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 **8**

4 Monad Operators **12**

5 Monad Rules **15**

6 Monotonicity lemmas **16**
- 6.1 Derived Program Constructs . . . 16
  - 6.1.1 Assertion . . . 16
- 6.2 SELECT . . . 17

7 RECT **18**

8 Generalized Weakest Precondition **20**
- 8.1 mm . . . 20
- 8.2 mii . . . 23
- 8.3 lst - latest starting time . . . 24
- 8.4 pointwise reasoning about lst via nres3 . . . 25
- 8.5 rules for lst . . . 28

9 consequence rules **28**

10 Experimental Hoare reasoning **31**

11 VCG **32**
11.1 Progress rules 32

12 rules for whileT **34**

13 some Monadic Refinement Automation **36**
13.1 Data Refinement 38
13.1.1 Examples 38
13.2 WHILET refine 44
13.3 ASSERT 46
13.4 VCG splitter 47
13.5 mm3 and emb 48
13.6 Setup Labeled VCG 49
13.7 Examples, labeled vcg 59
13.8 progress solver 61
13.9 more stuff involving mm3 and emb 61
13.10VCG for monadic programs 62
13.10.1 old 62
13.10.2 new setup 62
13.11setup for *refine-vcg* 63
13.12Relators 64
13.13Auxilliary Lemmas 65
13.14Definition 65
13.15Proof Rules 65
13.16Nres-Fold with Interruption (nfoldli) 67

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'*: $x \leq None \longleftrightarrow x = None$
by(*auto simp*: *less-eq-option-None-is-None*)

lemma *everywhereNone*: $(\forall x{\in}X.\ x = None) \longleftrightarrow X = \{\} \vee X = \{None\}$
by *auto*

1.2 Auxiliaries for enat

lemma *enat-minus-mono*: $a' \geq b \Longrightarrow a' \geq a \Longrightarrow a' - b \geq (a::enat) - b$
by (*cases a*; *cases b*; *cases a'*) *auto*

lemma *enat-plus-minus-aux1*: $a + b \leq a' \Longrightarrow \neg\ a' < a \Longrightarrow b \leq a' - (a::enat)$
by (*cases a*; *cases b*; *cases a'*) *auto*

lemma *enat-plus-minus-aux2*: $\neg\ a < a' \Longrightarrow b \leq a - a' \Longrightarrow a' + b \leq (a::enat)$
by (*cases a*; *cases b*; *cases a'*) *auto*

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

lemma *helper*: $x2 \leq x2a \Longrightarrow \neg\ x2 < a \Longrightarrow \neg\ x2a < a \Longrightarrow\ x2 - (a::enat) \leq x2a - a$
by (*cases x2*; *cases x2a*; *cases a*) *auto*

lemma *helper2*: $x2b \leq x2 \Longrightarrow \neg\ x2a < x2\ \Longrightarrow \neg\ x2a < x2b \Longrightarrow x2a - (x2::enat) \leq x2a - x2b$
by (*cases x2*; *cases x2a*; *cases x2b*) *auto*

lemma *Sup-finite-enat*: $Sup\ X = Some\ (enat\ a) \Longrightarrow Some\ (enat\ a) \in X$
by (*auto simp*: *Sup-option-def Sup-enat-def these-empty-eq Max-eq-iff in-these-eq split*: *if-splits*)

lemma *Sup-enat-less2*: $Sup\ X = \infty \Longrightarrow \exists\, x{\in}X.\ enat\ t < x$
by (*subst less-Sup-iff*[*symmetric*]) *simp*

lemma *enat-upper*[*simp*]: $t \leq Some\ (\infty::enat)$
by (*cases t*, *auto*)

1.3 Auxiliary (for Sup and Inf)

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

lemma *aux2*: $(\lambda f.\ f\ x)\ `\ \{[x \mapsto t1]\ |x\ t1.\ M\ x = Some\ t1\} = \{None\} \longleftrightarrow (M\ x = None \wedge M{\neq}Map.empty)$
by (*cases M x*; *force simp*: *aux11*)

lemma *aux3*: $(\lambda f.\ f\ x)\ `\ \{[x \mapsto t1]\ |x\ t1.\ M\ x = Some\ t1\}$
$= \{Some\ t1\ |\ t1.\ M\ x = Some\ t1\} \cup (\{None\ |\ y.\ y{\neq}x \wedge M\ y \neq None\ \})$
by (*fastforce split*: *if-splits simp*: *image-iff*)

lemma *Sup-pointwise-eq-fun*: $(\bigsqcup f \in \{[x \mapsto t1]\ |x\ t1.\ M\ x = Some\ t1\}.\ f\ x) = M\ x$
proof(*cases M x*)
case *None*

then show *?thesis*
by (*auto simp add*: *Sup-option-def aux2 split*: *if-splits*)
next
case (*Some a*)
have *s*: *Option.these* {*u*. *u* = *None* ∧ (∃ *y*. *y* ≠ *x* ∧ (∃ *ya*. *M y* = *Some ya*))} =
{}
by (*auto simp add*: *in-these-eq*)
from *Some* **show** *?thesis*
by (*auto simp add*: *Sup-option-def s aux3 split*: *if-splits*)
qed

lemma *SUP-eq-None-iff*: (⨆*f* ∈ *X*. *f x*) = *None* ⟷ *X*={} ∨ (∀*f*∈*X*. *f x* =
None)
by (*smt* (*z3*) *SUP-bot-conv*(*2*) *SUP-empty Sup-empty empty-Sup*)

lemma *SUP-eq-Some-iff*:
(⨆*f* ∈ *X*. *f x*) = *Some t* ⟷ (∃*f*∈*X*. *f x* ≠ *None*) ∧ (*t*=*Sup* {*t′* | *f t′*. *f*∈*X* ∧ *f*
x = *Some t′* })
proof *safe*
show (⨆*f*∈*X*. *f x*) = *Some t* ⟹ ∃*f*∈*X*. *f x* ≠ *None*
by (*metis SUP-eq-None-iff option.distinct*(*1*))
qed (*fastforce intro*!: *arg-cong*[**where** *f*=*Sup*] *simp*: *image-iff Option.these-def Sup-option-def*
split: *option.splits if-splits*)+

lemma *Sup-enat-less*: *X* ≠ {} ⟹ *enat t* ≤ *Sup X* ⟷ (∃ *x*∈*X*. *enat t* ≤ *x*)
using *finite-enat-bounded* **by** (*simp add*: *Max-ge-iff Sup-upper2 Sup-enat-def*)
(*meson nle-le*)

lemma fixes *Q P* **shows**
Inf { *P x* ≤ *Q x* |*x*. *True*} ⟷ *P* ≤ *Q*
unfolding *le-fun-def* **by** *simp*

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**
  - $\leq$ *FAILi* $\longleftrightarrow$ *True* |
  (*REST a*) $\leq$ (*REST b*) $\longleftrightarrow$ *a* $\leq$ *b* |
  FAILi $\leq$ (*REST* -) $\longleftrightarrow$ *False*

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

fun *sup-nrest* **where**
  sup - *FAILi* = *FAILi* |
  sup FAILi - = *FAILi* |
  sup (*REST a*) (*REST b*) = *REST* (λ*x*. *max* (*a x*) (*b x*))

fun *inf-nrest* **where**
  inf x FAILi = *x* |
  inf FAILi x = *x* |
  inf (*REST a*) (*REST b*) = *REST* (λ*x*. *min* (*a x*) (*b x*))

lemma *min* (*None*) (*Some* (*1*::*enat*)) = *None* **by** *simp*
lemma *max* (*None*) (*Some* (*1*::*enat*)) = *Some 1* **by** *eval*

definition *Sup X* $\equiv$ *if FAILi*$\in$*X then FAILi else REST* (*Sup* {*f* . *REST f* $\in$ *X*})
definition *Inf X* $\equiv$ *if* $\exists$*f*. *REST f*$\in$*X then REST* (*Inf* {*f* . *REST f* $\in$ *X*}) *else FAILi*

definition *bot* $\equiv$ *REST* (*Map.empty*)
definition *top* $\equiv$ *FAILi*

instance
proof(*intro-classes*, *goal-cases*)
  case (*1 x y*)
  then show *?case*
    by (*cases x*; *cases y*) *auto*
next
  case (*2 x*)
  then show *?case*
    by (*cases x*) *auto*
next
  case (*3 x y z*)
  then show *?case*
    by (*cases x*; *cases y*; *cases z*) *auto*
next
  case (*4 x y*)
  then show *?case*
    by (*cases x*; *cases y*) *auto*
next

```
  case (5 x y)
  then show ?case
    by (cases x; cases y) (auto intro: le-funI)
next
  case (6 x y)
  then show ?case
    by (cases x; cases y) (auto intro: le-funI)
next
  case (7 x y z)
  then show ?case
    by (cases x; cases y; cases z) (auto intro!: le-funI dest: le-funD)
next
  case (8 x y)
  then show ?case
    by (cases x; cases y) (auto intro: le-funI)
next
  case (9 y x)
  then show ?case
    by (cases x; cases y) (auto intro: le-funI)
next
  case (10 y x z)
  then show ?case
    by (cases x; cases y; cases z) (auto intro!: le-funI dest: le-funD)
next
  case (11 x A)
  then show ?case
    by (cases x) (auto simp add: Inf-nrest-def Inf-lower)
next
  case (12 A z)
  then show ?case
    by (cases z) (fastforce simp add: Inf-nrest-def le-Inf-iff)+
next
  case (13 x A)
  then show ?case
    by (cases x) (auto simp add: Sup-nrest-def Sup-upper)
next
  case (14 A z)
  then show ?case
    by (cases z) (fastforce simp add: Sup-nrest-def Sup-le-iff)+
next
  case 15
  then show ?case
    by (auto simp add: Inf-nrest-def top-nrest-def)
next
  case 16
  then show ?case
    by (auto simp add: Sup-nrest-def bot-nrest-def bot-option-def)
qed
end
```

definition *RETURNT x* ≡ *REST* (λ*e*. *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*]
unfolding *RETURNT-def* **by** *auto*

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*
unfolding *top-nrest-def bot-nrest-def RETURNT-def*
by *simp-all* (*metis option.distinct*(*1*))+

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*
unfolding *top-nrest-def bot-nrest-def RETURNT-def*
by (*auto simp add*: *fun-eq-iff*)

lemma *nres-simp-internals*:
REST Map.empty = *SUCCEEDT*
FAILi = *FAILT*
unfolding *top-nrest-def bot-nrest-def* **by** *simp-all*

lemma *nres-order-simps*[*simp*]:
¬ *FAILT* ≤ *REST M*
REST M ≤ *REST M'* ⟷ (*M*≤*M'*)
by (*auto simp*: *nres-simp-internals*[*symmetric*])

lemma *nres-top-unique*[*simp*]: *FAILT* ≤ *S'* ⟷ *S'* = *FAILT*
by (*rule top-unique*)

lemma *FAILT-cases*[*simp*]: (*case FAILT of FAILi* ⇒ *P* | *REST x* ⇒ *Q x*) = *P*
by (*auto simp*: *nres-simp-internals*[*symmetric*])

lemma *nrest-Sup-FAILT*:

