\text{-}p\text{-}inv\ a\ m\ (k,\ snd\ ij)\ ks|\!\}, A$
$\vdash G\ ij, \{|\lambda\text{-}.\ fw\text{-}p\text{-}inv\ a\ m\ ij\ (insert\ k\ ks)|\!\}$
proof −
from *assms*(*1*) **have** *finite ks*
using *finite-subset* **by** *auto*
from *assms*(*1−3*) **have** *ijk*: (*fst ij*, *k*) ∈ *set* (*Array.interval a*) (*k*, *snd ij*) ∈ *set* (*Array.interval a*)
by (*auto simp*: *Ix.square-def interval-prod-def*)
show *?thesis*
apply (*simp add*: *fw-update-def split-def*)
apply (*rule ag.pre-pre*)
apply (*rule ag.prog.bind*)+
apply (*rule ag.prog.if*)
apply (*rename-tac* v_{ij} v_{ik} v_{kj})
apply (*subst prog.Array.upd-def*)
apply (*rule-tac P*=λ*s. fw-p-inv a m ij ks s* ∧ *fw-p-inv a m* (*fst ij*, *k*) *ks s* ∧ *fw-p-inv a m* (*k*, *snd ij*) *ks s*
∧ v_{ij} = *Array.get a ij s* ∧ v_{ik} = *Array.get a* (*fst ij*, *k*) *s* ∧ v_{kj} = *Array.get a* (*k*, *snd ij*) *s*
in *ag.prog.action*)
apply (*clarsimp simp*: ‹*finite ks*› *fw-p-invD*(*2*) *fw-p-inv-fw-update ij*; *fail*)
using *ij* **apply** (*fastforce simp*: *G-def intro*: *Array.modifies.Array-set dest*: *fw-p-invD*(*1*))
using *ij ijk* **apply** (*fastforce simp*: *A-def G-def intro*: *stable.intro stable.Id-on-fw-p-inv stable.modifies-fw-p-inv*)
using *ij ijk* **apply** (*fastforce simp*: *A-def G-def intro*: *stable.intro stable.Id-on-fw-p-inv stable.modifies-fw-p-inv*)
apply (*rename-tac* v_{ij} v_{ik} v_{kj})
apply (*rule-tac Q*=λ*- s.* v_{ij} = *Array.get a ij s* ∧ v_{ik} = *Array.get a* (*fst ij*, *k*) *s* ∧ v_{kj} = *Array.get a* (*k*, *snd ij*) *s*
in *ag.augment-post*)
apply (*rule ag.prog.return*)
using *ij ijk* **apply** (*fastforce simp*: *A-def G-def intro*: *stable.intro stable.Id-on-fw-p-inv stable.modifies-fw-p-inv*)
apply (*rename-tac* v_{ij} v_{ik})
apply (*rule-tac Q*=λ*v s. fw-p-inv a m ij ks s* ∧ *fw-p-inv a m* (*fst ij*, *k*) *ks s* ∧ *fw-p-inv a m* (*k*, *snd ij*) *ks s*
∧ v_{ij} = *Array.get a ij s* ∧ v_{ik} = *Array.get a* (*fst ij*, *k*) *s* ∧ *v* = *Array.get a* (*k*, *snd ij*) *s*
in *ag.post-imp*)
apply (*force simp*: ‹*finite ks*› *fw-p-invD*(*2*) *fw-p-inv-return*)
apply (*subst prog.Array.nth-def*)
apply (*rule ag.prog.action*)
apply (*clarsimp split del*: *if-split*; *assumption*)
apply *fast*
using *ij ijk* **apply** (*fastforce simp*: *A-def G-def intro*: *stable.intro stable.Id-on-fw-p-inv stable.modifies-fw-p-inv*)
using *ij ijk* **apply** (*fastforce simp*: *A-def G-def intro*: *stable.intro stable.Id-on-fw-p-inv stable.modifies-fw-p-inv*)
apply (*subst prog.Array.nth-def*)
apply (*rule ag.prog.action*)
apply (*clarsimp*; *assumption*)
apply *fast*
using *ij ijk* **apply** (*fastforce simp*: *A-def G-def intro*: *stable.intro stable.Id-on-fw-p-inv stable.modifies-fw-p-inv*)
using *ij ijk* **apply** (*fastforce simp*: *A-def G-def intro*: *stable.intro stable.Id-on-fw-p-inv stable.modifies-fw-p-inv*)
apply (*subst prog.Array.nth-def*)
apply (*rule ag.prog.action*)
apply (*clarsimp*; *assumption*)
apply *fast*
using *ij ijk* **apply** (*fastforce simp*: *A-def G-def intro*: *stable.intro stable.Id-on-fw-p-inv stable.modifies-fw-p-inv*)
using *ij ijk* **apply** (*fastforce simp*: *A-def G-def intro*: *stable.intro stable.Id-on-fw-p-inv stable.modifies-fw-p-inv*)
apply *blast*
done
qed

