NREST

Maximilian P.L. Haslbeck

November 9, 2026

Abstract

This entry introduces NREST, a nondeterministc result monad with time, which was spun off a development of a framework for verifying functional correctness and worst-case complexity of algorithms refined down to LLVM, due to Haslbeck and Lammich.

Contents

1 Auxiliaries **2**
- 1.1 Auxiliaries for option . . . 2
- 1.2 Auxiliaries for enat . . . 3
- 1.3 Auxiliary (for Sup and Inf) . . . 3

2 NREST **4**

3 pointwise reasoning **7**

4 Monad Operators **9**

5 Monad Rules **10**

6 Monotonicity lemmas **10**
- 6.1 Derived Program Constructs . . . 10
  - 6.1.1 Assertion . . . 10
- 6.2 SELECT . . . 11

7 RECT **12**

8 Generalized Weakest Precondition **14**
- 8.1 mm . . . 14
- 8.2 mii . . . 15
- 8.3 lst - latest starting time . . . 16
- 8.4 pointwise reasoning about lst via nres3 . . . 17
- 8.5 rules for lst . . . 17

9 consequence rules **18**

10 Experimental Hoare reasoning **19**

11 VCG **19**
11.1 Progress rules 20

12 rules for whileT **21**

13 some Monadic Refinement Automation **22**
13.1 Data Refinement 22
13.1.1 Examples 22
13.2 WHILET refine 26
13.3 ASSERT 27
13.4 VCG splitter 28
13.5 mm3 and emb 28
13.6 Setup Labeled VCG 29
13.7 Examples, labeled vcg 35
13.8 progress solver 36
13.9 more stuff involving mm3 and emb 36
13.10VCG for monadic programs 37
13.10.1 old 37
13.10.2 new setup 37
13.11setup for *refine-vcg* 38
13.12Relators 38
13.13Auxilliary Lemmas 39
13.14Definition 39
13.15Proof Rules 40
13.16Nres-Fold with Interruption (nfoldli) 41

theory
NREST-Auxiliaries
imports
HOL−Library.Extended-Nat Automatic-Refinement.Automatic-Refinement
begin

unbundle *lattice-syntax*

1 Auxiliaries

1.1 Auxiliaries for option

lemma *less-eq-option-None-is-None'*: $x \leq None \longleftrightarrow x = None$
⟨*proof*⟩

lemma *everywhereNone*: $(\forall x{\in}X.\ x = None) \longleftrightarrow X = \{\} \vee X = \{None\}$
⟨*proof*⟩

1.2 Auxiliaries for enat

lemma *enat-minus-mono*: $a' \geq b \Longrightarrow a' \geq a \Longrightarrow a' - b \geq (a::enat) - b$
⟨*proof*⟩

lemma *enat-plus-minus-aux1*: $a + b \leq a' \Longrightarrow \neg\ a' < a \Longrightarrow b \leq a' - (a::enat)$
⟨*proof*⟩

lemma *enat-plus-minus-aux2*: $\neg\ a < a' \Longrightarrow b \leq a - a' \Longrightarrow a' + b \leq (a::enat)$
⟨*proof*⟩

lemma *enat-minus-inf-conv*[*simp*]: $a - enat\ n = \infty \longleftrightarrow a{=}\infty$ ⟨*proof*⟩
lemma *enat-minus-fin-conv*[*simp*]: $a - enat\ n = enat\ m \longleftrightarrow (\exists\, k.\ a{=}enat\ k \wedge m = k - n)$
⟨*proof*⟩

lemma *helper*: $x2 \leq x2a \Longrightarrow \neg\ x2 < a \Longrightarrow \neg\ x2a < a \Longrightarrow\ x2 - (a::enat) \leq x2a - a$
⟨*proof*⟩

lemma *helper2*: $x2b \leq x2 \Longrightarrow \neg\ x2a < x2\ \Longrightarrow \neg\ x2a < x2b \Longrightarrow x2a - (x2::enat) \leq x2a - x2b$
⟨*proof*⟩

lemma *Sup-finite-enat*: $Sup\ X = Some\ (enat\ a) \Longrightarrow Some\ (enat\ a) \in X$
⟨*proof*⟩

lemma *Sup-enat-less2*: $Sup\ X = \infty \Longrightarrow \exists\, x{\in}X.\ enat\ t < x$
⟨*proof*⟩

lemma *enat-upper*[*simp*]: $t \leq Some\ (\infty::enat)$
⟨*proof*⟩

1.3 Auxiliary (for Sup and Inf)

lemma *aux11*: $f`X{=}\{y\} \longleftrightarrow (X{\neq}\{\} \wedge (\forall\, x{\in}X.\ f\ x = y))$ ⟨*proof*⟩

lemma *aux2*: $(\lambda f.\ f\ x)\ `\ \{[x \mapsto t1]\ |x\ t1.\ M\ x = Some\ t1\} = \{None\} \longleftrightarrow (M\ x = None \wedge M{\neq}Map.empty)$
⟨*proof*⟩

lemma *aux3*: $(\lambda f.\ f\ x)\ `\ \{[x \mapsto t1]\ |x\ t1.\ M\ x = Some\ t1\}$
$= \{Some\ t1 \mid t1.\ M\ x = Some\ t1\} \cup (\{None \mid y.\ y{\neq}x \wedge M\ y \neq None\ \})$
⟨*proof*⟩

lemma *Sup-pointwise-eq-fun*: $(\bigsqcup f \in \{[x \mapsto t1]\ |x\ t1.\ M\ x = Some\ t1\}.\ f\ x) = M\ x$
⟨*proof*⟩

lemma *SUP-eq-None-iff*: $(\bigsqcup f\ \in\ X.\ f\ x)\ =\ None \longleftrightarrow X{=}\{\} \vee (\forall f{\in}X.\ f\ x =$

None)
⟨*proof*⟩

lemma *SUP-eq-Some-iff*:
(⨆ $f \in X.\ f\ x$) = *Some* $t \longleftrightarrow (\exists f{\in}X.\ f\ x \neq None) \wedge (t{=}Sup\ \{t' \mid f\ t'.\ f{\in}X \wedge f\ x = Some\ t'\})$
⟨*proof*⟩

lemma *Sup-enat-less*: $X \neq \{\} \Longrightarrow enat\ t \leq Sup\ X \longleftrightarrow (\exists x{\in}X.\ enat\ t \leq x)$
⟨*proof*⟩

lemma fixes $Q\ P$ **shows**
$Inf\ \{\ P\ x \leq Q\ x\ |x.\ True\} \longleftrightarrow P \leq Q$
⟨*proof*⟩

end
theory *NREST*
imports *HOL−Library.Extended-Nat Refine-Monadic.RefineG-Domain Refine-Monadic.Refine-Misc*

HOL−Library.Monad-Syntax NREST-Auxiliaries
begin

2 NREST

datatype *'a nrest = FAILi | REST 'a ⇒ enat option*

instantiation *nrest* :: (*type*) *complete-lattice*
begin

fun *less-eq-nrest* **where**
$\text{-} \leq FAILi \longleftrightarrow True\ |$
$(REST\ a) \leq (REST\ b) \longleftrightarrow a \leq b\ |$
$FAILi \leq (REST\ \text{-}) \longleftrightarrow False$

fun *less-nrest* **where**
$FAILi < \text{-} \longleftrightarrow False\ |$
$(REST\ \text{-}) < FAILi \longleftrightarrow True\ |$
$(REST\ a) < (REST\ b) \longleftrightarrow a < b$

fun *sup-nrest* **where**
$sup\ \text{-}\ FAILi = FAILi\ |$
$sup\ FAILi\ \text{-} = FAILi\ |$
$sup\ (REST\ a)\ (REST\ b) = REST\ (\lambda x.\ max\ (a\ x)\ (b\ x))$

fun *inf-nrest* **where**

inf x FAILi = x |
inf FAILi x = x |
inf (REST a) (REST b) = REST (λx. min (a x) (b x))

lemma *min (None) (Some (1::enat)) = None ⟨proof⟩*
lemma *max (None) (Some (1::enat)) = Some 1 ⟨proof⟩*

definition *Sup X ≡ if FAILi∈X then FAILi else REST (Sup {f . REST f ∈ X})*
definition *Inf X ≡ if ∃f. REST f∈X then REST (Inf {f . REST f ∈ X}) else FAILi*

definition *bot ≡ REST (Map.empty)*
definition *top ≡ FAILi*

instance
⟨*proof*⟩
end

definition *RETURNT x ≡ REST (λe. if e=x then Some 0 else None)*
abbreviation *FAILT ≡ top::'a nrest*
abbreviation *SUCCEEDT ≡ bot::'a nrest*
abbreviation *SPECT* **where** *SPECT ≡ REST*

lemma *RETURNT-alt*: *RETURNT x = REST [x↦0]*
⟨*proof*⟩

lemma *nrest-inequalities*[*simp*]:
FAILT ≠ REST X
FAILT ≠ SUCCEEDT
FAILT ≠ RETURNT x
SUCCEEDT ≠ FAILT
SUCCEEDT ≠ RETURNT x
REST X ≠ FAILT
RETURNT x ≠ FAILT
RETURNT x ≠ SUCCEEDT
⟨*proof*⟩

lemma *nrest-more-simps*[*simp*]:
SUCCEEDT = REST X ⟷ X=Map.empty
REST X = SUCCEEDT ⟷ X=Map.empty
REST X = RETURNT x ⟷ X=[x↦0]
REST X = REST Y ⟷ X=Y
RETURNT x = REST X ⟷ X=[x↦0]
RETURNT x = RETURNT y ⟷ x=y
⟨*proof*⟩

lemma *nres-simp-internals*:
REST Map.empty = SUCCEEDT

FAILi = FAILT
⟨*proof*⟩

lemma *nres-order-simps*[*simp*]:
¬ *FAILT* ≤ *REST M*
REST M ≤ *REST M′* ⟷ (*M*≤*M′*)
⟨*proof*⟩

lemma *nres-top-unique*[*simp*]: *FAILT* ≤ *S′* ⟷ *S′* = *FAILT*
⟨*proof*⟩

lemma *FAILT-cases*[*simp*]: (*case FAILT of FAILi* ⇒ *P* | *REST x* ⇒ *Q x*) = *P*
⟨*proof*⟩

lemma *nrest-Sup-FAILT*:
Sup X = *FAILT* ⟷ *FAILT* ∈ *X*
FAILT = *Sup X* ⟷ *FAILT* ∈ *X*
⟨*proof*⟩

lemma *nrest-Sup-SPECT-D*: *Sup X* = *SPECT m* ⟹ *m x* = *Sup* {*f x* | *f*. *REST f* ∈ *X*}
⟨*proof*⟩

declare *nres-simp-internals*(*2*)[*simp*]

lemma *nrest-noREST-FAILT*[*simp*]: (∀ *x2*. *m* ≠ *REST x2*) ⟷ *m*=*FAILT*
⟨*proof*⟩

lemma *no-FAILTE*:
assumes *g xa* ≠ *FAILT*
obtains *X* **where** *g xa* = *REST X*
⟨*proof*⟩

lemma *case-prod-refine*:
fixes *P Q* :: *′a* ⇒ *′b* ⇒ *′c nrest*
assumes ⋀*a b*. *P a b* ≤ *Q a b*
shows (*case x of* (*a*,*b*) ⇒ *P a b*) ≤ (*case x of* (*a*,*b*) ⇒ *Q a b*)
⟨*proof*⟩

lemma *case-option-refine*:
fixes *P Q* :: *′a* ⇒ *′b* ⇒ *′c nrest*
assumes
PN ≤ *QN*
⋀*a*. *PS a* ≤ *QS a*
shows (*case x of None* ⇒ *PN* | *Some a* ⇒ *PS a*) ≤ (*case x of None* ⇒ *QN* | *Some a* ⇒ *QS a*)
⟨*proof*⟩

lemma *SPECT-Map-empty*[*simp*]: *SPECT Map.empty* ≤ *a*

⟨*proof*⟩

lemma *FAILT-SUP*: $(FAILT \in X) \Longrightarrow Sup\ X = FAILT$
⟨*proof*⟩

3 pointwise reasoning

named-theorems *refine-pw-simps*
⟨*ML*⟩

definition *nofailT* :: *'a nrest* ⇒ *bool* **where** $nofailT\ S \equiv S \neq FAILT$

definition *le-or-fail* :: *'a nrest* ⇒ *'a nrest* ⇒ *bool* (**infix** $\langle \leq_n \rangle$ *50*) **where**
$m \leq_n m' \equiv nofailT\ m \longrightarrow m \leq m'$

lemma *nofailT-simps*[*simp*]:
$nofailT\ FAILT \longleftrightarrow False$
$nofailT\ (REST\ X) \longleftrightarrow True$
$nofailT\ (RETURNT\ x) \longleftrightarrow True$
$nofailT\ SUCCEEDT \longleftrightarrow True$
⟨*proof*⟩

definition *inresT* :: *'a nrest* ⇒ *'a* ⇒ *nat* ⇒ *bool* **where**
$inresT\ S\ x\ t \equiv (case\ S\ of\ FAILi \Rightarrow True \mid REST\ X \Rightarrow (\exists t'.\ X\ x = Some\ t' \wedge enat\ t \leq t'))$

lemma *inresT-alt*: $inresT\ S\ x\ t \longleftrightarrow REST\ ([x \mapsto enat\ t]) \leq S$
⟨*proof*⟩

lemma *inresT-mono*: $inresT\ S\ x\ t \Longrightarrow t' \leq t \Longrightarrow inresT\ S\ x\ t'$
⟨*proof*⟩

lemma *inresT-RETURNT*[*simp*]: $inresT\ (RETURNT\ x)\ y\ t \longleftrightarrow t = 0 \wedge y = x$
⟨*proof*⟩

lemma *inresT-FAILT*[*simp*]: $inresT\ FAILT\ r\ t$
⟨*proof*⟩

lemma *fail-inresT*[*refine-pw-simps*]: $\neg\ nofailT\ M \Longrightarrow inresT\ M\ x\ t$
⟨*proof*⟩