Sup X = FAILT $\longleftrightarrow$ *FAILT* $\in$ *X*
FAILT = Sup X $\longleftrightarrow$ *FAILT* $\in$ *X*
by (*auto simp*: *nres-simp-internals Sup-nrest-def*)

lemma *nrest-Sup-SPECT-D*: *Sup X = SPECT m* $\Longrightarrow$ *m x = Sup* {*f x* | *f*. *REST f* $\in$ *X*}
unfolding *Sup-nrest-def* **by**(*auto split*: *if-splits intro*!: *arg-cong*[**where** *f*=*Sup*])

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

lemma *nrest-noREST-FAILT*[*simp*]: ($\forall$ *x2*. *m* $\neq$ *REST x2*) $\longleftrightarrow$ *m*=*FAILT*
by (*cases m*) *auto*

lemma *no-FAILTE*:
assumes *g xa* $\neq$ *FAILT*
obtains *X* **where** *g xa = REST X*
using *assms* **by** (*cases g xa*) *auto*

lemma *case-prod-refine*:
fixes *P Q* :: *'a* $\Rightarrow$ *'b* $\Rightarrow$ *'c nrest*
assumes $\bigwedge$*a b*. *P a b* $\leq$ *Q a b*
shows (*case x of* (*a*,*b*) $\Rightarrow$ *P a b*) $\leq$ (*case x of* (*a*,*b*) $\Rightarrow$ *Q a b*)
using *assms* **by** (*simp add*: *split-def*)

lemma *case-option-refine*:
fixes *P Q* :: *'a* $\Rightarrow$ *'b* $\Rightarrow$ *'c nrest*
assumes
PN $\leq$ *QN*
$\bigwedge$*a*. *PS a* $\leq$ *QS a*
shows (*case x of None* $\Rightarrow$ *PN* | *Some a* $\Rightarrow$ *PS a*) $\leq$ (*case x of None* $\Rightarrow$ *QN* | *Some a* $\Rightarrow$ *QS a*)
using *assms* **by** (*auto split*: *option.splits*)

lemma *SPECT-Map-empty*[*simp*]: *SPECT Map.empty* $\leq$ *a*
by (*cases a*) (*auto simp*: *le-fun-def*)

lemma *FAILT-SUP*: (*FAILT* $\in$ *X*) $\Longrightarrow$ *Sup X = FAILT*
by (*simp add*: *nrest-Sup-FAILT*)

3 pointwise reasoning

named-theorems *refine-pw-simps*
ML ‹
structure refine-pw-simps = Named-Thms
(*val name = @{binding refine-pw-simps}*
val description = Refinement Framework: ^
Simplifier rules for pointwise reasoning)
›

definition *nofailT* :: *'a nrest ⇒ bool* **where** *nofailT S ≡ S≠FAILT*

definition *le-or-fail* :: *'a nrest ⇒ 'a nrest ⇒ bool* (**infix** ‹≤$_n$› *50*) **where**
m ≤$_n$ m' ≡ nofailT m ⟶ m ≤ m'

lemma *nofailT-simps*[*simp*]:
nofailT FAILT ⟷ False
nofailT (REST X) ⟷ True
nofailT (RETURNT x) ⟷ True
nofailT SUCCEEDT ⟷ True
unfolding *nofailT-def*
by (*simp-all add*: *RETURNT-def*)

definition *inresT* :: *'a nrest ⇒ 'a ⇒ nat ⇒ bool* **where**
inresT S x t ≡ (case S of FAILi ⇒ True | REST X ⇒ (∃ t'. X x = Some t' ∧ enat t≤t'))

lemma *inresT-alt*: *inresT S x t ⟷ REST ([x↦enat t]) ≤ S*
unfolding *inresT-def* **by** (*cases S*) (*auto dest*!: *le-funD*[**where** *x=x*] *simp*: *le-funI less-eq-option-def split*: *option.splits*)

lemma *inresT-mono*: *inresT S x t ⟹ t' ≤ t ⟹ inresT S x t'*
unfolding *inresT-def* **by** (*cases S*) (*auto simp add*: *order-subst2*)

lemma *inresT-RETURNT*[*simp*]: *inresT (RETURNT x) y t ⟷ t = 0 ∧ y = x*
by(*auto simp*: *inresT-def RETURNT-def enat-0-iff split*: *nrest.splits*)

lemma *inresT-FAILT*[*simp*]: *inresT FAILT r t*
by(*simp add*: *inresT-def*)

lemma *fail-inresT*[*refine-pw-simps*]: *¬ nofailT M ⟹ inresT M x t*
unfolding *nofailT-def* **by** *simp*

lemma *pw-inresT-Sup*[*refine-pw-simps*]: *inresT (Sup X) r t ⟷ (∃ M∈X. ∃ t'≥t. inresT M r t')*
proof
assume *a*: *inresT (Sup X) r t*
show *(∃ M∈X. ∃ t'≥t. inresT M r t')*
proof(*cases Sup X*)
case *FAILi*
then show *?thesis*
by (*force simp*: *nrest-Sup-FAILT*)
next
case (*REST Y*)
with *a* **obtain** *t'* **where** *t'*: *Y r = Some t' enat t≤t'*
by (*auto simp add*: *inresT-def*)
from *REST* **have** *Y*: *Y = (⨆ {f. SPECT f ∈ X})*
by (*auto simp add*: *Sup-nrest-def split*: *if-splits*)

```
    with t' REST have (⨆ {f r | f . SPECT f ∈ X}) = Some t'
      using nrest-Sup-SPECT-D by fastforce

    with t' Y obtain f t'' where f-t'': SPECT f ∈ X f r = Some t'' t' = Sup {t'
| f t'. SPECT f∈X ∧ f r = Some t'}
      by (auto simp add: SUP-eq-Some-iff Sup-apply[where A= {f. SPECT f ∈
X}])
    from f-t''(3) t'(2) have enat t ≤ Sup {t' | f t'. SPECT f∈X ∧ f r = Some t'}

      by blast
    with f-t'' obtain f' t''' where SPECT f' ∈ X f' r = Some t''' enat t ≤ t'''
      by (smt (verit) Sup-enat-less empty-iff mem-Collect-eq option.sel)
    with REST show ?thesis
      by (intro bexI[of - SPECT f'] exI[of - t]) (auto simp add: inresT-def)
  qed
next
  assume a: (∃ M∈X. ∃ t'≥t.  inresT M r t')
  from this obtain M t' where M-t': M∈X t'≥t inresT M r t'
    by blast

  show inresT (Sup X) r t
  proof (cases Sup X)
    case FAILi
    then show ?thesis
      by (auto simp: nrest-Sup-FAILT top-Sup)
  next
    case (REST Y)
    with M-t' have Y = ⨆ {f. SPECT f ∈ X}
      by (auto simp add: Sup-nrest-def split: if-splits)
    from REST have nf: FAILT ∉ X
      using FAILT-SUP by fastforce
    with M-t' REST obtain f t'' where SPECT f ∈ X f r = Some t'' enat t' ≤
t''
      by (auto simp add: inresT-def split: nrest.splits)
    with REST  M-t'(2) obtain a where Y r = Some a enat t ≤ a
    by (metis Sup-upper enat-ord-simps(1) le-fun-def le-some-optE nres-order-simps(2)
order-trans)
    then show ?thesis
      by (auto simp: REST inresT-def)
  qed
qed
lemma inresT-REST[simp]:
  inresT (REST X) x t ⟷ (∃ t'≥t. X x = Some t')
  unfolding inresT-def
  by auto

lemma pw-Sup-nofail[refine-pw-simps]: nofailT (Sup X) ⟷ (∀ x∈X. nofailT x)
  by (cases Sup X) (auto simp add: Sup-nrest-def nofailT-def split: if-splits)
```

lemma *inres-simps*[*simp*]:
inresT FAILT = (λ- -. *True*)
inresT SUCCEEDT = (λ- - . *False*)
unfolding *inresT-def* [*abs-def*]
by (*auto split*: *nrest.splits simp add*: *RETURNT-def*)

lemma *pw-le-iff*:
$S \leq S' \longleftrightarrow$ (*nofailT S'* $\longrightarrow$ (*nofailT S* $\wedge$ ($\forall$ *x t*. *inresT S x t* $\longrightarrow$ *inresT S' x t*)))
proof (*cases S*; *cases S'*)
fix *R R'*
assume *a*[*simp*]: *S*=*SPECT R S'*=*SPECT R'*
show $S \leq S' \longleftrightarrow$ (*nofailT S'* $\longrightarrow$ (*nofailT S* $\wedge$ ($\forall$ *x t*. *inresT S x t* $\longrightarrow$ *inresT S' x t*))) (**is** *?l* $\longleftrightarrow$ *?r*)
proof (*rule iffI*; *safe*)
assume *b*: *nofailT S* $\forall$ *x t*. *inresT S x t* $\longrightarrow$ *inresT S' x t*
hence *b-2'*: *inresT S x t* $\Longrightarrow$ *inresT S' x t* **for** *x t*
by *blast*
from *b*(*1*) **have** *nofailT S'*
by *auto*
with *a b-2'* **have** *nf*: *R x* $\neq$ *None* $\Longrightarrow$ *R' x* $\neq$ *None* **for** *x*
by *simp* (*metis enat-ile nle-le not-Some-eq*)
have *R x* $\leq$ *R' x* **for** *x*
proof (*cases R x*; *cases R' x*)
fix *r r'*
assume *c*[*simp*]: *R x* = *Some r R' x* = *Some r'*
show *?thesis*
proof(*rule ccontr*)
assume ¬ *R x* $\leq$ *R' x*
from *this* **obtain** *vt* **where** *inresT S x vt* ¬*inresT S' x vt*
by *simp* (*metis Suc-ile-eq enat.exhaust linorder-le-less-linear order-less-irrefl*)
with *b-2'* **show** *False*
by *blast*
qed
qed (*use nf* **in** ‹*auto simp add*: *inresT-def nofailT-def*›)
then show *S* $\leq$ *S'*
by (*auto intro*!: *le-funI*)
qed(*fastforce simp add*: *less-eq-option-def le-fun-def split*: *option.splits*)+
qed *auto*

lemma *RETURN-le-RETURN-iff*[*simp*]: *RETURNT x* $\leq$ *RETURNT y* $\longleftrightarrow$ *x*=*y*
by (*auto simp add*: *pw-le-iff*)

lemma *S*$\leq$*S'* $\Longrightarrow$ *inresT S x t* $\Longrightarrow$ *inresT S' x t*
unfolding *inresT-alt* **by** *auto*

lemma *pw-eq-iff*:
S=*S'* $\longleftrightarrow$ (*nofailT S* = *nofailT S'* $\wedge$ ($\forall$ *x t*. *inresT S x t* $\longleftrightarrow$ *inresT S' x t*))
by (*auto intro*: *antisym simp add*: *pw-le-iff*)

lemma *pw-flat-ge-iff*: *flat-ge S S′* ⟷
(*nofailT S*) ⟶ *nofailT S′* ∧ (∀ *x t*. *inresT S x t* ⟷ *inresT S′ x t*)
by (*metis flat-ord-def nofailT-def pw-eq-iff*)

lemma *pw-eqI*:
assumes *nofailT S* = *nofailT S′*
assumes ⋀*x t*. *inresT S x t* ⟷ *inresT S′ x t*
shows *S*=*S′*
using *assms* **by** (*simp add*: *pw-eq-iff*)