lemma *fw-chaotic*:

fixes $a :: ('i::Ix \times 'i, nat)$ *array*
fixes $m :: 'i$ *matrix*
shows *prog.p2s* (*fw-chaotic a*) $\leq$ ⦃*fw-pre a m*⦄, $heap.Id_{\{a\}} \vdash heap.modifies_{\{a\}}$, ⦃*fw-post a m*⦄
unfolding *fw-chaotic-def fw-pre-def*
apply (*simp add*: *prog.p2s.simps Let-def split-def*)
apply (*rule ag.gen-asm*)
apply (*rule ag.pre-pre-post*)
apply (*rule ag.prog.fst-app-chaotic*[**where** *P*=*fw-inv a m*])
apply (*rule ag.pre*)
apply (*rule ag.prog.Parallel*)
apply (*rule ag.fw-update*[**where** *m*=*m*])
apply (*simp*; *fail*)
apply (*simp*; *fail*)
apply (*simp*; *fail*)
apply (*fastforce simp*: *fw-inv-def split-def Ix.prod.interval-conv Ix.square.conv*)
apply *blast*
using *Array.modifies.heap-modifies-le* **apply** *blast*
apply (*simp add*: *fw-inv-def*; *fail*)
apply (*simp add*: *stable.Id-on-fw-inv*; *fail*)
apply (*fastforce simp*: *fw-pre-def fw-inv-def fw-p-inv-def min-path-weight-def paths.empty*)
apply (*fastforce simp*: *fw-post-def split-def stable.Id-on-fw-inv*)
done

setup ‹*Sign.parent-path*›

References

M. Abadi. An axiomatization of lamport's temporal logic of actions. In *CONCUR '90*, volume 458 of *LNCS*, pages 57–69. Springer, 1990. doi: 10.1007/BFB0039051.

M. Abadi and L. Lamport. The existence of refinement mappings. *Theoretical Computer Science*, 82(2):253–284, 1991. doi: 10.1016/0304-3975(91)90224-P.

M. Abadi and L. Lamport. Conjoining specifications. *ACM Transactions on Programming Languages and Systems*, 17(3):507–534, 1995. doi: 10.1145/203095.201069.

M. Abadi and S. Merz. An abstract account of composition. In *MFCS'95*, volume 969 of *LNCS*, pages 499–508. Springer, 1995. doi: 10.1007/3-540-60246-1_155.

M. Abadi and S. Merz. On TLA as a logic. In Manfred Broy, editor, *Proceedings of the NATO Advanced Study Institute on Deductive Program Design, Marktoberdorf, Germany*, pages 235–271. IOS Press, 1996. ISBN 3-540-60947-4.

M. Abadi and G. D. Plotkin. A logical view of composition and refinement. In *POPL'1991*, pages 323–332. ACM Press, 1991. doi: 10.1145/99583.99626.

M. Abadi and G. D. Plotkin. A logical view of composition. *Theoretical Computer Science*, 114(1):3–30, 1993. doi: 10.1016/0304-3975(93)90151-I.

B. Alpern and F. B. Schneider. Defining liveness. *Information Processing Letters*, 21(4):181–185, 1985. doi: 10.1016/0020-0190(85)90056-0.