lemma *pw-inresT-Sup*[*refine-pw-simps*]: $inresT\ (Sup\ X)\ r\ t \longleftrightarrow (\exists M \in X.\ \exists t' \geq t.\ inresT\ M\ r\ t')$
⟨*proof*⟩
lemma *inresT-REST*[*simp*]:
$inresT\ (REST\ X)\ x\ t \longleftrightarrow (\exists t' \geq t.\ X\ x = Some\ t')$
⟨*proof*⟩

lemma *pw-Sup-nofail*[*refine-pw-simps*]: *nofailT* (*Sup X*) ⟷ (∀ *x*∈*X*. *nofailT x*)
⟨*proof*⟩

lemma *inres-simps*[*simp*]:
inresT FAILT = (λ- -. *True*)
inresT SUCCEEDT = (λ- - . *False*)
⟨*proof*⟩

lemma *pw-le-iff*:
S ≤ *S'* ⟷ (*nofailT S'*⟶ (*nofailT S* ∧ (∀ *x t*. *inresT S x t* ⟶ *inresT S' x t*)))
⟨*proof*⟩

lemma *RETURN-le-RETURN-iff*[*simp*]: *RETURNT x* ≤ *RETURNT y* ⟷ *x*=*y*
⟨*proof*⟩

lemma *S*≤*S'* ⟹ *inresT S x t* ⟹ *inresT S' x t*
⟨*proof*⟩

lemma *pw-eq-iff*:
S=*S'* ⟷ (*nofailT S* = *nofailT S'* ∧ (∀ *x t*. *inresT S x t* ⟷ *inresT S' x t*))
⟨*proof*⟩

lemma *pw-flat-ge-iff*: *flat-ge S S'* ⟷
(*nofailT S*) ⟶ *nofailT S'* ∧ (∀ *x t*. *inresT S x t* ⟷ *inresT S' x t*)
⟨*proof*⟩

lemma *pw-eqI*:
assumes *nofailT S* = *nofailT S'*
assumes ⋀*x t*. *inresT S x t* ⟷ *inresT S' x t*
shows *S*=*S'*
⟨*proof*⟩

definition *consume M t* ≡ *case M of*
FAILi ⇒ *FAILT* |
REST X ⇒ *REST* (*map-option* ((+) *t*) *o* (*X*))

definition *SPEC P t* = *REST* (λ*v*. *if P v then Some* (*t v*) *else None*)

lemma *consume-mono*:
assumes *t*≤*t'* *M* ≤ *M'*
shows *consume M t* ≤ *consume M' t'*
⟨*proof*⟩

lemma *nofailT-SPEC*[*refine-pw-simps*]: *nofailT* (*SPEC a b*)
⟨*proof*⟩

lemma *inresT-SPEC*[*refine-pw-simps*]: *inresT* (*SPEC a b*) = (λ*x t*. *a x* ∧ *b x* ≥ *t*)
⟨*proof*⟩

4 Monad Operators

definition *bindT* :: *'b nrest* ⇒ (*'b* ⇒ *'a nrest*) ⇒ *'a nrest* **where**
bindT M f ≡ *case M of*
FAILi ⇒ *FAILT* |
REST X ⇒ *Sup* { (*case f x of FAILi* ⇒ *FAILT*
| *REST m2* ⇒ *REST* (*map-option* ((+) *t1*) *o* (*m2*)))
|*x t1*. *X x* = *Some t1*}

lemma *bindT-alt*: *bindT M f* = (*case M of*
FAILi ⇒ *FAILT* |
REST X ⇒ *Sup* { *consume* (*f x*) *t1* |*x t1*. *X x* = *Some t1*})
⟨*proof*⟩

lemma *bindT* (*REST X*) *f* =
(⨆ *x* ∈ *dom X*. *consume* (*f x*) (*the* (*X x*)))
⟨*proof*⟩

adhoc-overloading
Monad-Syntax.bind ⇌ *NREST.bindT*

lemma *bindT-FAIL*[*simp*]: *bindT FAILT g* = *FAILT*
⟨*proof*⟩

lemma *bindT SUCCEEDT f* = *SUCCEEDT*
⟨*proof*⟩

lemma *m r* = *Some* ∞ ⟹ *inresT* (*REST m*) *r t*
⟨*proof*⟩

lemma *pw-inresT-bindT-aux*: *inresT* (*bindT m f*) *r t* ⟷
(*nofailT m* ⟶ (∃ *r' t' t''*. *inresT m r' t'* ∧ *inresT* (*f r'*) *r t''* ∧ *t* ≤ *t'* + *t''*))
(**is** *?l* ⟷ *?r*)
⟨*proof*⟩

lemma *pw-inresT-bindT*[*refine-pw-simps*]: *inresT* (*bindT m f*) *r t* ⟷
(*nofailT m* ⟶ (∃ *r' t' t''*. *inresT m r' t'* ∧ *inresT* (*f r'*) *r t''* ∧ *t* = *t'* + *t''*))
⟨*proof*⟩

lemma *pw-bindT-nofailT*[*refine-pw-simps*]: *nofailT* (*bindT M f*) ⟷ (*nofailT M*
∧ (∀ *x t*. *inresT M x t* ⟶ *nofailT* (*f x*))) (**is** *?l* ⟷ *?r*)
⟨*proof*⟩

lemma *nat-plus-0-is-id*[*simp*]: ((+) (*0*::*enat*)) = *id* ⟨*proof*⟩

declare *map-option.id*[*simp*]

5 Monad Rules

lemma *nres-bind-left-identity*[*simp*]: $bindT\ (RETURNT\ x)\ f = f\ x$
⟨*proof*⟩

lemma *nres-bind-right-identity*[*simp*]: $bindT\ M\ RETURNT = M$
⟨*proof*⟩

lemma *nres-bind-assoc*[*simp*]: $bindT\ (bindT\ M\ (\lambda x.\ f\ x))\ g = bindT\ M\ (\%x.\ bindT\ (f\ x)\ g)$
⟨*proof*⟩

6 Monotonicity lemmas

lemma *bindT-mono*:
$m \le m' \Longrightarrow (\bigwedge x.\ (\exists\, t.\ inresT\ m\ x\ t) \Longrightarrow nofailT\ m' \Longrightarrow\ f\ x \le f'\ x) \Longrightarrow bindT\ m\ f \le bindT\ \ m'\ f'$
⟨*proof*⟩

lemma *bindT-mono'*[*refine-mono*]:
$m \le m' \Longrightarrow (\bigwedge x.\ \ f\ x \le f'\ x) \Longrightarrow bindT\ m\ f \le bindT\ \ m'\ f'$
⟨*proof*⟩

lemma *bindT-flat-mono*[*refine-mono*]:
$[\![\ flat\text{-}ge\ M\ M';\ \bigwedge x.\ flat\text{-}ge\ (f\ x)\ (f'\ x)\ ]\!] \Longrightarrow flat\text{-}ge\ (bindT\ M\ f)\ (bindT\ M'\ f')$
⟨*proof*⟩

6.1 Derived Program Constructs

6.1.1 Assertion

definition $iASSERT\ ret\ \Phi \equiv if\ \Phi\ then\ ret\ ()\ else\ top$

definition $ASSERT$ **where** $ASSERT \equiv iASSERT\ RETURNT$

lemma *ASSERT-True*[*simp*]: $ASSERT\ True = RETURNT\ ()$
⟨*proof*⟩
lemma *ASSERT-False*[*simp*]: $ASSERT\ False = FAILT$
⟨*proof*⟩

lemma *bind-ASSERT-eq-if*: $do\ \{\ ASSERT\ \Phi;\ m\ \} = (if\ \Phi\ then\ m\ else\ FAILT)$
⟨*proof*⟩

lemma *pw-ASSERT*[*refine-pw-simps*]:
$nofailT\ (ASSERT\ \Phi) \longleftrightarrow \Phi$
$inresT\ (ASSERT\ \Phi)\ x\ 0$
⟨*proof*⟩

lemma *ASSERT-refine*: $(Q \Longrightarrow P) \Longrightarrow ASSERT\ P \leq ASSERT\ Q$
⟨*proof*⟩

lemma *ASSERT-leI*: $\Phi \Longrightarrow (\Phi \Longrightarrow M \leq M') \Longrightarrow ASSERT\ \Phi \gg\!= (\lambda\text{-}.\ M) \leq M'$
⟨*proof*⟩

lemma *le-ASSERTI*: $(\Phi \Longrightarrow M \leq M') \Longrightarrow M \leq ASSERT\ \Phi \gg\!= (\lambda\text{-}.\ M')$
⟨*proof*⟩

lemma *inresT-ASSERT*: $inresT\ (ASSERT\ Q \gg\!= (\lambda\text{-}.\ M))\ x\ ta = (Q \longrightarrow inresT\ M\ x\ ta)$
⟨*proof*⟩

lemma *nofailT-ASSERT-bind*: $nofailT\ (ASSERT\ P \gg\!= (\lambda\text{-}.\ M)) \longleftrightarrow (P \wedge nofailT\ M)$
⟨*proof*⟩

6.2 SELECT

definition *emb'* **where** $\bigwedge Q\ T.\ emb'\ Q\ (T::'a \Rightarrow enat) = (\lambda x.\ if\ Q\ x\ then\ Some\ (T\ x)\ else\ None)$

abbreviation $emb\ Q\ t \equiv emb'\ Q\ (\lambda\text{-}.\ t)$

lemma *emb-eq-Some-conv*: $\bigwedge T.\ emb'\ Q\ T\ x = Some\ t' \longleftrightarrow (t'{=}T\ x \wedge Q\ x)$
⟨*proof*⟩

lemma *emb-le-Some-conv*: $\bigwedge T.\ Some\ t' \leq emb'\ Q\ T\ x \longleftrightarrow (\ t'{\leq}T\ x \wedge Q\ x)$
⟨*proof*⟩

lemma *SPEC-REST-emb'-conv*: $SPEC\ P\ t = REST\ (emb'\ P\ t)$
⟨*proof*⟩

lemma *SPECT-ub*: $T{\leq}T' \Longrightarrow SPECT\ (emb'\ M'\ T) \leq SPECT\ (emb'\ M'\ T')$
⟨*proof*⟩

Select some value with given property, or *None* if there is none.

definition $SELECT :: ('a \Rightarrow bool) \Rightarrow enat \Rightarrow 'a\ option\ nrest$
where $SELECT\ P\ tf \equiv if\ \exists x.\ P\ x\ then\ REST\ (emb\ (\lambda r.\ case\ r\ of\ Some\ p \Rightarrow P\ p \mid None \Rightarrow False)\ tf)$
$else\ REST\ [None \mapsto tf]$

lemma *inresT-SELECT-Some*: $inresT\ (SELECT\ Q\ tt)\ (Some\ x)\ t' \longleftrightarrow (Q\ x\ \wedge (t' \leq tt))$
⟨*proof*⟩

lemma *inresT-SELECT-None*: $inresT\ (SELECT\ Q\ tt)\ None\ t' \longleftrightarrow (\neg(\exists x.\ Q\ x) \wedge (t' \leq tt))$
⟨*proof*⟩

lemma *inresT-SELECT*[*refine-pw-simps*]:
inresT (*SELECT Q tt*) *x t'* ⟷ ((*case x of None* ⇒ ¬(∃ *x*. *Q x*) | *Some x* ⇒ *Q x*) ∧ (*t'* ≤ *tt*))
⟨*proof*⟩

lemma *nofailT-SELECT*[*refine-pw-simps*]: *nofailT* (*SELECT Q tt*)
⟨*proof*⟩

lemma *s1*: *SELECT P T* ≤ (*SELECT P T'*) ⟷ *T* ≤ *T'*
⟨*proof*⟩

lemma *s2*: *SELECT P T* ≤ (*SELECT P' T*) ⟷ (
(*Ex P'* ⟶ *Ex P*) ∧ (∀ *x*. *P x* ⟶ *P' x*))
⟨*proof*⟩

lemma *SELECT-refine*:
assumes ⋀*x'*. *P' x'* ⟹ ∃ *x*. *P x*
assumes ⋀*x*. *P x* ⟹ *P' x*
assumes *T* ≤ *T'*
shows *SELECT P T* ≤ (*SELECT P' T'*)
⟨*proof*⟩