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(*cases M*; *cases M′*)
fix *m m′*
assume *a*[*simp*]: *M*=*REST m M′*=*REST m′*
from *assms*(*2*) **have** *p*: *m x* ≤ *m′ x* **for** *x*
by (*auto dest*: *le-funD*)
from *assms*(*1*) **have** *map-option* ((+) *t*) (*m x*) ≤ *map-option* ((+) *t′*) (*m′ x*)
for *x*
using *p*[*of x*] **by** (*cases m′ x*; *cases m x*) (*auto intro*: *add-mono*)
then show *?thesis*
by (*auto intro*: *le-funI simp add*: *consume-def*)
qed (*use assms*(*2*) **in** ‹*auto simp add*: *consume-def*›)

lemma *nofailT-SPEC*[*refine-pw-simps*]: *nofailT* (*SPEC a b*)
unfolding *SPEC-def* **by** *auto*

lemma *inresT-SPEC*[*refine-pw-simps*]: *inresT* (*SPEC a b*) = (λ*x t*. *a x* ∧ *b x* ≥ *t*)
unfolding *SPEC-def inresT-REST* **by** (*auto split*: *if-splits*)

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})
  unfolding bindT-def consume-def by simp

lemma bindT (REST X) f =
  (⨆x ∈ dom X. consume (f x) (the (X x)))
proof −
  have ∗: ⋀f X. { f x t |x t. X x = Some t}
     = (λx. f x (the (X x))) ‘ (dom X)
    by force
  show ?thesis by (auto simp: bindT-alt ∗)
qed

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

lemma bindT-FAIL[simp]: bindT FAILT g = FAILT
  by (auto simp: bindT-def)

lemma bindT SUCCEEDT f = SUCCEEDT
  unfolding bindT-def by (auto split: nrest.split simp add: bot-nrest-def)

lemma m r = Some ∞ ⟹ inresT (REST m) r t
  by auto

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(intro iffI impI)
  assume ?l and nofailT m
  show ∃r' t' t''. inresT m r' t' ∧ inresT (f r') r t'' ∧ t ≤ t' + t''
  proof(cases m)
    case [simp]: (REST X)
    with ‹?l› obtain M x t1 t' where
      parts: X x = Some t1
      (∀x2. f x = SPECT x2 ⟶
        M = SPECT (map-option ((+) t1) ∘ x2))
      t'≥t inresT M r t'
     by (auto simp add: pw-inresT-Sup bindT-def simp flip: nrest-noREST-FAILT
split: nrest.splits)

    show ?thesis
    proof(cases f x)
      case FAILi
      show ?thesis
        by (rule exI[where x=x], cases t1) (auto intro: le-add2 simp add: FAILi
parts(1))
    next
      case [simp]: (REST re)
      with parts(2) have M: M = SPECT (map-option ((+) t1) ∘ re)
```

```
      by blast
    with parts(4) obtain rer where rer[simp]: re r = Some rer
      by auto
    show ?thesis
    proof(rule exI[where x=x])
     from M parts(1,3,4) show ∃ t' t''. inresT m x t' ∧ inresT (f x) r t'' ∧ t ≤
t' + t''
       by (cases t1;cases rer) (fastforce intro: le-add2)+
    qed
   qed
  qed (use ‹nofailT m› in auto)
next
  assume ?r
  show ?l
  proof(cases m)
   case [simp]: (REST X)
   then show ?thesis
   proof(cases nofailT m)
    case True
    with ‹?r› obtain t' t'' t''' r' where
     parts: enat t' ≤ t'''
     X r' = Some t'''
     inresT (f r') r t''
     t ≤ t' + t''
     by (auto simp: bindT-def split: nrest.splits)
    then show ?thesis
    proof(cases f r')
     case FAILi
     with parts(2) show ?thesis
      by (fastforce simp add: pw-inresT-Sup bindT-def split: nrest.splits)
    next
     case [simp]: (REST x)
     from parts(3) obtain ta where ta: x r = Some ta enat t''≤ta
      by auto
     with parts True obtain tf where tf: tf≥t enat tf ≤ t''' + ta
      using add-mono by fastforce
     with ta parts True show ?thesis
      by (force simp add: pw-inresT-Sup bindT-def split: nrest.splits
          intro!: exI[where x=REST (map-option ((+) t''') ∘ x)])
    qed
   qed auto
  qed auto
qed