B. Alpern, A. J. Demers, and F. B. Schneider. Safety without stuttering. *Information Processing Letters*, 23(4): 177–180, 1986. doi: 10.1016/0020-0190(86)90132-8.

K. R. Apt, F. S. de Boer, and E.-R. Olderog. *Verification of Sequential and Concurrent Programs*. Texts in Computer Science. Springer, 2009. ISBN 978-1-84882-744-8. doi: 10.1007/978-1-84882-745-5.

A. Armstrong, V. B. F. Gomes, and G. Struth. Algebraic principles for rely-guarantee style concurrency verification tools. In *FM 2014*, volume 8442 of *LNCS*, pages 78–93. Springer, 2014. doi: 10.1007/978-3-319-06410-9_6.

R. C. Backhouse. Galois connections and fixed point calculus. In R.C. Backhouse, R. L. Crole, and J. Gibbons, editors, *Algebraic and Coalgebraic Methods in the Mathematics of Program Construction*, volume 2297 of *LNCS*, pages 89–148. Springer, 2000. doi: 10.1007/3-540-47797-7_4.

S. D. Brookes. Full abstraction for a shared-variable parallel language. *Information and Computation*, 127(2): 145–163, 1996. doi: 10.1006/inco.1996.0056.

A. Cau and P. Collette. Parallel composition of assumption-commitment specifications: A unifying approach for shared variable and distributed message passing concurrency. *Acta Informatica*, 33(2):153–176, 1996. doi: 10.1007/s002360050039.

K. M. Chandy and J. Misra. *Parallel program design - a foundation*. Addison-Wesley, 1989. ISBN 978-0-201-05866-6.

B. A. Davey and H. A. Priestley. *Introduction to Lattices and Order, Second Edition*. Cambridge University Press, 2002. ISBN 978-0-521-78451-1. doi: 10.1017/CBO9780511809088.

G. De Giacomo and M. Y. Vardi. Linear temporal logic and linear dynamic logic on finite traces. In *IJCAI'13*, pages 854–860. IJCAI/AAAI, 2013.

W.-P. de Roever and K. Engelhardt. *Data Refinement: Model-Oriented Proof Methods and their Comparison*. Cambridge University Press, 1998.

W.-P. de Roever, F. S. de Boer, U. Hannemann, J. Hooman, Y. Lakhnech, M. Poel, and J. Zwiers. *Concurrency Verification: Introduction to Compositional and Noncompositional Methods*, volume 54 of *Cambridge Tracts in Theoretical Computer Science*. Cambridge University Press, 2001. ISBN 0-521-80608-9.

J. Dingel. Modular verification for shared-variable concurrent programs. In *CONCUR '96*, volume 1119 of *LNCS*, pages 703–718. Springer, 1996. doi: 10.1007/3-540-61604-7_85.

J. Dingel. *Systematic parallel programming*. PhD thesis, Carnegie Mellon University, May 2000. CMU Tech Report CS-99-172.

J. Dingel. A refinement calculus for shared-variable parallel and distributed programming. *Formal Aspects of Computing*, 14(2):123–197, 2002. doi: 10.1007/s001650200032.

E. A. Emerson. Alternative semantics for temporal logics. *Theoretical Computer Science*, 26:121–130, 1983. doi: 10.1016/0304-3975(83)90082-8.

L. Esakia, G. Bezhanishvili, W. H. Holliday, and A. Evseev. *Heyting Algebras: Duality Theory*. Springer, 1st edition, 2019. ISBN 3030120953.

S. Foster, J. Baxter, A. Cavalcanti, J. Woodcock, and F. Zeyda. Unifying semantic foundations for automated verification tools in isabelle/utp. *Science of Computer Programming*, 197:102510, 2020. ISSN 0167-6423. doi: 10.1016/j.scico.2020.102510.

G. Gierz, K. H. Hofmann, K. Keimel, J. D. Lawson, M. Mislove, and D. S. Scott. *Continuous Lattices and Domains*. Encyclopedia of Mathematics and its Applications. Cambridge University Press, 2003. doi: 10.1017/CBO9780511542725.