7 RECT

definition *mono2 B* ≡ *flatf-mono-ge B* ∧ *mono B*

lemma *trimonoD-flatf-ge*: *mono2 B* ⟹ *flatf-mono-ge B*
⟨*proof*⟩

lemma *trimonoD-mono*: *mono2 B* ⟹ *mono B*
⟨*proof*⟩

definition *RECT B x* =
(*if mono2 B then* (*gfp B x*) *else* (*top*::'*a*::*complete-lattice*))

lemma *RECT-flat-gfp-def*: *RECT B x* =
(*if mono2 B then* (*flatf-gfp B x*) *else* (*top*::'*a*::*complete-lattice*))
⟨*proof*⟩

lemma *RECT-unfold*: ⟦*mono2 B*⟧ ⟹ *RECT B* = *B* (*RECT B*)
⟨*proof*⟩

definition *whileT* :: ('*a* ⇒ *bool*) ⇒ ('*a* ⇒ '*a nrest*) ⇒ '*a* ⇒ '*a nrest* **where**
whileT b c = *RECT* (λ*whileT s*. (*if b s then bindT* (*c s*) *whileT else RETURNT s*))

definition *whileIET* :: (*'a* ⇒ *bool*) ⇒ (*'a* ⇒ *nat*) ⇒ (*'a* ⇒ *bool*) ⇒ (*'a* ⇒ *'a nrest*) ⇒ *'a* ⇒ *'a nrest* **where**
⋀*E c*. *whileIET I E b c* = *whileT b c*

definition *whileTI* :: (*'a* ⇒ *enat option*) ⇒ ((*'a*×*'a*) *set*) ⇒ (*'a* ⇒ *bool*) ⇒ (*'a* ⇒ *'a nrest*) ⇒ *'a* ⇒ *'a nrest* **where**
whileTI I R b c s = *whileT b c s*

lemma *trimonoI*[*refine-mono*]:
⟦*flatf-mono-ge B*; *mono B*⟧ ⟹ *mono2 B*
⟨*proof*⟩

lemma *mono-fun-transform*[*refine-mono*]: (⋀*f g x*. (⋀*x*. *f x* ≤ *g x*) ⟹ *B f x* ≤ *B g x*) ⟹ *mono B*
⟨*proof*⟩

lemma *whileT-unfold*: *whileT b c* = (λ*s*. (*if b s then bindT* (*c s*) (*whileT b c*) *else RETURNT s*))
⟨*proof*⟩

lemma *RECT-mono*[*refine-mono*]:
assumes [*simp*]: *mono2 B'*
assumes *LE*: ⋀*F x*. (*B' F x*) ≤ (*B F x*)
shows (*RECT B' x*) ≤ (*RECT B x*)
⟨*proof*⟩

lemma *whileT-mono*:
assumes ⋀*x*. *b x* ⟹ *c x* ≤ *c' x*
shows (*whileT b c x*) ≤ (*whileT b c' x*)
⟨*proof*⟩

lemma *wf-fp-induct*:
assumes *fp*: ⋀*x*. *f x* = *B* (*f*) *x*
assumes *wf*: *wf R*
assumes ⋀*x D*. ⟦⋀*y*. (*y*,*x*)∈*R* ⟹ *P y* (*D y*)⟧ ⟹ *P x* (*B D x*)
shows *P x* (*f x*)
⟨*proof*⟩

lemma *RECT-wf-induct-aux*:
assumes *wf*: *wf R*
assumes *mono*: *mono2 B*
assumes (⋀*x D*. (⋀*y*. (*y*, *x*) ∈ *R* ⟹ *P y* (*D y*)) ⟹ *P x* (*B D x*))
shows *P x* (*RECT B x*)
⟨*proof*⟩

theorem *RECT-wf-induct*[*consumes 1*]:
assumes *RECT B x* = *r*

assumes *wf R*
and *mono2 B*
and $\bigwedge x\ D\ r.\ (\bigwedge y\ r.\ (y, x) \in R \Longrightarrow D\ y = r \Longrightarrow P\ y\ r) \Longrightarrow B\ D\ x = r \Longrightarrow P\ x\ r$
shows *P x r*

⟨*proof*⟩

definition *monadic-WHILEIT I b f s* ≡ *do* {
RECT (λ*D s*. *do* {
ASSERT (*I s*);
bv ← *b s*;
if bv then do {
s ← *f s*;
D s
} *else do* {*RETURNT s*}
}) *s*
}

8 Generalized Weakest Precondition

8.1 mm

definition *mm* :: (*'a* ⇒ *enat option*) ⇒ (*'a* ⇒ *enat option*) ⇒ (*'a* ⇒ *enat option*) **where**
mm R m = (λ*x*. (*case m x of None* ⇒ *Some* ∞
| *Some mt* ⇒
(*case R x of None* ⇒ *None* | *Some rt* ⇒ (*if rt* < *mt then None else Some* (*rt* − *mt*)))))

lemma *mm-mono*: *Q1 x* ≤ *Q2 x* ⟹ *mm Q1 M x* ≤ *mm Q2 M x*
⟨*proof*⟩

lemma *mm-antimono*: *M1 x* ≥ *M2 x* ⟹ *mm Q M1 x* ≤ *mm Q M2 x*
⟨*proof*⟩

lemma *mm-continous*: *mm* (λ*x*. *Inf* {*u*. ∃ *y*. *u* = *f y x*}) *m x* = *Inf* {*u*. ∃ *y*. *u* = *mm* (*f y*) *m x*}
⟨*proof*⟩

definition *mm2* :: (*enat option*) ⇒ (*enat option*) ⇒ (*enat option*) **where**
mm2 r m = (*case m of None* ⇒ *Some* ∞
| *Some mt* ⇒
(*case r of None* ⇒ *None* | *Some rt* ⇒ (*if rt* < *mt then None else Some* (*rt* − *mt*))))

lemma *mm-alt*: *mm R m x = mm2 (R x) (m x)* ⟨*proof*⟩

lemma *mm2-None*[*simp*]: *mm2 q None = Some ∞* ⟨*proof*⟩

lemma *mm2-Some0*[*simp*]: *mm2 q (Some 0) = q* ⟨*proof*⟩

lemma *mm2-antimono*: *x ≤ y ⟹ mm2 q x ≥ mm2 q y*
⟨*proof*⟩

lemma *mm2-contiuous2*:
assumes *∀ x∈X. t ≤ mm2 q x* **shows** *t ≤ mm2 q (Sup X)*
⟨*proof*⟩

lemma *fl*: *(a::enat) − b = ∞ ⟹ a = ∞*
⟨*proof*⟩

lemma *mm-inf1*: *mm R m x = Some ∞ ⟹ m x = None ∨ R x = Some ∞*
⟨*proof*⟩

lemma *mm-inf2*: *m x = None ⟹ mm R m x = Some ∞*
⟨*proof*⟩

lemma *mm-inf3*: *R x = Some ∞ ⟹ mm R m x = Some ∞*
⟨*proof*⟩

lemma *mm-inf*: *mm R m x = Some ∞ ⟷ m x = None ∨ R x = Some ∞*
⟨*proof*⟩

lemma *InfQ-E*: *Inf Q = Some t ⟹ None ∉ Q*
⟨*proof*⟩

lemma *InfQ-iff*: *(∃ t'≥enat t. Inf Q = Some t') ⟷ None ∉ Q ∧ Inf (Option.these Q) ≥ t*
⟨*proof*⟩

lemma *mm2-fst-None*[*simp*]: *mm2 None q = (case q of None ⇒ Some ∞ | - ⇒ None)*
⟨*proof*⟩

lemma *mm2-auxXX1*: *Some t ≤ mm2 (Q x) (Some t') ⟹ Some t' ≤ mm2 (Q x) (Some t)*
⟨*proof*⟩

8.2 mii

definition *mii* :: *('a ⇒ enat option) ⇒ 'a nrest ⇒ 'a ⇒ enat option* **where**
mii Qf M x = (case M of FAILi ⇒ None

$\mid$ *REST Mf* $\Rightarrow$ (*mm Qf Mf*) *x*)

lemma *mii-alt*: *mii Qf M x* = (*case M of FAILi* $\Rightarrow$ *None*
$\mid$ *REST Mf* $\Rightarrow$ (*mm2* (*Qf x*) (*Mf x*)))
⟨*proof*⟩

lemma *mii-continuous*: *mii* (λx. *Inf* {*f y x*|*y*. *True*}) *m x* = *Inf* {*mii* (%*x*. *f y x*) *m x*|*y*. *True*}
⟨*proof*⟩

lemma *mii-continuous2*: (*mii Q* (*Sup* {*F x t1* |*x t1*. *P x t1*}) *x* $\geq$ *t*) = ($\forall$ *y t1*. *P y t1* $\longrightarrow$ *mii Q* (*F y t1*) *x* $\geq$ *t*)
⟨*proof*⟩

lemma *mii-inf*: *mii Qf M x* = *Some* ∞ $\longleftrightarrow$ ($\exists$ *Mf*. *M* = *SPECT Mf* $\wedge$ (*Mf x* = *None* $\vee$ *Qf x* = *Some* ∞))
⟨*proof*⟩

lemma *miiFailt*: *mii Q FAILT x* = *None*
⟨*proof*⟩

8.3 lst - latest starting time

definition *lst* :: *'a nrest* $\Rightarrow$ (*'a* $\Rightarrow$ *enat option*) $\Rightarrow$ *enat option*
where *lst M Qf* = *Inf* { *mii Qf M x* | *x*. *True*}

lemma *T-alt-def*: *lst M Qf* = *Inf* ((*mii Qf M*) ‘ *UNIV*)
⟨*proof*⟩

lemma *T-pw*: *lst M Q* $\geq$ *t* $\longleftrightarrow$ ($\forall$ *x*. *mii Q M x* $\geq$ *t*)
⟨*proof*⟩

lemma *T-specifies-I*:
assumes *lst m Q* $\geq$ *Some 0* **shows** *m* $\leq$ *SPECT Q*
⟨*proof*⟩

lemma *T-specifies-rev*:
assumes *m* $\leq$ *SPECT Q* **shows** *lst m Q* $\geq$ *Some 0*
⟨*proof*⟩

lemma *T-specifies*: *lst m Q* $\geq$ *Some 0* = (*m* $\leq$ *SPECT Q*)
⟨*proof*⟩

lemma *pointwise-lesseq*:
fixes *x* :: *'a*::*order*
shows ($\forall$ *t*. *x* $\geq$ *t* $\longrightarrow$ *x'* $\geq$ *t*) $\Longrightarrow$ *x* $\leq$ *x'*
⟨*proof*⟩

8.4 pointwise reasoning about lst via nres3

definition *nres3* **where** *nres3 Q M x t* $\longleftrightarrow$ *mii Q M x* $\geq$ *t*

lemma *pw-T-le*:
assumes $\bigwedge t.$ $(\forall x.$ *nres3 Q M x t*$) \Longrightarrow (\forall x.$ *nres3 Q' M' x t*$)$
shows *lst M Q* $\leq$ *lst M' Q'*
⟨*proof*⟩

lemma assumes $\bigwedge t.$ $(\forall x.$ *nres3 Q M x t*$) = (\forall x.$ *nres3 Q' M' x t*$)$
shows *pw-T-eq-iff*: *lst M Q* = *lst M' Q'*
⟨*proof*⟩

lemma assumes $\bigwedge t.$ $(\forall x.$ *nres3 Q M x t*$) \Longrightarrow (\forall x.$ *nres3 Q' M' x t*$)$
$\bigwedge t.$ $(\forall x.$ *nres3 Q' M' x t*$) \Longrightarrow (\forall x.$ *nres3 Q M x t*$)$
shows *pw-T-eqI*: *lst M Q* = *lst M' Q'*
⟨*proof*⟩

lemma *lem*: $\forall$ *t1*. *M y* = *Some t1* $\longrightarrow$ *t* $\leq$ *mii Q* (*SPECT* (*map-option* ((+) *t1*) $\circ$ *x2*)) *x* $\Longrightarrow$ *f y* = *SPECT x2* $\Longrightarrow$ *t* $\leq$ *mii* (λ*y*. *mii Q* (*f y*) *x*) (*SPECT M*) *y*
⟨*proof*⟩

lemma *diff-diff-add-enat*:*a* − (*b*+*c*) = *a* − *b* − (*c*::*enat*)
⟨*proof*⟩

lemma *lem2*: *t* $\leq$ *mii* (λ*y*. *mii Q* (*f y*) *x*) (*SPECT M*) *y* $\Longrightarrow$ *M y* = *Some t1* $\Longrightarrow$ *f y* = *SPECT fF* $\Longrightarrow$ *t* $\leq$ *mii Q* (*SPECT* (*map-option* ((+) *t1*) $\circ$ *fF*)) *x*
⟨*proof*⟩

lemma fixes *m* :: *'b nrest*
shows *mii-bindT*: (*t* $\leq$ *mii Q* (*bindT m f*) *x*) $\longleftrightarrow$ ($\forall$ *y*. *t* $\leq$ *mii* (λ*y*. *mii Q* (*f y*) *x*) *m y*)
⟨*proof*⟩

lemma *nres3-bindT*: $(\forall x.$ *nres3 Q* (*bindT m f*) *x t*$) = (\forall y.$ *nres3* (λ*y*. *lst* (*f y*) *Q*) *m y t*$)$
⟨*proof*⟩

8.5 rules for lst

lemma *T-bindT*: *lst* (*bindT M f*) *Q* = *lst M* (λ*y*. *lst* (*f y*) *Q*)
⟨*proof*⟩

lemma *T-REST*: *lst* (*REST* [*x*↦*t*]) *Q* = *mm2* (*Q x*) (*Some t*)
⟨*proof*⟩

lemma *T-RETURNT*: *lst* (*RETURNT x*) *Q* = *Q x*
⟨*proof*⟩

lemma *T-SELECT*:
assumes
∀ *x*. ¬ *P x* ⟹ *Some tt* ≤ *lst* (*SPECT* [*None* ↦ *tf*]) *Q*
and *p*: (⋀*x*. *P x* ⟹ *Some tt* ≤ *lst* (*SPECT* [*Some x* ↦ *tf*]) *Q*)
shows *Some tt* ≤ *lst* (*SELECT P tf*) *Q*
⟨*proof*⟩

9 consequence rules

lemma *aux1*: *Some t* ≤ *mm2 Q* (*Some t'*) ⟷ *Some* (*t*+*t'*) ≤ *Q*
⟨*proof*⟩

lemma *aux1a*: (∀ *x t''*. *Q' x* = *Some t''* ⟶ (*Q x*) ≥ *Some* (*t* + *t''*))
= (∀ *x*. *mm2* (*Q x*) (*Q' x*) ≥ *Some t*)
⟨*proof*⟩

lemma *T-conseq4*:
assumes
lst f Q' ≥ *Some t'*
⋀*x t'' M*. *Q' x* = *Some t''* ⟹ (*Q x*) ≥ *Some* ((*t* − *t'*) + *t''*)
shows *lst f Q* ≥ *Some t*
⟨*proof*⟩

lemma *T-conseq6*:
assumes
lst f Q' ≥ *Some t*
⋀*x t'' M*. *f* = *SPECT M* ⟹ *M x* ≠ *None* ⟹ *Q' x* = *Some t''* ⟹ (*Q x*) ≥ *Some* (*t''*)
shows *lst f Q* ≥ *Some t*
⟨*proof*⟩

lemma *T-conseq6'*:
assumes
lst f Q' ≥ *Some t*
⋀*x t'' M*. *f* = *SPECT M* ⟹ *M x* ≠ *None* ⟹ (*Q x*) ≥ *Q' x*
shows *lst f Q* ≥ *Some t*
⟨*proof*⟩

lemma *T-conseq5*:
assumes
lst f Q' ≥ *Some t'*
⋀*x t'' M*. *f* = *SPECT M* ⟹ *M x* ≠ *None* ⟹ *Q' x* = *Some t''* ⟹ (*Q x*) ≥ *Some* ((*t* − *t'*) + *t''*)

shows *lst f Q* ≥ *Some t*
⟨*proof*⟩

lemma *T-conseq3*:
assumes
lst f Q′ ≥ *Some t′*
⋀*x*. *mm2* (*Q x*) (*Q′ x*) ≥ *Some* (*t* − *t′*)
shows *lst f Q* ≥ *Some t*
⟨*proof*⟩