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''))
 by (simp add: pw-inresT-bindT-aux) (metis add-le-imp-le-left inresT-mono le-iff-add
nat-le-linear)
```

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
  assume *?l*
  then show *?r*
    by (*force elim*: *no-FAILTE simp*: *bindT-def refine-pw-simps split*: *nrest.splits*)

next
  assume *?r*
  hence *a*: *nofailT M* ⋀*x t* . *inresT M x t* ⟹ *nofailT* (*f x*)
    by *auto*
  show *?l*
  proof(*cases M*)
    case *FAILi*
    then show *?thesis*
      using ‹*?r*› **by** (*simp add*: *nofailT-def*)
  next
    case [*simp*]: (*REST m*)
    with *a* **have** *f x* ≠ *FAILi* **if** *m x* ≠ *None* **for** *x*
      using *that i0-lb* **by** (*auto simp add*: *nofailT-def zero-enat-def*)
    then show *?thesis*
      by (*force simp add*: *bindT-def nofailT-def nrest-Sup-FAILT split*: *nrest.splits*)
  qed
qed

lemma *nat-plus-0-is-id*[*simp*]: ((+) (*0*::*enat*)) = *id* **by** *auto*

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

5 Monad Rules

lemma *nres-bind-left-identity*[*simp*]: *bindT* (*RETURNT x*) *f* = *f x*
  unfolding *bindT-def RETURNT-def*
  by(*auto split*: *nrest.split*)

lemma *nres-bind-right-identity*[*simp*]: *bindT M RETURNT* = *M*
  by(*auto intro*!: *pw-eqI simp*: *refine-pw-simps*)

lemma *nres-bind-assoc*[*simp*]: *bindT* (*bindT M* (λ*x*. *f x*)) *g* = *bindT M* (%*x*. *bindT* (*f x*) *g*)
proof (*rule pw-eqI*)
  fix *x t*
  show *inresT* (*M* ⤜ *f* ⤜ *g*) *x t* = *inresT* (*M* ⤜ (λ*x*. *f x* ⤜ *g*)) *x t*
    by (*simp add*: *refine-pw-simps*) (*use inresT-mono* **in** ‹*fastforce*›)
qed (*fastforce simp add*: *refine-pw-simps*)

6 Monotonicity lemmas

lemma *bindT-mono*:
$m \leq m' \Longrightarrow (\bigwedge x.\ (\exists\, t.\ inresT\ m\ x\ t) \Longrightarrow nofailT\ m' \Longrightarrow\ f\ x \leq f'\ x)$
$\Longrightarrow bindT\ m\ f \leq bindT\ \ m'\ f'$
by(*fastforce simp*: *pw-le-iff refine-pw-simps*)

lemma *bindT-mono'*[*refine-mono*]:
$m \leq m' \Longrightarrow (\bigwedge x.\ \ f\ x \leq f'\ x)$
$\Longrightarrow bindT\ m\ f \leq bindT\ \ m'\ f'$
by (*erule bindT-mono*)

lemma *bindT-flat-mono*[*refine-mono*]:
$[\![\ flat\text{-}ge\ M\ M';\ \bigwedge x.\ flat\text{-}ge\ (f\ x)\ (f'\ x)\ ]\!] \Longrightarrow flat\text{-}ge\ (bindT\ M\ f)\ (bindT\ M'\ f')$
by (*fastforce simp*: *refine-pw-simps pw-flat-ge-iff*)

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*]: $ASSERT\ True = RETURNT\ ()$
by (*auto simp*: *ASSERT-def iASSERT-def*)
lemma *ASSERT-False*[*simp*]: $ASSERT\ False = FAILT$
by (*auto simp*: *ASSERT-def iASSERT-def*)

lemma *bind-ASSERT-eq-if*: $do\ \{\ ASSERT\ \Phi;\ m\ \} = (if\ \Phi\ then\ m\ else\ FAILT)$
unfolding *ASSERT-def iASSERT-def* **by** *simp*

lemma *pw-ASSERT*[*refine-pw-simps*]:
$nofailT\ (ASSERT\ \Phi) \longleftrightarrow \Phi$
$inresT\ (ASSERT\ \Phi)\ x\ 0$
by (*cases* Φ, *simp-all*)+

lemma *ASSERT-refine*: $(Q \Longrightarrow P) \Longrightarrow ASSERT\ P \leq ASSERT\ Q$
by(*auto simp*: *pw-le-iff refine-pw-simps*)

lemma *ASSERT-leI*: $\Phi \Longrightarrow (\Phi \Longrightarrow M \leq M') \Longrightarrow ASSERT\ \Phi \gg\!= (\lambda\text{-}.\ M) \leq M'$
by(*auto simp*: *pw-le-iff refine-pw-simps*)

lemma *le-ASSERTI*: $(\Phi \Longrightarrow M \leq M') \Longrightarrow M \leq ASSERT\ \Phi \gg\!= (\lambda\text{-}.\ M')$
by(*auto simp*: *pw-le-iff refine-pw-simps*)

lemma *inresT-ASSERT*: $inresT\ (ASSERT\ Q \gg\!= (\lambda\text{-}.\ M))\ x\ ta = (Q \longrightarrow inresT\ M\ x\ ta)$
unfolding *ASSERT-def iASSERT-def* **by** *auto*

lemma *nofailT-ASSERT-bind*: *nofailT* (*ASSERT P* $\gg\!=$ (λ-. *M*)) ⟷ (*P* ∧ *nofailT M*)
by(*auto simp*: *pw-bindT-nofailT pw-ASSERT*)

6.2 SELECT

definition *emb'* **where** ⋀*Q T*. *emb' Q* (*T*::'*a* ⇒ *enat*) = (λ*x*. *if Q x then Some* (*T x*) *else None*)

abbreviation *emb Q t* ≡ *emb' Q* (λ-. *t*)

lemma *emb-eq-Some-conv*: ⋀*T*. *emb' Q T x* = *Some t'* ⟷ (*t'*=*T x* ∧ *Q x*)
by (*auto simp*: *emb'-def*)

lemma *emb-le-Some-conv*: ⋀*T*. *Some t'* ≤ *emb' Q T x* ⟷ (*t'*≤*T x* ∧ *Q x*)
by (*auto simp*: *emb'-def*)

lemma *SPEC-REST-emb'-conv*: *SPEC P t* = *REST* (*emb' P t*)
unfolding *SPEC-def emb'-def* **by** *auto*

lemma *SPECT-ub*: *T*≤*T'* ⟹ *SPECT* (*emb' M' T*) ≤ *SPECT* (*emb' M' T'*)
unfolding *emb'-def* **by** (*auto simp*: *pw-le-iff le-funD order-trans refine-pw-simps*)

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

definition *SELECT* :: ('*a* ⇒ *bool*) ⇒ *enat* ⇒ '*a option nrest*
where *SELECT P tf* ≡ *if* ∃*x*. *P x then REST* (*emb* (λ*r*. *case r of Some p* ⇒ *P p* | *None* ⇒ *False*) *tf*)
else REST [*None* ↦ *tf*]

lemma *inresT-SELECT-Some*: *inresT* (*SELECT Q tt*) (*Some x*) *t'* ⟷ (*Q x* ∧ (*t'* ≤ *tt*))
by(*auto simp*: *inresT-def SELECT-def emb'-def*)

lemma *inresT-SELECT-None*: *inresT* (*SELECT Q tt*) *None t'* ⟷ (¬(∃*x*. *Q x*) ∧ (*t'* ≤ *tt*))
by(*auto simp*: *inresT-def SELECT-def emb'-def*)

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*))
by(*auto simp*: *inresT-def SELECT-def emb'-def*)

lemma *nofailT-SELECT*[*refine-pw-simps*]: *nofailT* (*SELECT Q tt*)
by(*auto simp*: *nofailT-def SELECT-def*)

lemma *s1*: *SELECT P T* ≤ (*SELECT P T'*) ⟷ *T* ≤ *T'*
by (*cases* ∃*x*. *P x*; *cases T*; *cases T'*)(*auto simp*: *pw-le-iff refine-pw-simps not-le split*: *option.splits*)

lemma *s2*: *SELECT P T* ≤ (*SELECT P′ T*) ⟷ (
  (*Ex P′* ⟶ *Ex P*) ∧ (∀ *x*. *P x* ⟶ *P′ x*))
 by (*cases T*) (*auto simp*: *pw-le-iff refine-pw-simps split*: *option.splits*)

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 −
 have *SELECT P T* ≤ *SELECT P T′*
  using *s1 assms*(*3*) **by** *auto*
 also have . . . ≤ *SELECT P′ T′*
  unfolding *s2* **using** *assms*(*1*,*2*) **by** *auto*
 finally show *?thesis* **.**
qed

7 RECT

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

lemma *trimonoD-flatf-ge*: *mono2 B* ⟹ *flatf-mono-ge B*
 unfolding *mono2-def* **by** *auto*

lemma *trimonoD-mono*: *mono2 B* ⟹ *mono B*
 unfolding *mono2-def* **by** *auto*

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*))
 unfolding *RECT-def*
 by (*auto simp*: *gfp-eq-flatf-gfp*[*OF trimonoD-flatf-ge trimonoD-mono*])

lemma *RECT-unfold*: ⟦*mono2 B*⟧ ⟹ *RECT B* = *B* (*RECT B*)
 unfolding *RECT-def* [*abs-def*]
 by (*auto dest*: *trimonoD-mono simp*: *gfp-unfold*[*symmetric*])

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*
unfolding *mono2-def* **by** *auto*

lemma *mono-fun-transform*[*refine-mono*]: (⋀*f g x*. (⋀*x*. *f x* ≤ *g x*) ⟹ *B f x* ≤ *B g x*) ⟹ *mono B*
by (*intro monoI le-funI*) (*simp add*: *le-funD*)

lemma *whileT-unfold*: *whileT b c* = (λ*s*. (*if b s then bindT* (*c s*) (*whileT b c*) *else RETURNT s*))
unfolding *whileT-def* **by** (*rule RECT-unfold*) (*refine-mono*)

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*)
unfolding *RECT-def* **by** *simp* (*meson LE gfp-mono le-fun-def*)

lemma *whileT-mono*:
assumes ⋀*x*. *b x* ⟹ *c x* ≤ *c' x*
shows (*whileT b c x*) ≤ (*whileT b c' x*)
unfolding *whileT-def* **proof** (*rule RECT-mono*)
show (*if b x then c x* ⤜ *F else RETURNT x*) ≤ (*if b x then c' x* ⤜ *F else RETURNT x*) **for** *F x*
using *assms* **by** (*auto intro*: *bindT-mono*)
qed *refine-mono*

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*)
using *wf*
apply *induction*
apply (*subst fp*)
apply *fact*
done

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*)

using *wf-fp-induct*[**where** *f*=*RECT B* **and** *B*=*B*] *RECT-unfold assms*
by *metis*

theorem *RECT-wf-induct*[*consumes 1*]:
assumes *RECT B x* = *r*
assumes *wf R*
and *mono2 B*
and ⋀*x D r*. (⋀*y r*. (*y*, *x*) ∈ *R* ⟹ *D y* = *r* ⟹ *P y r*) ⟹ *B D x* = *r* ⟹ *P x r*
shows *P x r*

using *RECT-wf-induct-aux*[**where** *P* = λ*x fx*. *P x fx*] *assms* **by** *metis*

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*
unfolding *mm-def* **by** (*cases M x*) (*auto split*: *option.splits elim*!: *le-some-optE intro*!: *helper*)

lemma *mm-antimono*: *M1 x* ≥ *M2 x* ⟹ *mm Q M1 x* ≤ *mm Q M2 x*
unfolding *mm-def* **by** (*auto split*: *option.splits intro*: *helper2*)

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(*rule antisym*)
show *mm* (λ*x*. ⨅ {*u*. ∃ *y*. *u* = *f y x*}) *m x* ≤ ⨅ {*u*. ∃ *y*. *u* = *mm* (*f y*) *m x*}

```
  proof (rule Inf-greatest, drule CollectD)
    fix z
    assume z: ∃ y. z = mm (f y) m x
    from this obtain y where y: z = mm (f y) m x
      by blast

    show mm (λx. ⨅ {u. ∃ y. u = f y x}) m x ≤ z
    proof (cases Inf {u. ∃ y. u = f y x})
      case None
      with z show ?thesis
        by (cases m x) (auto simp add: mm-def None)
    next
      case Some-Inf[simp]: (Some l)
      then have i: ⋀y. f y x ≥ Some l
        by (metis (mono-tags, lifting) Inf-lower mem-Collect-eq)
      then have I: ⋀y. mm (λx. Inf {u. ∃ y. u = f y x}) m x ≤ mm (f y) m x
        by (intro mm-mono) simp
      show ?thesis
      proof(cases m x)
        case None
        with y show ?thesis
          by (auto simp add: mm-def)
      next
        case [simp]: (Some a)
        with y I show ?thesis
          by (auto simp add: mm-def)
      qed
    qed
  qed
  show ⨅ {u. ∃ y. u = mm (f y) m x} ≤ mm (λx. ⨅ {u. ∃ y. u = f y x}) m x
  proof(rule Inf-lower, rule CollectI)
    have ∃ y. Inf {u. ∃ y. u = f y x} = f y x
    proof (cases Option.these {u. ∃ y. u = f y x} = {})
      case True
      then show ?thesis
        by (simp add: Inf-option-def Inf-enat-def these-empty-eq) blast
    next
      case False
      then show ?thesis
        by (auto simp add: Inf-option-def Inf-enat-def in-these-eq intro: LeastI)
    qed
    then obtain y where z: Inf {u. ∃ y. u = f y x} = f y x
      by blast
    show ∃ y. mm (λx. Inf {u. ∃ y. u = f y x}) m x = mm (f y) m x
      by (rule exI[where x=y]) (unfold mm-def z, rule refl)
  qed
qed

definition mm2 :: ( 'a enat option) ⇒ ( 'b enat option) ⇒ ( 'c 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) unfolding mm-def mm2-def ..

lemma mm2-None[simp]: mm2 q None = Some ∞ unfolding mm2-def by auto

lemma mm2-Some0[simp]: mm2 q (Some 0) = q unfolding mm2-def by (auto
split: option.splits)

lemma mm2-antimono: x ≤ y ⟹ mm2 q x ≥ mm2 q y
  unfolding mm2-def by (auto split: option.splits intro: helper2)

lemma mm2-contiuous2:
  assumes ∀ x∈X. t ≤ mm2 q x shows t ≤ mm2 q (Sup X)
proof (cases q; cases Sup X)
  fix q' assume q[simp]: q = Some q'
  fix S assume SupX[]: Sup X = Some S

  have t ≤ (if q' < S then None else Some (q' − S))
  proof(cases q' < S)
    case True
    with SupX obtain x where x ∈ Option.these X q' < x
      using less-Sup-iff by (auto simp add: Sup-option-def split: if-splits)
    hence Some x ∈ X q < Some x
      by (auto simp add: in-these-eq)
    with True assms show ?thesis
      by (auto simp add: mm2-def)
  next
    case False
    with assms SupX show ?thesis
      by (cases q'; cases S) (force simp: mm2-def dest: Sup-finite-enat)+
  qed
  then show ?thesis
    by (auto simp add: mm2-def SupX)
qed (use assms in ‹auto simp add: mm2-def Sup-option-def split: option.splits
if-splits›)

lemma fl: (a::enat) − b = ∞ ⟹ a = ∞
  by(cases b; cases a) auto

lemma mm-inf1: mm R m x = Some ∞ ⟹ m x = None ∨ R x = Some ∞
  by (auto simp: mm-def split: option.splits if-splits intro: fl)