R. Goldblatt. *Logics of Time and Computation*. Number 7 in CSLI Lecture Notes. Center for the Study of Language and Information, Stanford, 2 edition, 1992.

I. J. Hayes. Generalised rely-guarantee concurrency: an algebraic foundation. *Formal Aspects of Computing*, 28 (6):1057–1078, 2016. doi: 10.1007/s00165-016-0384-0.

I. J. Hayes and C. B. Jones. A guide to rely/guarantee thinking. In *SETSS 2017*, volume 11174 of *LNCS*, pages 1–38. Springer, 2017. doi: 10.1007/978-3-030-02928-9_1.

C. A. R. Hoare and J. He. The weakest prespecification. *Information Processing Letters*, 24(2):127–132, 1987. doi: 10.1016/0020-0190(87)90106-2. Oxford Technical Monograph PRG-44.

C. A. R. Hoare, I. J. Hayes, J. He, C. Morgan, A. W. Roscoe, J. W. Sanders, I. H. Sørensen, J. M. Spivey, and B. Sufrin. Laws of programming. *Communications of the ACM*, 30(8):672–686, 1987a. doi: 10.1145/27651.27653.

C. A. R. Hoare, J. He, and J. W. Sanders. Prespecification in data refinement. *Information Processing Letters*, 25(2):71–76, 1987b. doi: 10.1016/0020-0190(87)90224-9.

C. A. R. Hoare, J. He, and A. Sampaio. Algebraic derivation of an operational semantics. In G. D. Plotkin, C. Stirling, and M. Tofte, editors, *Proof, Language, and Interaction, Essays in Honour of Robin Milner*, pages 77–98. The MIT Press, 2000.

T. Hoare, B. Möller, G. Struth, and I. Wehrman. Concurrent kleene algebra and its foundations. *Journal of Logic and Algebraic Programming*, 80(6):266–296, 2011. doi: 10.1016/j.jlap.2011.04.005.

P. B. Jackson. Verifying a garbage collection algorithm. In *TPHOLs*, volume 1479 of *LNCS*, pages 225–244. Springer, 1998. doi: 10.1007/BFb0055139.

R. Jagadeesan, G. Petri, and J. Riely. Brookes is relaxed, almost! In Lars Birkedal, editor, *FOSSACS 2012*, volume 7213 of *LNCS*, pages 180–194. Springer, 2012. doi: 10.1007/978-3-642-28729-9_12.

C. B. Jones. Tentative steps toward a development method for interfering programs. *ACM Transactions on Programming Languages and Systems*, 5(4):596–619, 1983. doi: 10.1145/69575.69577.

B. Jonsson and Y.-K. Tsay. Assumption/guarantee specifications in linear-time temporal logic. *Theoretical Computer Science*, 167(1&2):47–72, 1996. doi: 10.1016/0304-3975(96)00069-2.

R. M. Karp and R. E. Miller. Parallel program schemata. *Journal of Computer and System Sciences*, 3(2):147–195, 1969. doi: 10.1016/S0022-0000(69)80011-5.

E. Kindler. Safety and liveness properties: A survey. *Bulletin of the European Association for Theoretical Computer Science*, 53(30):268–272, 6 1994.

D. Kozen. A completeness theorem for kleene algebras and the algebra of regular events. *Information and Computation*, 110(2):366–390, 1994. doi: 10.1006/inco.1994.1037.

F. Kröger and S. Merz. *Temporal Logic and State Systems*. Texts in Theoretical Computer Science. An EATCS Series. Springer, 2008. ISBN 978-3-540-67401-6.

L. Lamport. The temporal logic of actions. *ACM Transactions on Programming Languages and Systems*, 16(3): 872–923, 1994. doi: 10.1145/177492.177726.

L. Lamport. Specifying concurrent systems in TLA$^+$. In M. Broy and R. Steinbrüggen, editors, *Calculational System Design*, volume 173 of *NATO Science Series, III: Computer and Systems Sciences*, pages 183–247. IOS Press, January 2000. ISBN 9789051994599. Proceedings of Marktoberdorf 1998.