10 Experimental Hoare reasoning

named-theorems *vcg-rules*

method *vcg* **uses** *rls* = ((*rule rls vcg-rules*[*THEN T-conseq4*] | *clarsimp simp*: *emb-eq-Some-conv emb-le-Some-conv T-bindT T-RETURNT*)+)

experiment
begin

definition *P* **where** *P f g* = *bindT f* (λ*x*. *bindT g* (λ*y*. *RETURNT* (*x*+(*y*::*nat*))))

lemma assumes
f-spec[*vcg-rules*]: *lst f* (*emb′* (λ*x*. *x* > *2*) (*enat o* ((∗) *2*))) ≥ *Some 0*
and
g-spec[*vcg-rules*]: *lst g* (*emb′* (λ*x*. *x* > *2*) (*enat*)) ≥ *Some 0*
shows *lst* (*P f g*) (*emb′* (λ*x*. *x* > *5*) (*enat o* (∗) *3*)) ≥ *Some 0*
⟨*proof*⟩
end

11 VCG

named-theorems *vcg-simp-rules*
lemmas [*vcg-simp-rules*] = *T-RETURNT*

lemma *TbindT-I*: *Some t* ≤ *lst M* (λ*y*. *lst* (*f y*) *Q*) ⟹ *Some t* ≤ *lst* (*M* ⤜ *f*) *Q*
⟨*proof*⟩

method *vcg′* **uses** *rls* = ((*rule rls TbindT-I vcg-rules*[*THEN T-conseq6*] | *clarsimp split*: *if-splits simp*: *vcg-simp-rules*)+)

lemma *mm2-refl*: *A* < ∞ ⟹ *mm2* (*Some A*) (*Some A*) = *Some 0*
⟨*proof*⟩

definition *mm3* **where**
mm3 t A = (*case A of None* ⇒ *None* | *Some t′* ⇒ *if t′*≤*t then Some* (*enat* (*t*−*t′*)) *else None*)

lemma *mm3-same*[*simp*]: *mm3 t0* (*Some t0*) = *Some 0* ⟨*proof*⟩

lemma *mm3-Some-conv*: (*mm3 t0 A* = *Some t*) = (∃ *t'*. *A* = *Some t'* ∧ *t0* ≥ *t'* ∧ *t*=*t0*−*t'*)
⟨*proof*⟩

lemma *mm3-None*[*simp*]: *mm3 t0 None* = *None* ⟨*proof*⟩

lemma *T-FAILT*[*simp*]: *lst FAILT Q* = *None*
⟨*proof*⟩

definition *progress m* ≡ ∀ *s' M*. *m* = *SPECT M* ⟶ *M s'* ≠ *None* ⟶ *M s'* > *Some 0*
lemma *progressD*: *progress m* ⟹ *m*=*SPECT M* ⟹ *M s'* ≠ *None* ⟹ *M s'* > *Some 0*
⟨*proof*⟩

lemma *progress-FAILT*[*simp*]: *progress FAILT* ⟨*proof*⟩

11.1 Progress rules

named-theorems *progress-rules*

lemma *progress-SELECT-iff*: *progress* (*SELECT P t*) ⟷ *t* > *0*
⟨*proof*⟩

lemmas [*progress-rules*] = *progress-SELECT-iff*[*THEN iffD2*]

lemma *progress-REST-iff*: *progress* (*REST* [*x* ↦ *t*]) ⟷ *t*>*0*
⟨*proof*⟩

lemmas [*progress-rules*] = *progress-REST-iff*[*THEN iffD2*]

lemma *progress-ASSERT-bind*[*progress-rules*]: ⟦Φ ⟹ *progress* (*f* ()) ⟧ ⟹ *progress* (*ASSERT* Φ⤜*f*)
⟨*proof*⟩

lemma *progress-SPECT-emb*[*progress-rules*]: *t* > *0* ⟹ *progress* (*SPECT* (*emb P t*)) ⟨*proof*⟩

lemma *Sup-Some*: *Sup* (*S*::*enat option set*) = *Some e* ⟹ ∃ *x*∈*S*. (∃ *i*. *x* = *Some i*)
⟨*proof*⟩

lemma *progress-bind*[*progress-rules*]: **assumes** *progress m* ∨ (∀ *x*. *progress* (*f x*))
shows *progress* (*m*⤜*f*)
⟨*proof*⟩

lemma *mm2SomeleSome-conv*: *mm2* (*Qf*) (*Some t*) ≥ *Some 0* ⟷ *Qf* ≥ *Some t*
⟨*proof*⟩

12 rules for whileT

lemma
assumes *whileT b c s = r*
assumes *IS*[*vcg-rules*]: ⋀*s t'*. *I s = Some t'* ⟹ *b s*
⟹ *lst* (*c s*) (λ*s'*. *if* (*s'*,*s*)∈*R then I s' else None*) ≥ *Some t'*

assumes *I s = Some t*
assumes *wf*: *wf R*
shows *whileT-rule''*: *lst r* (λ*x*. *if b x then None else I x*) ≥ *Some t*
⟨*proof*⟩

lemma
fixes *I* :: *'a* ⇒ *nat option*
assumes *whileT b c s0 = r*
assumes *progress*: ⋀*s*. *progress* (*c s*)
assumes *IS*[*vcg-rules*]: ⋀*s t t'*. *I s = Some t* ⟹ *b s* ⟹
lst (*c s*) (λ*s'*. *mm3 t* (*I s'*)) ≥ *Some 0*

assumes [*simp*]: *I s0 = Some t0*

shows *whileT-rule'''*: *lst r* (λ*x*. *if b x then None else mm3 t0* (*I x*)) ≥ *Some 0*
⟨*proof*⟩

lemma *whileIET-rule*[*THEN T-conseq6*, *vcg-rules*]:
fixes *E*
assumes
(⋀*s t t'*.
(*if I s then Some* (*E s*) *else None*) = *Some t* ⟹
b s ⟹ *Some 0* ≤ *lst* (*C s*) (λ*s'*. *mm3 t* (*if I s' then Some* (*E s'*) *else None*)))
⋀*s*. *progress* (*C s*)
I s0
shows *Some 0* ≤ *lst* (*whileIET I E b C s0*) (λ*x*. *if b x then None else mm3* (*E s0*) (*if I x then Some* (*E x*) *else None*))
⟨*proof*⟩

lemma *transf*:
assumes *I s* ⟹ *b s* ⟹ *Some 0* ≤ *lst* (*C s*) (λ*s'*. *mm3* (*E s*) (*if I s' then Some* (*E s'*) *else None*))
shows
(*if I s then Some* (*E s*) *else None*) = *Some t* ⟹
b s ⟹ *Some 0* ≤ *lst* (*C s*) (λ*s'*. *mm3 t* (*if I s' then Some* (*E s'*) *else None*))
⟨*proof*⟩

lemma *whileIET-rule'*:
fixes *E*
assumes
(⋀*s t t'*. *I s* ⟹ *b s* ⟹ *Some 0* ≤ *lst* (*C s*) (λ*s'*. *mm3* (*E s*) (*if I s' then Some* (*E s'*) *else None*)))
⋀*s*. *progress* (*C s*)
I s0
shows *Some 0* ≤ *lst* (*whileIET I E b C s0*) (λ*x*. *if b x then None else mm3* (*E s0*) (*if I x then Some* (*E x*) *else None*))
⟨*proof*⟩

13 some Monadic Refinement Automation

⟨*ML*⟩

end
theory *DataRefinement*
imports *NREST*
begin

13.1 Data Refinement

lemma {(*1*,*2*),(*2*,*4*)}``{*1*,*2*}={*2*,*4*} ⟨*proof*⟩

definition *conc-fun* (‹⇓›) **where**
conc-fun R m ≡ *case m of FAILi* ⇒ *FAILT* | *REST X* ⇒ *REST* (λ*c*. *Sup* {*X a*| *a*. (*c*,*a*)∈*R*})

definition *abs-fun* (‹⇑›) **where**
abs-fun R m ≡ *case m of FAILi* ⇒ *FAILT*
| *REST X* ⇒ *if dom X*⊆*Domain R then REST* (λ*a*. *Sup* {*X c*| *c*. (*c*,*a*)∈*R*}) *else FAILT*

lemma
conc-fun-FAIL[*simp*]: ⇓*R FAILT* = *FAILT* **and**
conc-fun-RES: ⇓*R* (*REST X*) = *REST* (λ*c*. *Sup* {*X a*| *a*. (*c*,*a*)∈*R*})
⟨*proof*⟩

lemma *nrest-Rel-mono*: *A* ≤ *B* ⟹ ⇓ *R A* ≤ ⇓ *R B*
⟨*proof*⟩

13.1.1 Examples

definition *Rset* **where** *Rset* = { (*c*,*a*)| *c a*. *set c* = *a*}

lemma *RETURNT* [*1*,*2*,*3*] ≤ ⇓*Rset* (*RETURNT* {*1*,*2*,*3*})
⟨*proof*⟩

lemma *RETURNT [1,2,3] ≤ ⇓Rset (RETURNT {1,2,3})*
⟨*proof*⟩

definition *Reven* **where** *Reven = { (c,a)| c a. even c = a}*

lemma *RETURNT 3 ≤ ⇓Reven (RETURNT False)*
⟨*proof*⟩

lemma *m ≤ ⇓Id m*
⟨*proof*⟩

lemma *m ≤ ⇓{} m ⟷ (m=FAILT ∨ m = SPECT bot)*
⟨*proof*⟩

lemma *abs-fun-simps*[*simp*]:
⇑R FAILT = FAILT
dom X⊆Domain R ⟹ ⇑R (REST X) = REST (λa. Sup {X c| c. (c,a)∈R})
¬(dom X⊆Domain R) ⟹ ⇑R (REST X) = FAILT
⟨*proof*⟩

context fixes *R* **assumes** *SV*: *single-valued R* **begin**

lemma
fixes *m :: 'b ⇒ enat option*
shows
Sup-sv: *(c, a') ∈ R ⟹ Sup {m a| a. (c,a) ∈ R} = m a'*
⟨*proof*⟩

lemma *indomD*: *M c = Some y ⟹ dom M ⊆ Domain R ⟹ (∃ a. (c,a)∈R)*
⟨*proof*⟩

lemma *conc-abs-swap*: *m' ≤ ⇓R m ⟷ ⇑R m' ≤ m*
⟨*proof*⟩

lemma
fixes *m :: 'b ⇒ enat option*
shows
Inf-sv: *(c, a') ∈ R ⟹ Inf {m a| a. (c,a) ∈ R} = m a'*
⟨*proof*⟩

lemma *ac-galois*: *galois-connection (⇑R) (⇓R)*
⟨*proof*⟩

lemma
fixes *m :: 'b ⇒ enat option*
shows *Sup-some-svD*: *Sup {m a |a. (c, a) ∈ R} = Some t' ⟹ (∃ a. (c,a)∈R ∧*

$m\ a = Some\ t'$)
⟨*proof*⟩

end

lemma *pw-abs-nofail*[*refine-pw-simps*]:
$nofailT\ (\Uparrow R\ M) \longleftrightarrow (nofailT\ M \wedge (\forall x.\ (\exists t.\ inresT\ M\ x\ t) \longrightarrow x \in Domain\ R))$
(**is** *?l* $\longleftrightarrow$ *?r*)
⟨*proof*⟩

lemma *pw-conc-nofail*[*refine-pw-simps*]:
$nofailT\ (\Downarrow R\ S) = nofailT\ S$
⟨*proof*⟩

lemma *single-valued A* $\Longrightarrow$ *single-valued B* $\Longrightarrow$ *single-valued* $(A\ O\ B)$
⟨*proof*⟩

lemma *Sup-enatoption-less2*: $Sup\ X = Some\ \infty \Longrightarrow (\exists x.\ Some\ x \in X \wedge enat\ t < x)$
⟨*proof*⟩

lemma *pw-conc-inres*[*refine-pw-simps*]:
$inresT\ (\Downarrow R\ S')\ s\ t = (nofailT\ S'$
$\longrightarrow ((\exists s'.\ (s,s') \in R \wedge inresT\ S'\ s'\ t)))$ (**is** *?l* $\longleftrightarrow$ *?r*)
⟨*proof*⟩

lemma *sv-the*: *single-valued R* $\Longrightarrow (c,a) \in R \Longrightarrow (THE\ a.\ (c,\ a) \in R) = a$
⟨*proof*⟩

lemma
conc-fun-RES-sv:
assumes *SV*: *single-valued R*
shows $\Downarrow R\ (REST\ X) = REST\ (\lambda c.\ if\ c \in Domain\ R\ then\ X\ (THE\ a.\ (c,a) \in R)\ else\ None\ )$
⟨*proof*⟩

lemma *conc-fun-mono*[*simp*, *intro*!]:
shows *mono* $(\Downarrow R)$
⟨*proof*⟩

lemma *conc-fun-R-mono*:
assumes $R \subseteq R'$
shows $\Downarrow R\ M \leq \Downarrow R'\ M$
⟨*proof*⟩