lemma mm-inf2: m x = None ⟹ mm R m x = Some ∞
  by(auto simp: mm-def split: option.splits if-splits)
```

lemma *mm-inf3*: $R\ x = Some\ \infty \Longrightarrow mm\ R\ m\ x = Some\ \infty$
by(*auto simp*: *mm-def split*: *option.splits if-splits*)

lemma *mm-inf*: $mm\ R\ m\ x = Some\ \infty \longleftrightarrow m\ x = None \vee R\ x = Some\ \infty$
using *mm-inf1 mm-inf2 mm-inf3* **by** *metis*

lemma *InfQ-E*: $Inf\ Q = Some\ t \Longrightarrow None \notin Q$
unfolding *Inf-option-def* **by** *auto*

lemma *InfQ-iff*: $(\exists\, t' \geq enat\ t.\ Inf\ Q = Some\ t') \longleftrightarrow None \notin Q \wedge Inf\ (Option.these\ Q) \geq t$
unfolding *Inf-option-def*
by *auto*

lemma *mm2-fst-None*[*simp*]: $mm2\ None\ q = (case\ q\ of\ None \Rightarrow Some\ \infty \mid - \Rightarrow None)$
by (*cases q*) (*auto simp*: *mm2-def*)

lemma *mm2-auxXX1*: $Some\ t \leq mm2\ (Q\ x)\ (Some\ t') \Longrightarrow Some\ t' \leq mm2\ (Q\ x)\ (Some\ t)$
by (*auto dest*: *fl simp*: *less-eq-enat-def mm2-def split*: *enat.splits option.splits if-splits*)

8.2 mii

definition *mii* :: $('a \Rightarrow enat\ option) \Rightarrow 'a\ nrest \Rightarrow 'a \Rightarrow enat\ option$ **where**
$mii\ Qf\ M\ x = (case\ M\ of\ FAILi \Rightarrow None \mid REST\ Mf \Rightarrow (mm\ Qf\ Mf)\ x)$

lemma *mii-alt*: $mii\ Qf\ M\ x = (case\ M\ of\ FAILi \Rightarrow None \mid REST\ Mf \Rightarrow (mm2\ (Qf\ x)\ (Mf\ x)))$
unfolding *mii-def mm-alt* **..**

lemma *mii-continuous*: $mii\ (\lambda x.\ Inf\ \{f\ y\ x | y.\ True\})\ m\ x = Inf\ \{mii\ (\%x.\ f\ y\ x)\ m\ x | y.\ True\}$
unfolding *mii-def* **by** (*cases m*) (*auto simp add*: *mm-continous*)

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(*intro iffI allI impI*)
fix *y t1*
assume *a*: $t \leq mii\ Q\ (\bigsqcup\ \{F\ x\ t1\ |x\ t1.\ P\ x\ t1\})\ x\ P\ y\ t1$
then show $t \leq mii\ Q\ (F\ y\ t1)\ x$
proof(*cases F y t1*)
case *REST-F*[*simp*]: (*REST Ff*)
then show *?thesis*

proof(*cases* ($\bigsqcup$ {*F x t1* |*x t1*. *P x t1*}))
case *FAILi*
with *a* **show** *?thesis*
using *a* **by** (*simp add*: *mii-alt less-eq-option-None-is-None'*)
next
case [*simp*]: (*REST Sf*)
note *Sf-x* = *nrest-Sup-SPECT-D*[*OF this*, **where** *x*=*x*]
from *a*(*1*) **have** *t* ≤ *mm2* (*Q x*) (*Sf x*)
by (*auto simp add*: *mii-alt*)
also from *a*(*2*) **have** *mm2* (*Q x*) (*Sf x*) ≤ *mm2* (*Q x*) (*Ff x*)
by (*intro mm2-antimono*) (*force intro*: *Sup-upper simp add*: *Sf-x*)
finally show *?thesis*
by (*simp add*: *mii-alt*)
qed
qed (*auto simp*: *mii-alt Sup-nrest-def split*: *if-splits*)
next
assume ∀ *y t1*. *P y t1* ⟶ *t* ≤ *mii Q* (*F y t1*) *x*
then show *t* ≤ *mii Q* ($\bigsqcup$ {*F x t1* |*x t1*. *P x t1*}) *x*
by (*auto simp*: *mii-alt Sup-nrest-def split*: *nrest.splits intro*: *mm2-contiuous2*)
qed

lemma *mii-inf*: *mii Qf M x* = *Some* ∞ ⟷ (∃ *Mf*. *M* = *SPECT Mf* ∧ (*Mf x* = *None* ∨ *Qf x* = *Some* ∞))
by (*auto simp*: *mii-def mm-inf split*: *nrest.split*)

lemma *miiFailt*: *mii Q FAILT x* = *None*
unfolding *mii-def* **by** *auto*

8.3 lst - latest starting time

definition *lst* :: *'a nrest* ⇒ (*'a* ⇒ *enat option*) ⇒ *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*)
unfolding *lst-def* **by** (*simp add*: *full-SetCompr-eq*)

lemma *T-pw*: *lst M Q* ≥ *t* ⟷ (∀ *x*. *mii Q M x* ≥ *t*)
by (*auto simp add*: *T-alt-def mii-alt le-Inf-iff*)

lemma *T-specifies-I*:
assumes *lst m Q* ≥ *Some 0* **shows** *m* ≤ *SPECT Q*
proof (*cases m*)
case [*simp*]: (*REST q*)
from *assms* **have** *q x* ≤ *Q x* **for** *x*
by (*cases q x*; *cases Q x*)
(*force simp add*: *T-alt-def mii-alt mm2-def le-Inf-iff split*: *option.splits if-splits nrest.splits*)+
then show *?thesis*

by (*auto intro*: *le-funI*)
qed (*use assms* **in** ‹*auto simp add*: *T-alt-def mii-alt*›)

lemma *T-specifies-rev*:
assumes $m \leq SPECT\ Q$ **shows** $lst\ m\ Q \geq Some\ 0$
proof (*cases m*)
case [*simp*]: (*REST q*)
with *assms* **have** *le*: $q\ x \leq Q\ x$ **for** *x*
by (*auto dest*: *le-funD*)
show *?thesis*
proof(*subst T-pw*, *rule allI*)
fix *x*
from *le*[*of x*] **show** $Some\ 0 \leq mii\ Q\ m\ x$
by (*cases q x*; *cases Q x*) (*auto simp add*: *mii-alt mm2-def*)
qed
qed (*use assms* **in** ‹*auto simp add*: *T-alt-def mii-alt*›)

lemma *T-specifies*: $lst\ m\ Q \geq Some\ 0 = (m \leq SPECT\ Q)$
using *T-specifies-I T-specifies-rev* **by** *metis*

lemma *pointwise-lesseq*:
fixes $x :: 'a{::}order$
shows $(\forall t.\ x \geq t \longrightarrow x' \geq t) \Longrightarrow x \leq x'$
by *simp*

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 $lst\ \ M\ Q \leq lst\ \ M'\ Q'$
apply(*rule pointwise-lesseq*)
using *assms* **unfolding** *T-pw nres3-def* **by** *metis*

lemma assumes $\bigwedge t.\ (\forall x.\ nres3\ Q\ M\ x\ t) = (\forall x.\ nres3\ Q'\ M'\ x\ t)$
shows *pw-T-eq-iff*: $lst\ M\ Q\ \ = lst\ M'\ Q'$
apply (*rule antisym*)
apply(*rule pw-T-le*) **using** *assms* **apply** *metis*
apply(*rule pw-T-le*) **using** *assms* **apply** *metis*
done

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*: $lst\ M\ Q = lst\ M'\ Q'$
apply (*rule antisym*)
apply(*rule pw-T-le*)
apply *fact*

```
  apply(rule pw-T-le)
  apply fact
  done

lemma lem: ∀ t1. M y = Some t1 ⟶ t ≤ mii Q (SPECT (map-option ((+) t1)
∘ x2)) x ⟹ f y = SPECT x2 ⟹ t ≤ mii (λy. mii Q (f y) x) (SPECT M) y
proof(cases M y; cases t)
  fix m' t'
  assume a: ∀ t1. M y = Some t1 ⟶ t ≤ mii Q (SPECT (map-option ((+) t1)
∘ x2)) x f y = SPECT x2
  assume c[simp]: M y = Some m' t = Some t'

  from a c show t ≤ mii (λy. mii Q (f y) x) (SPECT M) y
  proof (cases x2 x; cases Q x)
    fix x2' Q'
    assume c'[simp]: x2 x = Some x2' Q x = Some Q'
    from a c show ?thesis
      by (cases m'; cases x2'; cases Q') (auto split: option.splits if-splits simp add:
add.commute mii-def mm-def)
  qed (auto split: option.splits if-splits simp add: add.commute mii-def mm-def)
qed(auto simp add: mii-def mm-def)


lemma diff-diff-add-enat:a − (b+c) = a − b − (c::enat)
  by (cases a; cases b; cases c) auto


lemma lem2: t ≤ mii (λy. mii Q (f y) x) (SPECT M) y ⟹ M y = Some t1 ⟹
f y = SPECT fF ⟹ t ≤ mii Q (SPECT (map-option ((+) t1) ∘ fF)) x
 by (cases fF x; cases Q x; cases t) (auto simp add: mii-def mm-def enat-plus-minus-aux2
     add.commute linorder-not-less diff-diff-add-enat split: if-splits)

lemma fixes m :: 'b nrest
  shows mii-bindT: (t ≤ mii Q (bindT m f) x) ⟷ (∀ y. t ≤ mii (λy. mii Q (f
y) x) m y)
proof −
  { fix M
    assume mM: m = SPECT M
    let ?P = %x t1. M x = Some t1
   let ?F = %x t1. case f x of FAILi ⇒ FAILT | REST m2 ⇒ SPECT (map-option
((+) t1) ∘ m2)
    let ?Sup = (Sup {?F x t1 |x t1. ?P x t1})

    { fix y
      have 1: mii (λy. mii Q (f y) x) (SPECT M) y = None if f y = FAILT M y
≠ None
       using that by (auto simp add: mii-def mm-def less-eq-option-None-is-None'
)
      have (∀ t1. ?P y t1 ⟶ mii Q (?F y t1) x ≥ t)
```