H. Liang, X. Feng, and M. Fu. Rely-guarantee-based simulation for compositional verification of concurrent program transformations. *ACM Transactions on Programming Languages and Systems*, 36(1):3:1–3:55, 2014. doi: 10.1145/2576235.

N. A. Lynch. *Distributed Algorithms*. Morgan Kaufmann, 1996. ISBN 1-55860-348-4.

P. Maier. A set-theoretic framework for assume-guarantee reasoning. In *ICALP'2001*, volume 2076 of *LNCS*, pages 821–834. Springer, 2001. doi: 10.1007/3-540-48224-5_67.

P. Maier. Intuitionistic LTL and a new characterization of safety and liveness. In *CSL 2004*, volume 3210 of *LNCS*, pages 295–309. Springer, 2004. doi: 10.1007/978-3-540-30124-0_24.

Z. Manna and A. Pnueli. The anchored version of the temporal framework. In J. W. de Bakker, W. P. de Roever, and G. Rozenberg, editors, *Linear Time, Branching Time and Partial Order in Logics and Models for Concurrency, School/Workshop, Noordwijkerhout, The Netherlands, May 30 - June 3, 1988, Proceedings*, volume 354 of *LNCS*, pages 201–284. Springer, 1988. doi: 10.1007/BFb0013024.

Z. Manna and A. Pnueli. Tools and rules for the practicing verifier. In R. F. Rashid, editor, *CMU Computer Science: A 25th Anniversary Commemorative*, pages 121–156. ACM Press and Addison-Wesley, 1991. Also Technical Report STAN-CS-90-1321.

P. Manolios and R. J. Trefler. A lattice-theoretic characterization of safety and liveness. In E. Borowsky and S. Rajsbaum, editors, *PODC'2003*, pages 325–333. ACM, 2003. doi: 10.1145/872035.872083.

A. Melton, D. A. Schmidt, and G. E. Strecker. Calois connections and computer science applications. In *Category Theory and Computer Programming*, volume 240 of *LNCS*, pages 299–312. Springer, 1985. doi: 10.1007/3-540-17162-2_130.

M. Müller-Olm. *Modular Compiler Verification - A Refinement-Algebraic Approach Advocating Stepwise Abstraction*, volume 1283 of *LNCS*. Springer, 1997. doi: 10.1007/BFb0027453.

H. Ono. *Proof Theory and Algebra in Logic*. Short Textbooks in Logic. Springer, 2019. doi: 10.1007/978-981-13-7997-0.

S. Owens, S. Sarkar, and P. Sewell. A better x86 memory model: x86-TSO. In *TPHOLs'2009*, volume 5674 of *LNCS*, pages 391–407. Springer, 2009. doi: 10.1007/978-3-642-03359-9_27. URL https://www.cl.cam.ac.uk/~pes20/weakmemory/x86tso-paper.pdf.

S. S. Owicki and D. Gries. An axiomatic proof technique for parallel programs I. *Acta Informatica*, 6:319–340, 1976. doi: 10.1007/BF00268134.

S. S. Owicki and L. Lamport. Proving liveness properties of concurrent programs. *ACM Transactions on Programming Languages and Systems*, 4(3):455–495, 1982. doi: 10.1145/357172.357178.

J. L. Pfaltz and J. Šlapal. Transformations of discrete closure systems. *Acta Mathematica Hungarica*, 138(4):386–405, 2013. doi: 10.1007/s10474-012-0262-z.

V. R. Pratt. Action logic and pure induction. In J. van Eijck, editor, *Logics in AI, European Workshop, JELIA '90, Amsterdam, The Netherlands, September 10-14, 1990, Proceedings*, volume 478 of *LNCS*, pages 97–120. Springer, 1990. doi: 10.1007/BFb0018436.

L. Prensa Nieto. The rely-guarantee method in isabelle/hol. In *ESOP 2003*, volume 2618 of *LNCS*, pages 348–362. Springer, 2003. doi: 10.1007/3-540-36575-3_24.

B. K. Rosen. Correctness of parallel programs: The church-rosser approach. *Theoretical Computer Science*, 2(2):183–207, 1976. doi: 10.1016/0304-3975(76)90032-3.