lemma *SupSup-2*: *Sup* $\{m\ a\ |a.\ (c,\ a) \in R\ O\ S\} =$ *Sup* $\{m\ a\ |a\ b.\ (b,a) \in S \wedge (c,b) \in R\ \}$
⟨*proof*⟩

lemma
fixes $m :: 'a \Rightarrow enat\ option$
shows *SupSup*: *Sup* $\{Sup\ \{m\ aa\ |aa.\ P\ a\ aa\ c\}\ |a.\ Q\ a\ c\} = Sup\ \{m\ aa\ |a\ aa.\ P\ a\ aa\ c \wedge Q\ a\ c\}$
⟨*proof*⟩

lemma
fixes $m :: 'a \Rightarrow enat\ option$
shows
SupSup-1: *Sup* $\{Sup\ \{m\ aa\ |aa.\ (a,\ aa) \in S\}\ |a.\ (c,\ a) \in R\} = Sup\ \{m\ aa\ |a\ aa.\ (a,aa) \in S \wedge (c,a) \in R\}$
⟨*proof*⟩

lemma *conc-fun-chain*:
shows $\Downarrow R\ (\Downarrow S\ M) = \Downarrow(R\ O\ S)\ M$
⟨*proof*⟩

lemma *conc-fun-chain-sv*:
assumes *SVR*: *single-valued R* **and** *SVS*: *single-valued S*
shows $\Downarrow R\ (\Downarrow S\ M) = \Downarrow(R\ O\ S)\ M$
⟨*proof*⟩

lemma *conc-Id*[*simp*]: $\Downarrow Id = id$
⟨*proof*⟩

lemma *conc-fun-fail-iff*[*simp*]:
$\Downarrow R\ S = FAILT \longleftrightarrow S{=}FAILT$
$FAILT = \Downarrow R\ S \longleftrightarrow S{=}FAILT$
⟨*proof*⟩

lemma *conc-trans*[*trans*]:
assumes *A*: $C \leq \Downarrow R\ B$ **and** *B*: $B \leq \Downarrow R'\ A$
shows $C \leq \Downarrow R\ (\Downarrow R'\ A)$
⟨*proof*⟩

lemma *conc-trans-sv*:
assumes *SV*: *single-valued R single-valued* R'
and *A*: $C \leq \Downarrow R\ B$ **and** *B*: $B \leq \Downarrow R'\ A$
shows $C \leq \Downarrow R\ (\Downarrow R'\ A)$
⟨*proof*⟩

WARNING: The order of the single statements is important here!

lemma *conc-trans-additional*[*trans*]:
assumes *single-valued R*
shows
$\bigwedge A\ B\ C.\ A{\leq}{\Downarrow}R\ \ B \Longrightarrow B{\leq}\ \ \ C \Longrightarrow A{\leq}{\Downarrow}R\ \ C$
$\bigwedge A\ B\ C.\ A{\leq}{\Downarrow}Id\ B \Longrightarrow B{\leq}{\Downarrow}R\ \ C \Longrightarrow A{\leq}{\Downarrow}R\ \ C$
$\bigwedge A\ B\ C.\ A{\leq}{\Downarrow}R\ \ B \Longrightarrow B{\leq}{\Downarrow}Id\ C \Longrightarrow A{\leq}{\Downarrow}R\ \ C$

$\bigwedge A\ B\ C.\ A{\leq}{\Downarrow}Id\ B \Longrightarrow B{\leq}{\Downarrow}Id\ C \Longrightarrow A{\leq}\ \ \ C$
$\bigwedge A\ B\ C.\ A{\leq}{\Downarrow}Id\ B \Longrightarrow B{\leq}\ \ \ C \Longrightarrow A{\leq}\ \ \ C$
$\bigwedge A\ B\ C.\ A{\leq}\ \ \ B \Longrightarrow B{\leq}{\Downarrow}Id\ C \Longrightarrow A{\leq}\ \ \ C$
⟨*proof*⟩

lemma *RETURNT-refine*:
assumes $(x,x')\in R$
shows $RETURNT\ x \leq {\Downarrow}R\ (RETURNT\ x')$
⟨*proof*⟩

lemma *bindT-refine'*:
fixes R' :: $('a\times'b)$ *set* **and** R::$('c\times'd)$ *set*
assumes *R1*: $M \leq {\Downarrow}\ R'\ M'$
assumes *R2*: $\bigwedge x\ x'\ t$. ⟦ $(x,x')\in R'$; *inresT* $M\ x\ t$; *inresT* $M'\ x'\ t$;
nofailT M; *nofailT* M'
⟧ $\Longrightarrow f\ x \leq {\Downarrow}\ R\ (f'\ x')$
shows *bindT* $M\ (\lambda x.\ f\ x) \leq {\Downarrow}\ R\ (bindT\ M'\ (\lambda x'.\ f'\ x'))$
⟨*proof*⟩

lemma *bindT-refine*:
fixes R' :: $('a\times'b)$ *set* **and** R::$('c\times'd)$ *set*
assumes *R1*: $M \leq {\Downarrow}\ R'\ M'$
assumes *R2*: $\bigwedge x\ x'.$ ⟦ $(x,x')\in R'$ ⟧
$\Longrightarrow f\ x \leq {\Downarrow}\ R\ (f'\ x')$
shows *bindT* $M\ (\lambda x.\ f\ x) \leq {\Downarrow}\ R\ (bind\ M'\ (\lambda x'.\ f'\ x'))$
⟨*proof*⟩

13.2 WHILET refine

lemma *RECT-refine*:
assumes *M*: *mono2 body*
assumes *R0*: $(x,x')\in R$
assumes *RS*: $\bigwedge f\ f'\ x\ x'.$ ⟦ $\bigwedge x\ x'.\ (x,x')\in R \Longrightarrow f\ x \leq {\Downarrow}S\ (f'\ x')$; $(x,x')\in R$ ⟧
$\Longrightarrow body\ f\ x \leq {\Downarrow}S\ (body'\ f'\ x')$
shows $RECT\ (\lambda f\ x.\ body\ f\ x)\ x \leq {\Downarrow}S\ (RECT\ (\lambda f'\ x'.\ body'\ f'\ x')\ x')$
⟨*proof*⟩

lemma *WHILET-refine*:
assumes *R0*: $(x,x')\in R$
assumes *COND-REF*: $\bigwedge x\ x'.$ ⟦ $(x,x')\in R$ ⟧ $\Longrightarrow b\ x = b'\ x'$
assumes *STEP-REF*:
$\bigwedge x\ x'.$ ⟦ $(x,x')\in R$; $b\ x$; $b'\ x'$ ⟧ $\Longrightarrow f\ x \leq {\Downarrow}R\ (f'\ x')$

shows *whileT b f x* ≤ ⇓*R* (*whileT b′ f′ x′*)
⟨*proof*⟩

lemma *SPECT-refines-conc-fun′*:
assumes *a*: ⋀*m c*. *M* = *SPECT m*
⟹ *c* ∈ *dom n* ⟹ (∃ *a*. (*c*,*a*)∈*R* ∧ *n c* ≤ *m a*)
shows *SPECT n* ≤ ⇓ *R M*
⟨*proof*⟩

lemma *SPECT-refines-conc-fun*:
assumes *a*: ⋀*m c*. (∃ *a*. (*c*,*a*)∈*R* ∧ *n c* ≤ *m a*)
shows *SPECT n* ≤ ⇓ *R* (*SPECT m*)
⟨*proof*⟩

lemma *conc-fun-br*: ⇓ (*br* α *I1*) (*SPECT* (*emb I2 t*))
= (*SPECT* (*emb* (λ*x*. *I1 x* ∧ *I2* (α *x*)) *t*))
⟨*proof*⟩

end
theory *RefineMonadicVCG*
imports *NREST DataRefinement*
Case-Labeling.Case-Labeling
begin

— Might look similar to *repeat-new* from *HOL−Eisbach.Eisbach*, however the placement of the *?* is different, in particular that means this method is allowed to failed
method *repeat-all-new* **methods** *m* = (*m*;*repeat-all-new* ‹*m*›)*?*

lemma *R-intro*: *A* ≤ ⇓*Id B* ⟹ *A* ≤ *B* ⟨*proof*⟩

13.3 ASSERT

lemma *le-R-ASSERTI*: (Φ ⟹ *M* ≤ ⇓ *R M′*) ⟹ *M* ≤ ⇓ *R* (*ASSERT* Φ ⤜ (λ-. *M′*))
⟨*proof*⟩

lemma *T-ASSERT*[*vcg-simp-rules*]: *Some t* ≤ *lst* (*ASSERT* Φ) *Q* ⟷ *Some t* ≤ *Q* () ∧ Φ
⟨*proof*⟩

lemma *T-ASSERT-I*: (Φ ⟹ *Some t* ≤ *Q* ()) ⟹ Φ ⟹ *Some t* ≤ *lst* (*ASSERT* Φ) *Q*
⟨*proof*⟩

lemma *T-RESTemb-iff*: *Some t′*
≤ *lst* (*REST* (*emb′ P t*)) *Q* ⟷ (∀ *x*. *P x* ⟶ *Some* (*t′* + *t x*) ≤ *Q x*)
⟨*proof*⟩

lemma *T-RESTemb*: $(\bigwedge x.\ P\ x \Longrightarrow Some\ (t' + t\ x) \leq Q\ x)$
$\Longrightarrow Some\ t' \leq lst\ (REST\ (emb'\ P\ t))\ Q$
⟨*proof*⟩

lemma *T-SPEC*: $(\bigwedge x.\ P\ x \Longrightarrow Some\ (t' + t\ x) \leq Q\ x)$
$\Longrightarrow Some\ t' \leq lst\ (SPEC\ P\ t)\ Q$
⟨*proof*⟩

lemma *T-SPECT-I*: $(Some\ (t' + t\ ) \leq Q\ x)$
$\Longrightarrow Some\ t' \leq lst\ (SPECT\ [\ x \mapsto t])\ Q$
⟨*proof*⟩

lemma *mm2-map-option*:
assumes $Some\ (t'+t) \leq mm2\ (Q\ x)\ (x2\ x)$
shows $Some\ t' \leq mm2\ (Q\ x)\ (map\text{-}option\ ((+)\ t)\ (x2\ x))$
⟨*proof*⟩

lemma *T-consume*: $(Some\ (t' + t) \leq lst\ M\ Q)$
$\Longrightarrow Some\ t' \leq lst\ (consume\ M\ t)\ Q$
⟨*proof*⟩

definition $valid\ t\ Q\ M = (Some\ t \leq lst\ M\ Q)$

13.4 VCG splitter

⟨*ML*⟩

13.5 mm3 and emb

lemma *Some-eq-mm3-Some-conv*[*vcg-simp-rules*]: $Some\ t = mm3\ t'\ (Some\ t'') \longleftrightarrow (t'' \leq t' \wedge t = enat\ (t' - t''))$
⟨*proof*⟩

lemma *Some-eq-mm3-Some-conv'*: $mm3\ t'\ (Some\ t'') = Some\ t \longleftrightarrow (t'' \leq t' \wedge t = enat\ (t' - t''))$
⟨*proof*⟩

lemma *Some-le-emb'-conv*[*vcg-simp-rules*]: $Some\ t \leq emb'\ Q\ ft\ x \longleftrightarrow Q\ x \wedge t \leq ft\ x$
⟨*proof*⟩

lemma *Some-eq-emb'-conv*[*vcg-simp-rules*]: $emb'\ Q\ tf\ s = Some\ t \longleftrightarrow (Q\ s \wedge t = tf\ s)$
⟨*proof*⟩

13.6 Setup Labeled VCG

context
begin
interpretation *Labeling-Syntax* ⟨*proof*⟩

lemma *LCondRule*:
fixes *IC CT* **defines** $CT' \equiv (''cond'', IC, []) \# CT$
assumes $b \Longrightarrow C\langle Suc\ IC, (''the'', IC, []) \# (''cond'', IC, []) \# CT, OC1 : valid\ t\ Q\ c1\ \rangle$
and $\sim b \Longrightarrow C\langle Suc\ OC1, (''els'', Suc\ OC1, []) \# (''cond'', IC, []) \# CT, OC : valid\ t\ Q\ c2\ \rangle$
shows $C\langle IC, CT, OC : valid\ t\ Q\ (if\ b\ then\ c1\ else\ c2)\rangle$
⟨*proof*⟩

lemma *LouterCondRule*:
fixes *IC CT* **defines** $CT' \equiv (''cond2'', IC, []) \# CT$
assumes $b \Longrightarrow C\langle Suc\ IC, (''the'', IC, []) \# (''cond2'', IC, []) \# CT, OC1 : t \leq A\ \rangle$
and $\sim b \Longrightarrow C\langle Suc\ OC1, (''els'', Suc\ OC1, []) \# (''cond2'', IC, []) \# CT, OC : t \leq B\ \rangle$
shows $C\langle IC, CT, OC : t \leq (if\ b\ then\ A\ else\ B)\rangle$
⟨*proof*⟩

lemma *While*:
assumes $I\ s0\ \ (\bigwedge s.\ I\ s \Longrightarrow b\ s \Longrightarrow Some\ 0 \leq lst\ (C\ s)\ (\lambda s'.\ mm3\ (E\ s)\ (if\ I\ s'\ then\ Some\ (E\ s')\ else\ None)))$
$(\bigwedge s.\ progress\ (C\ s))$
$(\bigwedge x.\ \neg\ b\ x \Longrightarrow\ I\ x \Longrightarrow\ (E\ x) \leq (E\ s0) \Longrightarrow\ \ Some\ (t + enat\ ((E\ s0) - E\ x)) \leq Q\ x)$
shows $Some\ t \leq lst\ (whileIET\ I\ E\ b\ C\ s0)\ Q$
⟨*proof*⟩

definition *monadic-WHILE b f s* ≡ *do* {
RECT (λ*D s*. *do* {
bv ← *b s*;
if bv then do {
s ← *f s*;
D s
} *else do* {*RETURNT s*}
}) *s*
}

lemma *monadic-WHILE-mono*:
assumes $\bigwedge x.\ bm\ x \leq bm'\ x$
and $\bigwedge x\ t.\ nofailT\ (bm'\ x) \Longrightarrow inresT\ (bm\ x)\ True\ t \Longrightarrow c\ x \leq c'\ x$
shows $(monadic\text{-}WHILE\ bm\ c\ x) \leq (monadic\text{-}WHILE\ bm'\ c'\ x)$
⟨*proof*⟩

lemma *z*: $inresT\ A\ x\ t \Longrightarrow A \leq B \Longrightarrow inresT\ B\ x\ t$