```
        = (t ≤ mii (λy. mii Q (f y) x) m y)
      by (cases f y) (auto intro: lem lem2 meta-le-everything-if-top simp add: 1
mM miiFailt mii-inf
          top-enat-def top-option-def less-eq-option-None-is-None')
  } note h=this


  from mM have mii Q (bindT m f) x = mii Q ?Sup x by (auto simp: bindT-def)
   then have (t ≤ mii Q (bindT m f) x) = (t ≤ mii Q ?Sup x) by simp
    also have ... = (∀ y t1. ?P y t1 ⟶ mii Q (?F y t1) x ≥ t) by (rule
mii-continuous2)
   also have ... = (∀ y. t ≤ mii (λy. mii Q (f y) x) m y) by (simp only: h)
   finally have ?thesis .
 } note bl=this

 show ?thesis
 proof(cases m)
  case FAILi
  then show ?thesis
   by (simp add: mii-def)
 next
  case (REST x2)
  then show ?thesis
   by (rule bl)
 qed
qed


lemma nres3-bindT: (∀ x. nres3 Q (bindT m f) x t) = (∀ y. nres3 (λy. lst (f y)
Q) m y t)
proof −
 have t: ⋀f and t::enat option. (∀ y. t ≤ f y) ⟷ (t≤Inf {f y|y. True})
  using le-Inf-iff by fastforce

 have (∀ x. nres3 Q (bindT m f) x t) = (∀ x. t ≤ mii Q (bindT m f) x) unfolding
nres3-def by auto
  also have ... = (∀ x. (∀ y. t ≤ mii (λy. mii Q (f y) x) m y)) by(simp only:
mii-bindT)
 also have ... = (∀ y. (∀ x. t ≤ mii (λy. mii Q (f y) x) m y)) by blast
 also have ... = (∀ y. t ≤ mii (λy. Inf {mii Q (f y) x|x. True}) m y)
  using t by (fastforce simp only: mii-continuous)
 also have ... = (∀ y. t ≤ mii (λy. lst (f y) Q) m y) unfolding lst-def by auto
  also have ... = (∀ y. nres3 (λy. lst (f y) Q) m y t) unfolding nres3-def by
auto
 finally show ?thesis .
 have (∀ y. t ≤ mii (λy. lst (f y) Q) m y) = (t ≤ Inf{ mii (λy. lst (f y) Q) m
y|y. True}) using t by metis
qed
```

8.5 rules for lst

lemma *T-bindT*: *lst* (*bindT M f*) *Q* = *lst M* (λ*y*. *lst* (*f y*) *Q*)
by (*rule pw-T-eq-iff*, *rule nres3-bindT*)

lemma *T-REST*: *lst* (*REST* [*x*↦*t*]) *Q* = *mm2* (*Q x*) (*Some t*)
proof−
have ∗: *Inf* {*uu*. ∃ *xa*. (*xa* = *x* ⟶ *uu*= *v*) ∧ (*xa* ≠ *x* ⟶ *uu* = *Some* ∞)} = *v*
(**is** *Inf ?S* = *v*) **for** *v* :: *enat option*
proof −
have *?S* ∈ { {*v*} ∪ {*Some* ∞}, {*v*} } **by** *auto*
then show *?thesis*
by *safe* (*simp-all add*: *top-enat-def*[*symmetric*] *top-option-def*[*symmetric*])
qed
then show *?thesis*
unfolding *lst-def mii-alt* **by** *auto*
qed

lemma *T-RETURNT*: *lst* (*RETURNT x*) *Q* = *Q x*
unfolding *RETURNT-alt* **by** (*rule trans*, *rule T-REST*) *simp*

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(*cases* ∃ *x*. *P x*)
case *True*
from *p*[*unfolded T-pw mii-alt*] **have**
p′: ⋀*y x*. *P y* ⟹ *Some tt* ≤ *mm2* (*Q x*) ([*Some y* ↦ *tf*] *x*)
by *auto*
hence *p″*: ⋀*y x*. *P y* ⟹ *x*=*Some y* ⟹ *Some tt* ≤ *mm2* (*Q x*) (*Some tf*)
by (*metis fun-upd-same*)
with *True* **show** *?thesis*
by (*auto simp*: *SELECT-def emb′-def T-pw mii-alt split*: *if-splits option.splits*)
next
case *False*
with *assms* **show** *?thesis*
by (*auto simp*: *SELECT-def*)
qed

9 consequence rules

lemma *aux1*: *Some t* ≤ *mm2 Q* (*Some t′*) ⟷ *Some* (*t*+*t′*) ≤ *Q*
proof (*cases t*; *cases t′*; *cases Q*)
fix *n n′ t″*
assume *a*[*simp*]: *t* = *enat n t′* = *enat n′ Q* = *Some t″*
show *?thesis*

```
    by (cases t'') (auto simp: mm2-def)
qed (auto simp: mm2-def elim: less-enatE split: option.splits)

lemma aux1a: (∀ x t''. Q' x = Some t'' ⟶ (Q x) ≥ Some (t + t''))
     = (∀ x. mm2 (Q x) (Q' x) ≥ Some t)
proof(intro iffI allI impI)
  fix x
  assume a: ∀ x t''. Q' x = Some t'' ⟶ Some (t + t'') ≤ Q x
  then show Some t ≤ mm2 (Q x) (Q' x)
    by (cases Q' x) (simp-all add: aux1)
next
  fix x t''
  assume a: ∀ x. Some t ≤ mm2 (Q x) (Q' x) Q' x = Some t''
  then show Some (t + t'') ≤ Q x
    using aux1 by metis
qed

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 −
  {
    fix x
    from assms(1)[unfolded T-pw] have i: Some t' ≤ mii Q' f x by auto
    from assms(2) have ii: ⋀t''. Q' x = Some t'' ⟹ (Q x) ≥ Some ((t − t') +
t'') by auto
    from i ii have Some t ≤ mii Q f x
    proof (cases f)
      case [simp]: (REST Mf)
      then show ?thesis
      proof(cases Mf x)
        case [simp]: (Some a)
        have arith: t' + a ≤ b ⟹ t − t' + b ≤ b' ⟹ t + a ≤ b' for b b'
          by (cases t; cases t'; cases a; cases b; cases b') auto
        with i ii show ?thesis
          by (cases Q' x; cases Q x) (auto simp add: mii-alt aux1)
      qed (auto simp add: mii-alt)
    qed (auto simp add: mii-alt)
  }
  thus ?thesis
    unfolding T-pw ..
qed

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 −
  {
    fix x
    from assms(1)[unfolded T-pw] have i: Some t ≤ mii Q' f x by auto
    from assms(2) have ii: ⋀t'' M.  f = SPECT M ⟹ M x ≠ None ⟹ Q' x
= Some t'' ⟹ (Q x) ≥ Some ( t'')
      by auto
    from i ii have Some t ≤ mii Q f x
    proof (cases f)
      case [simp]: (REST Mf)
      then show ?thesis
      proof(cases Mf x)
        case [simp]: (Some a)
        with i ii show ?thesis
          by (cases Q' x; cases Q x) (fastforce simp add: mii-alt aux1)+
      qed (auto simp add: mii-alt)
    qed (auto simp add: mii-alt)
  }
  thus ?thesis
    unfolding T-pw ..
qed



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
  by (rule T-conseq6) (auto intro: assms(1) dest: assms(2))

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 −
  {
    fix x
    from assms(1)[unfolded T-pw] have i: Some t' ≤ mii Q' f x by auto
    from assms(2) have ii: ⋀t'' M.  f = SPECT M ⟹ M x ≠ None ⟹ Q' x
= Some t'' ⟹ (Q x) ≥ Some ((t − t') + t'') by auto
    from i ii have Some t ≤ mii Q f x

    proof (cases f)
      case [simp]: (REST Mf)
```

```
    then show ?thesis
    proof(cases Mf x)
      case [simp]: (Some a)
      have arith: t' + a ≤ b ⟹ t − t' + b ≤ b' ⟹ t + a ≤ b' for b b'
        by (cases a; cases b; cases b'; cases t; cases t') auto
      with i ii show ?thesis
        by (cases Q' x; cases Q x) (fastforce simp add: mii-alt aux1)+
    qed (auto simp add: mii-alt)
  qed (auto simp add: mii-alt)
 }
 thus ?thesis
  unfolding T-pw ..
qed

lemma T-conseq3:
 assumes
  lst f Q' ≥ Some t'
  ⋀x. mm2 (Q x) (Q' x) ≥ Some (t − t')
 shows lst f Q ≥ Some t
 using assms T-conseq4 aux1a by metis
```

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 (λx. bindT g (λy. RETURNT (x+(y::nat))))

lemma assumes
 f-spec[vcg-rules]: lst f ( emb' (λx. x > 2) (enat o ((*) 2)) ) ≥ Some 0
and
 g-spec[vcg-rules]: lst g ( emb' (λx. x > 2) (enat) ) ≥ Some 0
shows lst (P f g) ( emb' (λx. x > 5) (enat o (*) 3) ) ≥ Some 0
proof −
 have ?thesis
  unfolding P-def by vcg

 have ?thesis
  unfolding P-def
  apply(simp add: T-bindT )
  apply(simp add: T-RETURNT)
  apply(rule T-conseq4[OF f-spec])
  apply(clarsimp simp: emb-eq-Some-conv)
  apply(rule T-conseq4[OF g-spec])
```

apply (*clarsimp simp*: *emb-eq-Some-conv emb-le-Some-conv*)
done
thus *?thesis* **.**
qed
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*
by(*simp add*: *T-bindT*)

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*
unfolding *mm2-def* **by** *auto*

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* **by** (*auto simp*: *mm3-def zero-enat-def*)

lemma *mm3-Some-conv*: (*mm3 t0 A* = *Some t*) = (∃ *t'*. *A* = *Some t'* ∧ *t0* ≥ *t'* ∧ *t*=*t0*−*t'*)
unfolding *mm3-def* **by**(*auto split*: *option.splits*)

lemma *mm3-None*[*simp*]: *mm3 t0 None* = *None* **unfolding** *mm3-def* **by** *auto*

lemma *T-FAILT*[*simp*]: *lst FAILT Q* = *None*
unfolding *lst-def mii-alt* **by** *simp*

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*
by (*auto simp*: *progress-def*)

lemma *progress-FAILT*[*simp*]: *progress FAILT* **by**(*auto simp*: *progress-def*)

11.1 Progress rules

named-theorems *progress-rules*

lemma *progress-SELECT-iff*: *progress* (*SELECT P t*) ⟷ *t* > *0*
unfolding *progress-def SELECT-def emb'-def* **by** (*auto split*: *option.splits*)

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

lemma *progress-REST-iff*: *progress* (*REST* [*x* ↦ *t*]) ⟷ *t*>*0*
by (*auto simp*: *progress-def*)