F. B. Schneider. Decomposing properties into safety and liveness using predicate logic. Technical Report 87-874, Department of Computer Science, Cornell University, October 1987.

D. S. Scott. *The Kleene Symposium*, volume 101 of *Studies in Logic and the Foundations of Mathematics*, chapter Lambda Calculus: Some Models, Some Philosophy, pages 223–265. Elsevier, 1980. doi: 10.1016/S0049-237X(08)71262-X.

A. P. Sistla. Safety, liveness and fairness in temporal logic. *Formal Aspects of Computing*, 6(5):495–512, 1994. doi: 10.1007/BF01211865.

M. H. Stone. Topological representations of distributive lattices and Brouwerian logics. *Časopis pro pěstování matematiky a fysiky*, 67(1):1–25, 1938. doi: 10.21136/CPMF.1938.124080.

V. Vafeiadis. *Modular fine-grained concurrency verification*. PhD thesis, University of Cambridge, UK, 2008.

D. van Dalen. *Logic and structure (4. ed.)*. Universitext. Springer, 2004. ISBN 978-3-540-57839-0.

R. van Glabbeek and P. Höfner. Progress, justness, and fairness. *ACM Computing Surveys*, 52(4):69:1–69:38, 2019. doi: 10.1145/3329125.

S. van Staden. Constructing the views framework. In *UTP 2014*, volume 8963 of *LNCS*, pages 62–83. Springer, 2014. doi: 10.1007/978-3-319-14806-9_4.

S. van Staden. On rely-guarantee reasoning. In *MPC 2015*, volume 9129 of *LNCS*, pages 30–49. Springer, 2015. doi: 10.1007/978-3-319-19797-5_2.

S. Vickers. *Topology via Logic*. Cambridge University Press, 1989. ISBN 0521360625.

J. S. Warford, D. Vega, and S. M. Staley. A calculational deductive system for linear temporal logic. *ACM Computing Surveys*, 53(3):53:1–53:38, 2020. doi: 10.1145/3387109.

J. Wickerson, M. Dodds, and M. J. Parkinson. Explicit stabilisation for modular rely-guarantee reasoning. In *ESOP 2010*, volume 6012 of *LNCS*, pages 610–629. Springer, 2010. doi: 10.1007/978-3-642-11957-6_32. URL https://johnwickerson.github.io/expstab.thy.html. Extended version in UCAM-CL-TR-774.

Q. Xu and J. He. A theory of state-based parallel programming: Part 1. In Joseph M. Morris and Roger C. Shaw, editors, *4th Refinement Workshop*, pages 326–359. Springer, 1991.

Q. Xu and J. He. Laws of parallel programming with shared variables. In David Till, editor, *6th Refinement Workshop, Proceedings of the 6th Refinement Workshop, organised by BCS-FACS, London, UK, 5-7 January 1994*, Workshops in Computing, pages 205–216. Springer, 1994. doi: 10.1007/978-1-4471-3240-0_11.

Q. Xu, A. Cau, and P. Collette. On unifying assumption-commitment style proof rules for concurrency. In B. Jonsson and J. Parrow, editors, *CONCUR '94*, volume 836 of *LNCS*, pages 267–282. Springer, 1994. doi: 10.1007/978-3-540-48654-1_22.

Q. Xu, W.-P. de Roever, and J. He. The rely-guarantee method for verifying shared variable concurrent programs. *Formal Aspects of Computing*, 9(2):149–174, 1997. doi: 10.1007/BF01211617.

Y. Zakowski, D. Cachera, D. Demange, G. Petri, D. Pichardie, S. Jagannathan, and J. Vitek. Verifying a concurrent garbage collector with a rely-guarantee methodology. *Journal of Automated Reasoning*, 63(2):489–515, 2019. doi: 10.1007/s10817-018-9489-x.

J. Zwiers. *Compositionality, Concurrency and Partial Correctness - Proof Theories for Networks of Processes, and Their Relationship*, volume 321 of *LNCS*. Springer, 1989. ISBN 3-540-50845-7. doi: 10.1007/BFb0020836.