⟨*proof*⟩

lemma *monadic-WHILE-mono'*:
assumes
⋀*x*. *bm x* ≤ *bm' x*
and ⋀*x t*. *nofailT* (*bm' x*) ⟹ *inresT* (*bm' x*) *True t* ⟹ *c x* ≤ *c' x*
shows (*monadic-WHILE bm c x*) ≤ (*monadic-WHILE bm' c' x*)
⟨*proof*⟩

lemma *monadic-WHILE-refine*:
assumes
(*x*, *x'*) ∈ *R*
⋀*x x'*. (*x*, *x'*) ∈ *R* ⟹ *bm x* ≤ ⇓*Id* (*bm' x'*)
and ⋀*x x' t*. (*x*, *x'*) ∈ *R* ⟹ *nofailT* (*bm' x'*) ⟹ *inresT* (*bm' x'*) *True t* ⟹ *c x* ≤ ⇓*R* (*c' x'*)
shows (*monadic-WHILE bm c x*) ≤ ⇓*R* (*monadic-WHILE bm' c' x'*)
⟨*proof*⟩

lemma *monadic-WHILE-aux*: *monadic-WHILE b f s* = *monadic-WHILEIT* (λ-. *True*) *b f s*
⟨*proof*⟩
lemma *lst* (*c x*) *Q* = *Some t* ⟹ *Some t* ≤ *lst* (*c x*) *Q'*
⟨*proof*⟩

lemma *TbindT-I2*: *tt* ≤ *lst M* (λ*y*. *lst* (*f y*) *Q*) ⟹ *tt* ≤ *lst* (*M* ⤜ *f*) *Q*
⟨*proof*⟩

lemma *T-conseq7*:
assumes
lst f Q' ≥ *tt*
⋀*x t'' M*. *f* = *SPECT M* ⟹ *M x* ≠ *None* ⟹ *Q' x* = *Some t''* ⟹ (*Q x*) ≥ *Some* (*t''*)
shows *lst f Q* ≥ *tt*
⟨*proof*⟩

lemma *monadic-WHILE-ruleaaa''*:
assumes *monadic-WHILE bm c s* = *r*
assumes *IS*[*vcg-rules*]: ⋀*s*.
lst (*bm s*) (λ*b*. *if b then lst* (*c s*) (λ*s'*. *if* (*s'*,*s*)∈*R then I s' else None*) *else Q s*) ≥ *I s*

assumes *wf*: *wf R*
shows *lst r Q* ≥ *I s*
⟨*proof*⟩

lemma *monadic-WHILE-rule''*:
assumes *monadic-WHILE bm c s* = *r*
assumes *IS*[*vcg-rules*]: ⋀*s t'*. *I s* = *Some t'*
⟹ *lst* (*bm s*) (λ*b*. *if b then lst* (*c s*) (λ*s'*. *if* (*s'*,*s*)∈*R then I s' else*

None)*else Q s*) ≥ *Some t'*

assumes *I s* = *Some t*
assumes *wf*: *wf R*
shows *lst r Q* ≥ *Some t*
⟨*proof*⟩

lemma *whileT-rule'''*:
fixes *I* :: *'a* ⇒ *nat option*
assumes *whileT b c s0* = *r*
assumes *progress*: ⋀*s*. *progress* (*c s*)
assumes *IS*[*vcg-rules*]: ⋀*s t t'*. *I s* = *Some t* ⟹ *b s* ⟹
lst (*c s*) (λ*s'*. *mm3 t* (*I s'*)) ≥ *Some 0*

assumes [*simp*]: *I s0* = *Some t0*

shows *lst r* (λ*x*. *if b x then None else mm3 t0* (*I x*)) ≥ *Some 0*
⟨*proof*⟩

fun *Someplus* **where**
Someplus None - = *None*
| *Someplus* - *None* = *None*
| *Someplus* (*Some a*) (*Some b*) = *Some* (*a*+*b*)

lemma *l*: ~ (*a*::*enat*) < *b* ⟷ *a* ≥ *b* ⟨*proof*⟩

lemma *kk*: *a*≥*b* ⟹ (*b*::*enat*) + (*a* −*b*) = *a*
⟨*proof*⟩

lemma *Tea*: *Someplus A B* = *Some t* ⟷ (∃ *a b*. *A* = *Some a* ∧ *B* = *Some b* ∧
t = *a* + *b*)
⟨*proof*⟩

lemma *TTT-Some-nofailT*: *lst c Q* = *Some l* ⟹ *c* ≠ *FAILT*
⟨*proof*⟩

lemma *GRR*: **assumes** *lst* (*SPECT Mf*) *Q* = *Some l*
shows *Mf x* = *None* ∨ (*Q x*≠ *None* ∧ (*Q x*) ≥ *Mf x*)
⟨*proof*⟩

lemma *Someplus-None*: *Someplus A B* = *None* ⟷ (*A* = *None* ∨ *B* = *None*)
⟨*proof*⟩

lemma *Somemm3*: *A* ≥ *B* ⟹ *mm3 A* (*Some B*) = *Some* (*A* − *B*)
⟨*proof*⟩

lemma *neueWhile-rule*: **assumes** *monadic-WHILE bm c s0* = *r*
and *step*: ⋀*s*. *I s* ⟹
Some 0 ≤ *lst* (*bm s*) (λ*b*. *if b*

then lst (c s) (λs'. (if I s' ∧ (E s' ≤ E s) then Some (enat (E s − E s')) else None))
else mm2 (Q s) (Someplus (Some t) (mm3 (E s0) (Some (E s)))))

and *progress*: ⋀*s*. *progress* (*c s*)

and *I0*: *I s0*
shows *Some t* ≤ *lst r Q*
⟨*proof*⟩

definition *monadic-WHILEIE* **where**
monadic-WHILEIE I E bm c s = *monadic-WHILE bm c s*

definition *G b d* = (*if b then Some d else None*)
definition *H Qs t Es0 Es* = *mm2 Qs* (*Someplus* (*Some t*) (*mm3* (*Es0*) (*Some* (*Es*))))

lemma *neueWhile-rule'*:
fixes *s0* :: *'a* **and** *I* :: *'a* ⇒ *bool* **and** *E* :: *'a* ⇒ *nat*
assumes
step: (⋀*s*. *I s* ⟹ *Some 0* ≤ *lst* (*bm s*) (λ*b*. *if b then lst* (*c s*) (λ*s'*. *if I s'* ∧ *E s'* ≤ *E s then Some* (*enat* (*E s* − *E s'*)) *else None*) *else mm2* (*Q s*) (*Someplus* (*Some t*) (*mm3* (*E s0*) (*Some* (*E s*))))))
and *progress*: ⋀*s*. *progress* (*c s*)
and *i*: *I s0*
shows *Some t* ≤ *lst* (*monadic-WHILEIE I E bm c s0*) *Q*
⟨*proof*⟩

lemma *neueWhile-rule''*:
fixes *s0* :: *'a* **and** *I* :: *'a* ⇒ *bool* **and** *E* :: *'a* ⇒ *nat*
assumes
step: (⋀*s*. *I s* ⟹ *Some 0* ≤ *lst* (*bm s*) (λ*b*. *if b then lst* (*c s*) (λ*s'*. *G* (*I s'* ∧ *E s'* ≤ *E s*) (*enat* (*E s* − *E s'*))) *else H* (*Q s*) *t* (*E s0*) (*E s*)))
and *progress*: ⋀*s*. *progress* (*c s*)
and *i*: *I s0*
shows *Some t* ≤ *lst* (*monadic-WHILEIE I E bm c s0*) *Q*
⟨*proof*⟩

lemma *LmonWhileRule*:
fixes *IC CT*
assumes *V*⟨(*''precondition''*, *IC*, []),(*''monwhile''*, *IC*, []) # *CT*: *I s0*⟩
and ⋀*s*. *I s* ⟹ *C*⟨*Suc IC*,(*''invariant''*, *Suc IC*, []) # (*''monwhile''*, *IC*, []) # *CT*,*OC*: *valid 0* (λ*b*. *if b then lst* (*C s*) (λ*s'*. *if I s'* ∧ *E s'* ≤ *E s then Some* (*enat* (*E s* − *E s'*)) *else None*) *else mm2* (*Q s*) (*Someplus* (*Some t*) (*mm3* (*E s0*) (*Some* (*E s*))))) (*bm s*)⟩
and ⋀*s*. *V*⟨(*''progress''*, *IC*, []),(*''monwhile''*, *IC*, []) # *CT*: *progress* (*C s*)⟩
shows *C*⟨*IC*,*CT*,*OC*: *valid t Q* (*monadic-WHILEIE I E bm C s0*)⟩
⟨*proof*⟩

lemma *LWhileRule*:
fixes *IC CT*
assumes $V\langle(''precondition'', IC, []),(''while'', IC, []) \# CT\colon I\ s0\rangle$
and $\bigwedge s.\ I\ s \Longrightarrow\ b\ s \Longrightarrow\ C\langle Suc\ IC,(''invariant'', Suc\ IC, []) \# (''while'', IC, []) \# CT, OC1\colon valid\ 0\ (\lambda s'.\ mm3\ (E\ s)\ (if\ I\ s'\ then\ Some\ (E\ s')\ else\ None))\ (C\ s)\rangle$
and $\bigwedge s.\ V\langle(''progress'', IC, []),(''while'', IC, []) \# CT\colon progress\ (C\ s)\rangle$
and $\bigwedge x.\ \neg\ b\ x \Longrightarrow\ I\ x \Longrightarrow\ (E\ x) \leq (E\ s0) \Longrightarrow C\langle Suc\ OC1,(''postcondition'', IC, [])\#(''while'', IC, []) \# CT, OC\colon Some\ (t + enat\ ((E\ s0) - E\ x)) \leq Q\ x\rangle$
shows $C\langle IC, CT, OC\colon valid\ t\ Q\ (whileIET\ I\ E\ b\ C\ s0)\rangle$
⟨*proof*⟩

lemma *validD*: $valid\ t\ Q\ M \Longrightarrow Some\ t \leq lst\ M\ Q$ ⟨*proof*⟩

lemma *LABELs-to-concl*:
$C\langle IC,\ CT,\ OC\colon True\rangle \Longrightarrow C\langle IC,\ CT,\ OC\colon P\rangle \Longrightarrow P$
$V\langle x,\ ct\colon True\rangle \Longrightarrow V\langle x,\ ct\colon P\rangle \Longrightarrow P$
⟨*proof*⟩

lemma *LASSERTRule*:
assumes $V\langle(''ASSERT'', IC, []), CT\colon \Phi\rangle$
$C\langle Suc\ IC,\ CT, OC\colon valid\ t\ Q\ (RETURNT\ ())\rangle$
shows $C\langle IC, CT, OC\colon valid\ t\ Q\ (ASSERT\ \Phi)\rangle$
⟨*proof*⟩

lemma *LbindTRule*:
assumes $C\langle IC, CT, OC\colon valid\ t\ (\lambda y.\ lst\ (f\ y)\ Q)\ m\rangle$
shows $C\langle IC, CT, OC\colon valid\ t\ Q\ (bindT\ m\ f)\rangle$
⟨*proof*⟩

lemma *LRETURNTRule*:
assumes $C\langle IC, CT, OC\colon Some\ t \leq Q\ x\rangle$
shows $C\langle IC, CT, OC\colon valid\ t\ Q\ (RETURNT\ x)\rangle$
⟨*proof*⟩

lemma *LSELECTRule*:
fixes *IC CT* **defines** $CT' \equiv (''cond'', IC, []) \# CT$
assumes $\bigwedge x.\ P\ x \Longrightarrow C\langle Suc\ IC,(''Some'', IC, []) \# (''SELECT'', IC, []) \# CT, OC1\colon valid\ (t+T)\ Q\ (RETURNT\ (Some\ x))\ \rangle$
and $\forall x.\ \neg\ P\ x \Longrightarrow C\langle Suc\ OC1,(''None'', Suc\ OC1, []) \# (''SELECT'', IC, []) \# CT, OC\colon valid\ (t+T)\ Q\ (RETURNT\ None)\ \rangle$
shows $C\langle IC, CT, OC\colon valid\ t\ Q\ (SELECT\ P\ T)\rangle$
⟨*proof*⟩

lemma *LRESTembRule*:
assumes $\bigwedge x.\ P\ x \Longrightarrow C\langle IC, CT, OC\colon Some\ (t + T\ x) \leq Q\ x\rangle$
shows $C\langle IC, CT, OC\colon valid\ t\ Q\ (REST\ (emb'\ P\ T))\rangle$
⟨*proof*⟩

lemma *LRESTsingleRule*:
assumes $C\langle IC,CT,OC\colon Some\ (t + t') \leq Q\ x\rangle$
shows $C\langle IC,CT,OC\colon valid\ t\ Q\ (REST\ [x\mapsto t'])\rangle$
⟨*proof*⟩

lemma *LTTTinRule*:
assumes $C\langle IC,CT,OC\colon valid\ t\ Q\ M\rangle$
shows $C\langle IC,CT,OC\colon Some\ t \leq lst\ M\ Q\rangle$
⟨*proof*⟩

lemma *LfinaltimeRule*:
assumes $V\langle(''time'',\ IC,\ []),\ CT\colon t \leq t'\ \rangle$
shows $C\langle IC,CT,IC\colon Some\ t \leq Some\ t'\rangle$
⟨*proof*⟩

lemma *LfinalfuncRule*:
assumes $V\langle(''func'',\ IC,\ []),\ CT\colon False\ \rangle$
shows $C\langle IC,CT,IC\colon Some\ t \leq None\rangle$
⟨*proof*⟩

lemma *LembRule*:
assumes $V\langle(''time'',\ IC,\ []),\ CT\colon t \leq T\ x\ \rangle$
and $V\langle(''func'',\ IC,\ []),\ CT\colon P\ x\ \rangle$
shows $C\langle IC,CT,IC\colon Some\ t \leq emb'\ P\ T\ x\rangle$
⟨*proof*⟩

lemma *Lmm3Rule*:
assumes $V\langle(''time'',\ IC,\ []),\ CT\colon Va' \leq Va \wedge t \leq enat\ (Va - Va')\ \rangle$
and $V\langle(''func'',\ IC,\ []),\ CT\colon b\ \rangle$
shows $C\langle IC,CT,IC\colon Some\ t \leq mm3\ Va\ (if\ b\ then\ Some\ Va'\ else\ None)\ \rangle$
⟨*proof*⟩

lemma *LinjectRule*:
assumes $Some\ t \leq lst\ A\ Q \Longrightarrow Some\ t \leq lst\ B\ Q$
$C\langle IC,CT,OC\colon valid\ t\ Q\ A\rangle$
shows $C\langle IC,CT,OC\colon valid\ t\ Q\ B\rangle$
⟨*proof*⟩

lemma *Linject2Rule*:
assumes $A = B$
$C\langle IC,CT,OC\colon valid\ t\ Q\ A\rangle$
shows $C\langle IC,CT,OC\colon valid\ t\ Q\ B\rangle$
⟨*proof*⟩

method *labeled-VCG-init* = ((*rule T-specifies-I validD*)+), *rule Initial-Label*
method *labeled-VCG-step* **uses** *rules* = (*rule rules*[*symmetric*, *THEN Linject2Rule*]