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

lemma *progress-ASSERT-bind*[*progress-rules*]: ⟦Φ ⟹ *progress* (*f* ()) ⟧ ⟹ *progress* (*ASSERT* Φ⤜*f*)
by (*cases* Φ) (*auto simp*: *progress-def*)

lemma *progress-SPECT-emb*[*progress-rules*]: *t* > *0* ⟹ *progress* (*SPECT* (*emb P t*)) **by**(*auto simp*: *progress-def emb'-def*)

lemma *Sup-Some*: *Sup* (*S*::*enat option set*) = *Some e* ⟹ ∃ *x*∈*S*. (∃ *i*. *x* = *Some i*)
unfolding *Sup-option-def* **by** (*auto split*: *if-splits*)

lemma *progress-bind*[*progress-rules*]: **assumes** *progress m* ∨ (∀ *x*. *progress* (*f x*))
shows *progress* (*m*⤜*f*)
proof (*cases m*)
case *FAILi*
then show *?thesis* **by** (*auto simp*: *progress-def*)
next
case (*REST x2*)
then show *?thesis* **unfolding** *bindT-def progress-def*
proof (*safe*, *goal-cases*)
case (*1 s' M y*)
let *?P* = λ*fa*. ∃ *x*. *f x* ≠ *FAILT* ∧
(∃ *t1*. ∀ *x2a*. *f x* = *SPECT x2a* ⟶ *fa* = *map-option* ((+) *t1*) ∘ *x2a* ∧ *x2 x* = *Some t1*)
from *1* **have** *A*: *Sup* {*fa s'* |*fa*. *?P fa*} = *Some y*
by (*auto dest*: *nrest-Sup-SPECT-D*[**where** *x*=*s'*] *split*: *nrest.splits*)
from *Sup-Some*[*OF this*] **obtain** *fa i* **where** *P*: *?P fa* **and** *3*: *fa s'* = *Some i* **by** *blast*
then obtain *x t1 x2a* **where** *a3*: *f x* = *SPECT x2a*
and ∀ *x2a*. *f x* = *SPECT x2a* ⟶ *fa* = *map-option* ((+) *t1*) ∘ *x2a* **and** *a2*: *x2 x* = *Some t1*
by *fastforce*
then have *a1*: *fa* = *map-option* ((+) *t1*) ∘ *x2a* **by** *auto*
have *progress m* ⟹ *t1* > *0*
by (*drule progressD*) (*use 1*(*1*) *a2 a1 a3* **in** *auto*)
moreover
have *progress* (*f x*) ⟹ *x2a s'* > *Some 0*
using *a1 1*(*1*) *a2 3* **by** (*auto dest*!: *progressD*[*OF - a3*])
ultimately

have *t1 > 0 ∨ x2a s' > Some 0* **using** *assms* **by** *auto*

then have *Some 0 < fa s'* **using** *a1 3* **by** *auto*
also have *... ≤ Sup {fa s'|fa. ?P fa}*
by (*rule Sup-upper*) (*blast intro*: *P*)
also have *... = M s'* **using** *A 1(3)* **by** *simp*
finally show *?case* **.**
qed
qed

lemma *mm2SomeleSome-conv*: *mm2 (Qf) (Some t) ≥ Some 0 ⟷ Qf ≥ Some t*
unfolding *mm2-def* **by** (*auto split*: *option.split*)

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) (λs'. 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 (λx. if b x then None else I x) ≥ Some t*
using *assms*(*1*,*3*)
unfolding *whileT-def*
proof (*induction arbitrary*: *t rule*: *RECT-wf-induct*[**where** *R*=*R*])
case *1*
show *?case* **by** *fact*
next
case *2*
then show *?case* **by** *refine-mono*
next
case *step*: (*3 x D r t'*)
note *IH*[*vcg-rules*] = *step.IH*[*OF - refl*]
note *step.hyps*[*symmetric*, *simp*]

from *step.prems* **show** *?case*
by *simp vcg'*
qed

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) (λs'. 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*
apply(*rule T-conseq4*)
apply(*rule whileT-rule''*[**where** *I*=λ*s*. *mm3 t0* (*I s*)
and *R*=*measure* (*the-enat o the o I*), *OF assms*(*1*)])
subgoal for *s t'*
apply(*cases I s*; *simp*)
subgoal for *ti*
using *IS*[*of s ti*]
apply (*cases c s*; *simp*)
subgoal for *M*
— TODO
using *progress*[*of s*, *THEN progressD*, *of M*]
apply(*auto simp*: *T-pw mm3-Some-conv mii-alt mm2-def mm3-def split*: *option.splits if-splits*)
apply *fastforce*
apply (*metis enat-ord-simps*(*1*) *le-diff-iff le-less-trans option.distinct*(*1*))
apply (*metis diff-is-0-eq' leI less-option-Some option.simps*(*3*) *zero-enat-def*)
apply (*smt Nat.add-diff-assoc enat-ile enat-ord-code*(*1*) *idiff-enat-enat leI le-add-diff-inverse2 nat-le-iff-add option.simps*(*3*))
using *dual-order.trans* **by** *blast*
done
done
by *auto*

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*))
unfolding *whileIET-def*
apply(*rule whileT-rule'''*[*OF refl*, **where** *I*=(λ*e*. *if I e then Some* (*E e*) *else None*)])
using *assms* **by** *auto*

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*))
using *assms* **by** (*cases I s*) *auto*

lemma *whileIET-rule'*:
fixes *E*
assumes
$(\bigwedge s\ t\ t'.\ I\ s \Longrightarrow\ b\ s \Longrightarrow Some\ 0 \leq lst\ (C\ s)\ (\lambda s'.\ mm3\ (E\ s)\ (if\ I\ s'\ then\ Some\ (E\ s')\ else\ None)))$
$\bigwedge s.\ progress\ (C\ s)$
I s0
shows *Some 0* $\leq$ *lst* (*whileIET I E b C s0*) ($\lambda x.$ *if b x then None else mm3* (*E s0*) (*if I x then Some* (*E x*) *else None*))
by (*rule whileIET-rule*, *rule transf*[**where** *b*=*b*]) (*use assms* **in** *auto*)

13 some Monadic Refinement Automation

ML ‹
structure Refine = struct

structure vcg = Named-Thms
(*val name =* @{*binding refine-vcg*}
val description = Refinement Framework: ^
Verification condition generation rules (*intro*))

structure vcg-cons = Named-Thms
(*val name =* @{*binding refine-vcg-cons*}
val description = Refinement Framework: ^
Consequence rules tried by VCG)

structure refine0 = Named-Thms
(*val name =* @{*binding refine0*}
val description = Refinement Framework: ^
Refinement rules applied first (*intro*))

structure refine = Named-Thms
(*val name =* @{*binding refine*}
val description = Refinement Framework: *Refinement rules* (*intro*))

structure refine2 = Named-Thms
(*val name =* @{*binding refine2*}
val description = Refinement Framework: ^
Refinement rules 2nd stage (*intro*))

(∗ *If set to true, the product splitter of refine-rcg is disabled.* ∗)
val no-prod-split =
Attrib.setup-config-bool @{*binding refine-no-prod-split*} (*K false*);

fun rcg-tac add-thms ctxt =
let
val cons-thms = vcg-cons.get ctxt
val ref-thms = (*refine0.get ctxt*

```
      @ add-thms @ refine.get ctxt @ refine2.get ctxt);
    val prod-ss = (Splitter.add-split @{thm prod.split}
      (put-simpset HOL-basic-ss ctxt));
    val prod-simp-tac =
      if Config.get ctxt no-prod-split then
        K no-tac
      else
        (simp-tac prod-ss THEN'
          REPEAT-ALL-NEW (resolve-tac ctxt @{thms impI allI}));
  in
    REPEAT-ALL-NEW-FWD (DETERM o FIRST' [
      resolve-tac ctxt ref-thms,
      resolve-tac ctxt cons-thms THEN' resolve-tac ctxt ref-thms,
      prod-simp-tac
    ])
  end;

 fun post-tac ctxt = REPEAT-ALL-NEW-FWD (FIRST' [
   eq-assume-tac,
   (*match-tac ctxt thms,*)
   SOLVED' (Tagged-Solver.solve-tac ctxt)])


end;
›
setup ‹Refine.vcg.setup›
setup ‹Refine.vcg-cons.setup›
setup ‹Refine.refine0.setup›
setup ‹Refine.refine.setup›
setup ‹Refine.refine2.setup›


method-setup refine-rcg =
 ‹Attrib.thms >> (fn add-thms => fn ctxt => SIMPLE-METHOD' (
    Refine.rcg-tac add-thms ctxt THEN-ALL-NEW-FWD (TRY o Refine.post-tac
ctxt)
 ))›
 Refinement framework: Generate refinement conditions

method-setup refine-vcg =
 ‹Attrib.thms >> (fn add-thms => fn ctxt => SIMPLE-METHOD' (
    Refine.rcg-tac (add-thms @ Refine.vcg.get ctxt) ctxt THEN-ALL-NEW-FWD
(TRY o Refine.post-tac ctxt)
 ))›
 Refinement framework: Generate refinement and verification conditions

end
theory DataRefinement
imports NREST
```

begin

13.1 Data Refinement

lemma {(*1*,*2*),(*2*,*4*)}``{*1*,*2*}={*2*,*4*} **by** *auto*

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*})
unfolding *conc-fun-def* **by** (*auto split*: *nrest.split*)

lemma *nrest-Rel-mono*: *A* ≤ *B* ⟹ ⇓ *R A* ≤ ⇓ *R B*
unfolding *conc-fun-def* **by** (*fastforce split*: *nrest.split simp*: *le-fun-def intro*!: *Sup-mono*)

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*})
by (*auto simp add*: *le-fun-def conc-fun-def RETURNT-def Rset-def*)

lemma *RETURNT* [*1*,*2*,*3*] ≤ ⇓*Rset* (*RETURNT* {*1*,*2*,*3*})
by (*auto simp add*: *le-fun-def conc-fun-def RETURNT-def Rset-def*)

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

lemma *RETURNT 3* ≤ ⇓*Reven* (*RETURNT False*)
by (*auto simp add*: *le-fun-def conc-fun-def RETURNT-def Reven-def*)

lemma *m* ≤ ⇓*Id m*
unfolding *conc-fun-def RETURNT-def* **by** (*cases m*) *auto*

lemma *m* ≤ ⇓{} *m* ⟷ (*m*=*FAILT* ∨ *m* = *SPECT bot*)
unfolding *conc-fun-def RETURNT-def* **by** (*cases m*) (*auto simp add*: *bot.extremum-uniqueI le-fun-def*)

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
  unfolding abs-fun-def by (auto split: nrest.split)

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 −
  assume (c,a') ∈ R
   with SV have singleton: {m a| a. (c,a) ∈ R} = {m a'} by(auto dest: sin-
gle-valuedD)
  show ?thesis unfolding singleton by simp
qed

lemma indomD:  M c = Some y ⟹ dom M ⊆ Domain R ⟹ (∃ a. (c,a)∈R)
  by auto

lemma conc-abs-swap: m' ≤ ⇓R m ⟷ ⇑R m' ≤ m
proof(cases m; cases m')
  fix M M'
  note [simp] = conc-fun-def abs-fun-def
  assume b[simp]: m = REST M m' = REST M'
  show ?thesis
  proof
    assume a: m' ≤ ⇓ R m
    from a b have R: m' ≤ REST (λc. ⨆ {M a |a. (c, a) ∈ R})
      using conc-fun-RES by metis
    with R have Domain: dom M' ⊆ Domain R
      by (auto simp add: le-fun-def Sup-option-def split: if-splits option.splits)
    from R have  M' x' ≤ M x if (x',x) ∈ R for x x'
      by (auto simp add: Sup-sv[OF that] intro: le-funI dest: le-funD[of - - x'])
    with R SV Domain show ⇑ R m' ≤ m
      by (auto intro!: le-funI simp add: Sup-le-iff)
  next
    assume a: ⇑ R m' ≤ m
    show m' ≤ ⇓R m
    proof(cases dom M' ⊆ Domain R)
      case True

      have M' x ≤ ⨆ {M a |a. (x, a) ∈ R} for x
      proof(cases M' x)
        case (Some y)
        with True obtain x' where x-x': (x,x') ∈ R
          by blast
        with a SV show ?thesis
```

by (*force simp add*: *Sup-sv*[*OF x-x'*] *Sup-le-iff dest*!: *le-funD*[*of - - x'*] *split*: *if-splits*)
qed *auto*
then show $m' \leq \Downarrow R\ m$
by (*auto intro*!: *le-funI*)
next
case *False*
with *a* **show** *?thesis*
by *auto*
qed
qed
qed (*auto simp add*: *conc-fun-def abs-fun-def*)

lemma
fixes $m :: 'b \Rightarrow enat\ option$
shows
Inf-sv: $(c,\ a') \in R \Longrightarrow\ Inf\ \{m\ a|\ a.\ (c,a) \in R\} = m\ a'$
proof −
assume $(c,a') \in R$
with *SV* **have** *singleton*: $\{m\ a|\ a.\ (c,a) \in R\} = \{m\ a'\}$ **by**(*auto dest*: *single-valuedD*)
show *?thesis* **unfolding** *singleton* **by** *simp*
qed

lemma *ac-galois*: *galois-connection* $(\Uparrow R)$ $(\Downarrow R)$
by (*unfold-locales*) (*rule conc-abs-swap*)

lemma
fixes $m :: 'b \Rightarrow enat\ option$
shows *Sup-some-svD*: $Sup\ \{m\ a\ |a.\ (c,\ a) \in R\} = Some\ t' \Longrightarrow (\exists\, a.\ (c,a) \in R \wedge m\ a = Some\ t')$
by (*frule Sup-Some*) (*use SV* **in** ‹*auto simp add*: *single-valued-def Sup-sv*›)

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 (*cases M*)
case *FAILi*
then show *?thesis*
by *auto*
next
case [*simp*]: (*REST m*)
show *?thesis*
proof
assume *?l*