LTTTinRule LbindTRule
LembRule Lmm3Rule
LRETURNTRule LASSERTRule LCondRule LSELECTRule
LRESTsingleRule LRESTembRule
LouterCondRule
LfinaltimeRule LfinalfuncRule
LmonWhileRule LWhileRule) | *vcg-split-case*

method *labeled-VCG* **uses** *rules* = *labeled-VCG-init*, *repeat-all-new* ‹*labeled-VCG-step rules*: *rules*›
method *casified-VCG* **uses** *rules* = *labeled-VCG rules*: *rules*, *casify*

13.7 Examples, labeled vcg

lemma *do* { *x* ← *SELECT P T*;
(*case x of None* ⇒ *RETURNT* (*1*::*nat*) | *Some t* ⇒ *RETURNT* (*2*::*nat*))
} ≤ *SPECT* (*emb Q T'*)
⟨*proof*⟩

lemma
assumes *b c*
shows *do* { *ASSERT b*;
ASSERT c;
RETURNT 1 } ≤ *SPECT* (*emb* (λ*x*. *x*>(*0*::*nat*)) *1*)
⟨*proof*⟩

lemma *do* {
(*if b then RETURNT* (*1*::*nat*) *else RETURNT 2*)
} ≤ *SPECT* (*emb* (λ*x*. *x*>*0*) *1*)
⟨*proof*⟩

lemma
assumes *b*
shows *do* {
ASSERT b;
(*if b then RETURNT* (*1*::*nat*) *else RETURNT 2*)
} ≤ *SPECT* (*emb* (λ*x*. *x*>*0*) *1*)
⟨*proof*⟩

end

lemma *SPECT-ub'*: *T*≤*T'* ⟹ *SPECT* (*emb' M' T*) ≤ ⇓*Id* (*SPECT* (*emb' M' T'*))
⟨*proof*⟩

lemma *REST-single-rule*[*vcg-simp-rules*]: *Some t* ≤ *lst* (*REST* [*x*↦*t'*]) *Q* ⟷ *Some* (*t*+*t'*) ≤ (*Q x*)
⟨*proof*⟩

13.8 progress solver

method *progress* **methods** *solver* =
(*rule asm-rl*[*of progress* -],
(*simp split*: *prod.splits* | *intro allI impI conjI* | *determ* ‹*rule progress-rules*›
| *rule disjI1*; *progress* ‹*solver*›; *fail* | *rule disjI2*; *progress* ‹*solver*›; *fail* | *solver*)+) []
method *progress'* **methods** *solver* =
(*rule asm-rl*[*of progress* -], (*vcg-split-case* | *intro allI impI conjI* | *determ* ‹*rule progress-rules*›
| *rule disjI1 disjI2*; *progress'*‹*solver*› | (*solver*;*fail*))+) []

lemma
assumes (⋀*s t*. *P s* = *Some t* ⟹ ∃ *s'*. *Some t* ≤ *Q s'* ∧ (*s*, *s'*) ∈ *R*)
shows *SPECT-refine*: *SPECT P* ≤ ⇓ *R* (*SPECT Q*)
⟨*proof*⟩

13.9 more stuff involving mm3 and emb

lemma *Some-le-mm3-Some-conv*[*vcg-simp-rules*]: *Some t* ≤ *mm3 t'* (*Some t''*) ⟷ (*t''* ≤ *t'* ∧ *t* ≤ *enat* (*t'* − *t''*))
⟨*proof*⟩

lemma *embtimeI*: *T* ≤ *T'* ⟹ *emb P T* ≤ *emb P T'* ⟨*proof*⟩

lemma *not-cons-is-Nil-conv*[*simp*]: (∀ *y ys*. *l* ≠ *y* # *ys*) ⟷ *l*=[] ⟨*proof*⟩

lemma *mm3-Some0-eq*[*simp*]: *mm3 t* (*Some 0*) = *Some t*
⟨*proof*⟩

lemma *ran-emb'*: *c* ∈ *ran* (*emb' Q t*) ⟷ (∃ *s'*. *Q s'* ∧ *t s'* = *c*)
⟨*proof*⟩

lemma *ran-emb-conv*: *Ex Q* ⟹ *ran* (*emb Q t*) = {*t*}
⟨*proof*⟩

lemma *in-ran-emb-special-case*: *c*∈*ran* (*emb Q t*) ⟹ *c*≤*t*
⟨*proof*⟩

lemma *dom-emb'-eq*[*simp*]: *dom* (*emb' Q f*) = *Collect Q*
⟨*proof*⟩

lemma *emb-le-emb*: *emb' P T* ≤ *emb' P' T'* ⟷ (∀ *x*. *P x* ⟶ *P' x* ∧ *T x* ≤ *T' x*)

⟨*proof*⟩

13.10 VCG for monadic programs

13.10.1 old

declare *mm3-Some-conv* [*vcg-simp-rules*]

lemma *SS*[*vcg-simp-rules*]: *Some t* = *Some t'* ⟷ *t* = *t'* ⟨*proof*⟩
lemma *SS'*: (*if b then Some t else None*) = *Some t'* ⟷ (*b* ∧ *t* = *t'*) ⟨*proof*⟩

term (*case s of* (*a*,*b*) ⇒ *M a b*)
lemma *case-T*[*vcg-rules*]: (⋀*a b*. *s* = (*a*, *b*) ⟹ *t* ≤ *lst Q* (*M a b*)) ⟹ *t* ≤ *lst Q* (*case s of* (*a*,*b*) ⇒ *M a b*)
⟨*proof*⟩

13.10.2 new setup

named-theorems *vcg-rules'*
lemma *if-T*[*vcg-rules'*]: (*b* ⟹ *t* ≤ *lst Ma Q*) ⟹ (¬*b* ⟹ *t* ≤ *lst Mb Q*) ⟹ *t* ≤ *lst* (*if b then Ma else Mb*) *Q*
⟨*proof*⟩
lemma *RETURNT-T-I*[*vcg-rules'*]: *t* ≤ *Q x* ⟹ *t* ≤ *lst* (*RETURNT x*) *Q*
⟨*proof*⟩

declare *T-SPECT-I* [*vcg-rules'*]
declare *TbindT-I* [*vcg-rules'*]
declare *T-RESTemb* [*vcg-rules'*]
declare *T-ASSERT-I* [*vcg-rules'*]
declare *While*[*vcg-rules'*]

named-theorems *vcg-simps'*

declare *option.case* [*vcg-simps'*]

declare *neueWhile-rule''* [*vcg-rules'*]

method *vcg'-step* **methods** *solver* **uses** *rules* =
(*intro rules vcg-rules'* | *vcg-split-case* | (*progress simp*;*fail*) | (*solver*; *fail*))

method *vcg'* **methods** *solver* **uses** *rules* = *repeat-all-new* ‹*vcg'-step solver rules*: *rules*›

declare *T-SELECT* [*vcg-rules'*]

— No proof
lemma ⋀*c*. *do* { *c* ← *RETURNT None*;
(*case-option* (*RETURNT* (*1*::*nat*)) (λ*p*. *RETURNT* (*2*::*nat*))) *c*
} ≤ *SPECT* (*emb* (λ*x*. *x*>(*0*::*nat*)) *1*)
⟨*proof*⟩

thm *option.case*

13.11 setup for *refine-vcg*

lemma *If-refine*[*refine*]: $b = b' \Longrightarrow$
$(b \Longrightarrow b' \Longrightarrow S1 \leq \Downarrow R\ S1') \Longrightarrow$
$(\neg\ b \Longrightarrow \neg\ b' \Longrightarrow S2 \leq \Downarrow R\ S2') \Longrightarrow (\mathit{if}\ b\ \mathit{then}\ S1\ \mathit{else}\ S2) \leq \Downarrow R\ (\mathit{if}\ b'\ \mathit{then}\ S1'\ \mathit{else}\ S2')$
⟨*proof*⟩

lemma *Case-option-refine*[*refine*]: $(x,x') \in \langle S \rangle \mathit{option\text{-}rel} \Longrightarrow$
$(\bigwedge y\ y'.\ (y,y') \in S \Longrightarrow S2\ y\ \leq \Downarrow R\ (S2'\ y')) \Longrightarrow S1 \leq \Downarrow R\ S1'$
$\Longrightarrow (\mathit{case}\ x\ \mathit{of}\ \mathit{None} \Rightarrow S1 \mid \mathit{Some}\ y \Rightarrow S2\ y) \leq \Downarrow R\ (\mathit{case}\ x'\ \mathit{of}\ \mathit{None} \Rightarrow S1' \mid \mathit{Some}\ y' \Rightarrow S2'\ y')$
⟨*proof*⟩
lemma *conc-fun-Id-refined*[*refine0*]: $\bigwedge S.\ S \leq \Downarrow \mathit{Id}\ S$ ⟨*proof*⟩
lemma *conc-fun-ASSERT-refine*[*refine0*]: $\Phi \Longrightarrow (\Phi \Longrightarrow S \leq \Downarrow R\ S') \Longrightarrow \mathit{ASSERT}\ \Phi \gg\!= (\lambda\text{-}.\ S) \leq \Downarrow R\ S'$
⟨*proof*⟩
declare *le-R-ASSERTI* [*refine0*]

declare *bindT-refine* [*refine*]
declare *WHILET-refine* [*refine*]

— Check name
lemma *SPECT-refine-vcg-cons*[*refine-vcg-cons*]: $m \leq \mathit{SPECT}\ \Phi \Longrightarrow (\bigwedge x.\ \Phi\ x \leq \Psi\ x) \Longrightarrow m \leq \mathit{SPECT}\ \Psi$
⟨*proof*⟩

end
theory *SepLogic-Misc*
imports *DataRefinement*
begin

13.12 Relators

definition *nrest-rel* **where**
nres-rel-def-internal: $\mathit{nrest\text{-}rel}\ R \equiv \{(c,a).\ c \leq \Downarrow R\ a\}$

lemma *nrest-rel-def*: $\langle R \rangle \mathit{nrest\text{-}rel} \equiv \{(c,a).\ c \leq \Downarrow R\ a\}$
⟨*proof*⟩

lemma *nrest-relD*: $(c,a) \in \langle R \rangle \mathit{nrest\text{-}rel} \Longrightarrow c \leq \Downarrow R\ a$ ⟨*proof*⟩
lemma *nrest-relI*: $c \leq \Downarrow R\ a \Longrightarrow (c,a) \in \langle R \rangle \mathit{nrest\text{-}rel}$ ⟨*proof*⟩

lemma *nrest-rel-comp*: $\langle A \rangle \mathit{nrest\text{-}rel}\ O\ \langle B \rangle \mathit{nrest\text{-}rel} = \langle A\ O\ B \rangle \mathit{nrest\text{-}rel}$
⟨*proof*⟩

lemma *pw-nrest-rel-iff*: $(a,b)\in\langle A\rangle nrest\text{-}rel \longleftrightarrow nofailT\ (\Downarrow\ A\ b) \longrightarrow nofailT\ a \wedge (\forall x\ t.\ inresT\ a\ x\ t \longrightarrow inresT\ (\Downarrow\ A\ b)\ x\ t)$
⟨*proof*⟩

lemma *param-FAIL*[*param*]: $(FAILT,FAILT) \in \langle R\rangle nrest\text{-}rel$
⟨*proof*⟩

lemma *param-RETURN*[*param*]:
$(RETURNT,RETURNT) \in R \to \langle R\rangle nrest\text{-}rel$
⟨*proof*⟩

lemma *param-bind*[*param*]:
$(bindT,bindT) \in \langle Ra\rangle nrest\text{-}rel \to (Ra\to\langle Rb\rangle nrest\text{-}rel) \to \langle Rb\rangle nrest\text{-}rel$
⟨*proof*⟩

end
theory *Refine-Foreach*
imports *NREST RefineMonadicVCG SepLogic-Misc*
begin

A common pattern for loop usage is iteration over the elements of a set. This theory provides the *FOREACH*-combinator, that iterates over each element of a set.

13.13 Auxilliary Lemmas

The following lemma is commonly used when reasoning about iterator invariants. It helps converting the set of elements that remain to be iterated over to the set of elements already iterated over.

lemma *it-step-insert-iff*:
$it \subseteq S \Longrightarrow x\in it \Longrightarrow S-(it-\{x\}) = insert\ x\ (S-it)$ ⟨*proof*⟩

13.14 Definition

Foreach-loops come in different versions, depending on whether they have an annotated invariant (I), a termination condition (C), and an order (O).

Note that asserting that the set is finite is not necessary to guarantee termination. However, we currently provide only iteration over finite sets, as this also matches the ICF concept of iterators.

definition *FOREACH-body* $f \equiv \lambda(xs,\ \sigma).\ do\ \{$
$x \leftarrow RETURNT(\ hd\ xs);\ \sigma'\leftarrow f\ x\ \sigma;\ RETURNT\ (tl\ xs,\sigma')$
$\}$

definition *FOREACH-cond* **where** *FOREACH-cond* $c \equiv (\lambda(xs,\sigma).\ xs\neq[]\ \wedge\ c\ \sigma)$

Foreach with continuation condition, order and annotated invariant:

definition *FOREACHoci* (‹*FOREACH*$_{OC}$ ⁻ ⁻›) **where** *FOREACHoci R Φ S c f σ0 inittime body-time* ≡ *do* {
ASSERT (*finite S*);
xs ← *SPECT* (*emb* (λ*xs*. *distinct xs* ∧ *S* = *set xs* ∧ *sorted-wrt R xs*) *inittime*);
(-,*σ*) ← *whileIET*
(λ(*it*,*σ*). ∃ *xs'*. *xs* = *xs'* @ *it* ∧ Φ (*set it*) *σ*) (λ(*it*,-). *length it* ∗ *body-time*) (*FOREACH-cond c*) (*FOREACH-body f*) (*xs*,*σ0*);
RETURNT σ }

Foreach with continuation condition and annotated invariant:

definition *FOREACHci* (‹*FOREACH*$_{C}$ ⁻›) **where** *FOREACHci* ≡ *FOREACHoci* (λ- -. *True*)

13.15 Proof Rules

lemma *FOREACHoci-rule*:
assumes *IP*:
⋀*x it σ*. ⟦ *c σ*; *x*∈*it*; *it*⊆*S*; *I it σ*; ∀ *y*∈*it* − {*x*}. *R x y*;
∀ *y*∈*S* − *it*. *R y x* ⟧ ⟹ *f x σ* ≤ *SPECT* (*emb* (*I* (*it*−{*x*})) (*enat body-time*))
assumes *FIN*: *finite S*
assumes *I0*: *I S σ0*
assumes *II1*: ⋀*σ*. ⟦*I* {} *σ*⟧ ⟹ *P σ*
assumes *II2*: ⋀*it σ*. ⟦ *it*≠{}; *it*⊆*S*; *I it σ*; ¬*c σ*;
∀ *x*∈*it*. ∀ *y*∈*S* − *it*. *R y x* ⟧ ⟹ *P σ*
assumes *time-ub*: *inittime* + *enat* ((*card S*) ∗ *body-time*) ≤ *enat overall-time*
assumes *progress-f*[*progress-rules*]: ⋀*a b*. *progress* (*f a b*)
shows *FOREACHoci R I S c f σ0 inittime body-time* ≤ *SPECT* (*emb P* (*enat overall-time*))
⟨*proof*⟩

lemma *FOREACHci-rule* :
assumes *IP*:
⋀*x it σ*. ⟦ *x*∈*it*; *it*⊆*S*; *I it σ*; *c σ* ⟧ ⟹ *f x σ* ≤ *SPECT* (*emb* (*I* (*it*−{*x*})) (*enat body-time*))
assumes *II1*: ⋀*σ*. ⟦*I* {} *σ*⟧ ⟹ *P σ*
assumes *II2*: ⋀*it σ*. ⟦ *it*≠{}; *it*⊆*S*; *I it σ*; ¬*c σ* ⟧ ⟹ *P σ*
assumes *FIN*: *finite S*
assumes *I0*: *I S σ0*
assumes *progress-f*: ⋀*a b*. *progress* (*f a b*)
assumes *inittime* + *enat* (*card S* ∗ *body-time*) ≤ *enat overall-time*
shows *FOREACHci I S c f σ0 inittime body-time* ≤ *SPECT* (*emb P* (*enat overall-time*))
⟨*proof*⟩

We define a relation between distinct lists and sets.

definition [*to-relAPP*]: *list-set-rel R* ≡ ⟨*R*⟩*list-rel O br set distinct*

13.16 Nres-Fold with Interruption (nfoldli)

A foreach-loop can be conveniently expressed as an operation that converts the set to a list, followed by folding over the list.

This representation is handy for automatic refinement, as the complex foreach-operation is expressed by two relatively simple operations.

We first define a fold-function in the nrest-monad

definition *nfoldli* **where**
nfoldli l c f s = RECT (λ*D* (*l*,*s*). (*case l of*
[] ⇒ *RETURNT s*
| *x*#*ls* ⇒ *if c s then do* { *s*←*f x s*; *D* (*ls*, *s*)} *else RETURNT s*
)) (*l*, *s*)

lemma *nfoldli-simps*[*simp*]:
nfoldli [] *c f s = RETURNT s*
nfoldli (*x*#*ls*) *c f s* =
(*if c s then do* { *s*←*f x s*; *nfoldli ls c f s*} *else RETURNT s*)
⟨*proof*⟩

lemma *param-nfoldli*[*param*]:
shows (*nfoldli*,*nfoldli*) ∈
⟨*Ra*⟩*list-rel* → (*Rb*→*Id*) → (*Ra*→*Rb*→⟨*Rb*⟩*nrest-rel*) → *Rb* → ⟨*Rb*⟩*nrest-rel*
⟨*proof*⟩

lemma *nfoldli-no-ctd*[*simp*]: ¬*ctd s* ⟹ *nfoldli l ctd f s = RETURNT s*
⟨*proof*⟩

lemma *nfoldli-append*: *nfoldli* (*l1*@*l2*) *ctd f s = nfoldli l1 ctd f s* ≫= *nfoldli l2 ctd f*
⟨*proof*⟩

lemma *nfoldli-assert*:
assumes *set l* ⊆ *S*
shows *nfoldli l c* (λ *x s*. *ASSERT* (*x* ∈ *S*) ≫ *f x s*) *s = nfoldli l c f s*
⟨*proof*⟩

lemmas *nfoldli-assert′ = nfoldli-assert*[*OF order.refl*]

definition *nfoldliIE* :: (′*d list* ⇒ ′*d list* ⇒ ′*a* ⇒ *bool*) ⇒ *nat* ⇒ ′*d list* ⇒ (′*a* ⇒ *bool*) ⇒ (′*d* ⇒ ′*a* ⇒ ′*a nrest*) ⇒ ′*a* ⇒ ′*a nrest* **where**
nfoldliIE I E l c f s = nfoldli l c f s

lemma *nfoldliIE-rule′*:
assumes *IS*: ⋀*x l1 l2 σ*. ⟦ *l0*=*l1*@*x*#*l2*; *I l1* (*x*#*l2*) *σ*; *c σ* ⟧ ⟹ *f x σ* ≤ *SPECT* (*emb* (*I* (*l1*@[*x*]) *l2*) (*enat body-time*))
assumes *FNC*: ⋀*l1 l2 σ*. ⟦ *l0*=*l1*@*l2*; *I l1 l2 σ*; ¬*c σ* ⟧ ⟹ *P σ*
assumes *FC*: ⋀*σ*. ⟦ *I l0* [] *σ* ⟧ ⟹ *P σ*

shows $lr@l = l0 \Longrightarrow I\ lr\ l\ \sigma \Longrightarrow nfoldliIE\ I\ body\text{-}time\ l\ c\ f\ \sigma \leq SPECT\ (emb\ P\ (body\text{-}time * length\ l))$
⟨*proof*⟩

lemma *nfoldliIE-rule*:
assumes *I0*: $I\ []\ l0\ \sigma 0$
assumes *IS*: $\bigwedge x\ l1\ l2\ \sigma.\ [\![\ l0{=}l1@x\#l2;\ I\ l1\ (x\#l2)\ \sigma;\ c\ \sigma\ ]\!] \Longrightarrow f\ x\ \sigma \leq SPECT\ (emb\ (I\ (l1@[x])\ l2)\ (enat\ body\text{-}time))$
assumes *FNC*: $\bigwedge l1\ l2\ \sigma.\ [\![\ l0{=}l1@l2;\ I\ l1\ l2\ \sigma;\ \neg c\ \sigma\ ]\!] \Longrightarrow P\ \sigma$
assumes *FC*: $\bigwedge \sigma.\ [\![\ I\ l0\ []\ \sigma\ ]\!] \Longrightarrow P\ \sigma$
assumes *T*: $(body\text{-}time * length\ l0) \leq t$
shows *nfoldliIE I body-time l0 c f* $\sigma 0 \leq SPECT\ (emb\ P\ t)$
⟨*proof*⟩

lemma *nfoldli-refine*[*refine*]:
assumes $(li,\ l) \in \langle S\rangle list\text{-}rel$
and $(si,\ s) \in R$
and *CR*: $(ci,\ c) \in R \to bool\text{-}rel$
and [*refine*]: $\bigwedge xi\ x\ si\ s.\ [\![\ (xi,x)\in S;\ (si,s)\in R;\ c\ s\ ]\!] \Longrightarrow fi\ xi\ si \leq \Downarrow R\ (f\ x\ s)$
shows *nfoldli li ci fi si* $\leq \Downarrow\ R\ (nfoldli\ l\ c\ f\ s)$
⟨*proof*⟩

definition *nfoldliIE′ I bt l0 f s0* = *nfoldliIE I bt l0* (λ*-. True*) *f s0*

lemma *nfoldliIE′-rule*:
assumes
$\bigwedge x\ l1\ l2\ \sigma.$
$l0 = l1\ @\ x\ \#\ l2 \Longrightarrow$
$I\ l1\ (x\ \#\ l2)\ \sigma \Longrightarrow Some\ 0 \leq lst\ (f\ x\ \sigma)\ (emb\ (I\ (l1\ @\ [x])\ l2)\ (enat\ body\text{-}time))$
$I\ []\ l0\ \sigma 0$
$(\bigwedge \sigma.\ I\ l0\ []\ \sigma \Longrightarrow Some\ (t + enat\ (body\text{-}time * length\ l0)) \leq Q\ \sigma)$
shows *Some t* $\leq lst\ (nfoldliIE′\ I\ body\text{-}time\ l0\ f\ \sigma 0)\ Q$
⟨*proof*⟩

We relate our fold-function to the while-loop that we used in the original definition of the foreach-loop

lemma *nfoldli-while*: *nfoldli l c f* σ
$\leq$
(*whileIET I E*
(*FOREACH-cond c*) (*FOREACH-body f*) $(l,\ \sigma)$ $\gg\!=$
(λ(*-*, σ). *RETURNT* σ))
⟨*proof*⟩

lemma *rr*: $l0 = l1\ @\ a \Longrightarrow a \neq [] \Longrightarrow l0 = l1\ @\ hd\ a\ \#\ tl\ a$ ⟨*proof*⟩

lemma *nfoldli-rule*:
assumes *I0*: $I\ []\ l0\ \sigma 0$
assumes *IS*: $\bigwedge x\ l1\ l2\ \sigma.\ [\![\ l0{=}l1@x\#l2;\ I\ l1\ (x\#l2)\ \sigma;\ c\ \sigma\ ]\!] \Longrightarrow f\ x\ \sigma \leq SPECT$

(*emb* (*I* (*l1*@[*x*]) *l2*) (*enat body-time*))
assumes *FNC*: ⋀*l1 l2 σ*. ⟦ *l0*=*l1*@*l2*; *I l1 l2 σ*; ¬*c σ* ⟧ ⟹ *P σ*
assumes *FC*: ⋀*σ*. ⟦ *I l0* [] *σ*; *c σ* ⟧ ⟹ *P σ*
assumes *progressf*: ⋀*x y*. *progress* (*f x y*)
shows *nfoldli l0 c f σ0* ≤ *SPECT* (*emb P* (*body-time* ∗ *length l0*))
⟨*proof*⟩

definition *LIST-FOREACH′ tsl c f σ* ≡ *do* {*xs* ← *tsl*; *nfoldli xs c f σ*}

This constant is a placeholder to be converted to custom operations by pattern rules

definition *it-to-sorted-list R s to-sorted-list-time*
≡ *SPECT* (*emb* (λ*l*. *distinct l* ∧ *s* = *set l* ∧ *sorted-wrt R l*) *to-sorted-list-time*)

definition *LIST-FOREACH* Φ *tsl c f σ0 bodytime* ≡ *do* {
xs ← *tsl*;
(-,*σ*) ← *whileIET* (λ(*it*, *σ*). ∃ *xs′*. *xs* = *xs′* @ *it* ∧ Φ (*set it*) *σ*) (λ(*it*, *σ*). *length it* ∗ *bodytime*)
(*FOREACH-cond c*) (*FOREACH-body f*) (*xs*, *σ0*);
RETURNT σ}

lemma *FOREACHoci-by-LIST-FOREACH*:
FOREACHoci R Φ *S c f σ0 to-sorted-list-time bodytime* = *do* {
ASSERT (*finite S*);
LIST-FOREACH Φ (*it-to-sorted-list R S to-sorted-list-time*) *c f σ0 bodytime*
}
⟨*proof*⟩

end

References

[1] M. P. L. Haslbeck and P. Lammich. For a few dollars more. In N. Yoshida, editor, *Programming Languages and Systems*, pages 292–319, Cham, 2021. Springer International Publishing.

[2] M. P. L. Haslbeck and P. Lammich. For a few dollars more: Verified fine-grained algorithm analysis down to llvm. *ACM Trans. Program. Lang. Syst.*, 44(3), jul 2022.