```
    then show ?r
      by (auto simp: abs-fun-def split: if-splits)
  next
    assume a: ?r
    with REST have x∈Domain R if m x = Some y for x y
    proof−
      have enat 0 ≤ y
        by (simp add: enat-0)
      with that REST a show ?thesis
        by auto
    qed
    then show ?l
      by (auto simp: abs-fun-def)
  qed
qed

lemma pw-conc-nofail[refine-pw-simps]:
  nofailT (⇓R S) = nofailT S
  by (cases S) (auto simp: conc-fun-RES)

lemma single-valued A ⟹ single-valued B ⟹ single-valued (A O B)
  by (simp add: single-valued-relcomp)

lemma Sup-enatoption-less2: Sup X = Some ∞ ⟹ (∃ x. Some x ∈ X ∧ enat t
< x)
  using Sup-enat-less2
  by (metis Sup-option-def in-these-eq option.distinct(1) option.sel)

lemma pw-conc-inres[refine-pw-simps]:
  inresT (⇓R S') s t = (nofailT S'
  ⟶ ((∃ s'. (s,s')∈R ∧ inresT S' s' t))) (is ?l ⟷ ?r)
proof(cases S')
  case [simp]: (REST m')
  show ?thesis
  proof
    assume a: ?l
    from this obtain t' where t': enat t ≤ t' ⨆ {m' a |a. (s, a) ∈ R} = Some t'
      by (auto simp: conc-fun-RES)
    then obtain a t'' where (s, a) ∈ R m' a = Some t'' enat t ≤ t''
    proof(cases t')
      case (enat n)
      with t' that[where t''=n] show ?thesis
        by(auto dest: Sup-finite-enat)
    next
      case infinity
      with t' that show ?thesis
        by (force dest!: Sup-enatoption-less2[where t=t] simp add: less-le-not-le
t'(1))
    qed
```

then show *?r*
by *auto*
next
assume *a*: *?r*
then obtain *s' t'* **where** *s'-t'*: $(s,s') \in R$ m' $s' = Some\ t'$ *enat* $t \leq t'$
by (*auto simp*: *conc-fun-RES*)
then have $\exists\, t' \geq enat\ t.\ \bigsqcup \{m'\ a\ |a.\ (s,\ a) \in R\} \geq Some\ t'$
by (*intro exI*[*of* - t']) (*force intro*: *Sup-upper*)
then show *?l*
by (*auto simp*: *conc-fun-RES elim*!: *le-some-optE*)
qed
qed *simp*

lemma *sv-the*: *single-valued* $R \Longrightarrow (c,a) \in R \Longrightarrow (THE\ a.\ (c,\ a) \in R) = a$
by (*simp add*: *single-valued-def the-equality*)

lemma
conc-fun-RES-sv:
assumes *SV*: *single-valued R*
shows $\Downarrow R\ (REST\ X) = REST\ (\lambda c.\ if\ c \in Domain\ R\ then\ X\ (THE\ a.\ (c,a) \in R)\ else\ None\ )$
using *SV* **by** (*auto simp add*: *Sup-sv sv-the Domain-iff conc-fun-def bot-option-def intro*!: *ext split*: *nrest.split*)

lemma *conc-fun-mono*[*simp*, *intro*!]:
shows *mono* ($\Downarrow R$)
by (*rule monoI*) (*auto simp*: *pw-le-iff refine-pw-simps*)

lemma *conc-fun-R-mono*:
assumes $R \subseteq R'$
shows $\Downarrow R\ M \leq \Downarrow R'\ M$
using *assms*
by (*auto simp*: *pw-le-iff refine-pw-simps*)

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 −
have *i*: $\bigwedge a.\ (c,a) \in R\ O\ S \longleftrightarrow (\exists\, b.\ (b,a) \in S \wedge (c,b) \in R)$ **by** *auto*
have $Sup\ \{m\ a\ |a.\ (c,\ a) \in R\ O\ S\} = Sup\ \{m\ a\ |a.\ (\exists\, b.\ (b,a) \in S \wedge (c,b) \in R)\}$
unfolding *i* **by** *auto*
also have ... $= Sup\ \{m\ a\ |a\ b.\ (b,a) \in S \wedge (c,b) \in R\}$ **by** *auto*
finally show *?thesis* **.**
qed

lemma
fixes *m* :: *'a* ⇒ *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* ∧ *Q a c*}
by (*rule antisym*) (*auto intro*: *Sup-least Sup-subset-mono Sup-upper2*)

lemma
fixes *m* :: *'a* ⇒ *enat option*
shows
SupSup-1: *Sup* {*Sup* {*m aa* |*aa*. (*a*, *aa*) ∈ *S*} |*a*. (*c*, *a*) ∈ *R*} = *Sup* {*m aa* |*a aa*. (*a*,*aa*)∈*S* ∧ (*c*,*a*)∈*R*}
by(*rule SupSup*)

lemma *conc-fun-chain*:
shows ⇓*R* (⇓*S M*) = ⇓(*R O S*) *M*
proof(*cases M*)
case [*simp*]: (*REST x*)
have ⨆ {⨆ {*x aa* |*aa*. (*a*, *aa*) ∈ *S*} |*a*. (*c*, *a*) ∈ *R*} = ⨆ {*x a* |*a*. (*c*, *a*) ∈ *R O S*} **for** *c*
by (*unfold SupSup-1 SupSup-2*) *meson*
then show *?thesis*
by (*simp add*: *conc-fun-RES*)
qed *auto*

lemma *conc-fun-chain-sv*:
assumes *SVR*: *single-valued R* **and** *SVS*: *single-valued S*
shows ⇓*R* (⇓*S M*) = ⇓(*R O S*) *M*
using *assms conc-fun-chain* **by** *blast*

lemma *conc-Id*[*simp*]: ⇓*Id* = *id*
unfolding *conc-fun-def* [*abs-def*] **by** (*auto split*: *nrest.split*)

lemma *conc-fun-fail-iff*[*simp*]:
⇓*R S* = *FAILT* ⟷ *S*=*FAILT*
FAILT = ⇓*R S* ⟷ *S*=*FAILT*
by (*auto simp add*: *pw-eq-iff refine-pw-simps*)

lemma *conc-trans*[*trans*]:
assumes *A*: *C* ≤ ⇓*R B* **and** *B*: *B* ≤ ⇓*R' A*
shows *C* ≤ ⇓*R* (⇓*R' A*)
using *assms* **by** (*fastforce simp*: *pw-le-iff refine-pw-simps*)

lemma *conc-trans-sv*:
assumes *SV*: *single-valued R single-valued R'*
and *A*: *C* ≤ ⇓*R B* **and** *B*: *B* ≤ ⇓*R' A*
shows *C* ≤ ⇓*R* (⇓*R' A*)
using *assms* **by** (*fastforce simp*: *pw-le-iff refine-pw-simps*)

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

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

$\bigwedge A\ B\ C.\ A{\leq}{\Downarrow}Id\ B \Longrightarrow B{\leq}{\Downarrow}Id\ C \Longrightarrow A{\leq}\ \ \ C$
$\bigwedge A\ B\ C.\ A{\leq}{\Downarrow}Id\ B \Longrightarrow B{\leq}\ \ \ C \Longrightarrow A{\leq}\ \ \ C$
$\bigwedge A\ B\ C.\ A{\leq}\ \ \ B \Longrightarrow B{\leq}{\Downarrow}Id\ C \Longrightarrow A{\leq}\ \ \ C$
using *assms conc-trans*[**where** $R{=}R$ **and** $R'{=}Id$]
by (*auto intro*: *order-trans*)

lemma *RETURNT-refine*:
assumes $(x,x')\in R$
shows $RETURNT\ x \leq {\Downarrow}R\ (RETURNT\ x')$
using *assms*
by (*auto simp*: *RETURNT-def conc-fun-RES le-fun-def Sup-upper*)

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'* $(\lambda x'.\ f'\ x')$)
using *assms*
by (*simp add*: *pw-le-iff refine-pw-simps*) *blast*

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'* $(\lambda x'.\ f'\ x')$)
apply (*rule bindT-refine'*) **using** *assms* **by** *auto*

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(*cases mono2 body'*)
case *True*
have *flatf-gfp body x* $\leq \Downarrow\ S$ (*flatf-gfp body' x'*)
by (*rule flatf-fixp-transfer*[**where**

```
      fp'=flatf-gfp body
    and B'=body
    and P=λx x'. (x',x)∈R,
    OF - - flatf-ord.fixp-unfold[OF M[THEN trimonoD-flatf-ge]] R0])
    (use True in ‹auto intro: RS dest: trimonoD-flatf-ge›)
  then show ?thesis
    by (auto simp add: M RECT-flat-gfp-def)
qed (auto simp add: RECT-flat-gfp-def)

lemma WHILET-refine:
  assumes R0: (x,x')∈R
  assumes COND-REF: ⋀x x'. ⟦ (x,x')∈R ⟧ ⟹ b x = b' x'
  assumes STEP-REF:
    ⋀x x'. ⟦ (x,x')∈R; b x; b' x' ⟧ ⟹ f x ≤ ⇓R (f' x')
  shows whileT b f x ≤ ⇓R (whileT b' f' x')
  unfolding whileT-def
  apply (rule RECT-refine[OF - R0])
   subgoal by refine-mono
   subgoal by (auto simp add: COND-REF STEP-REF RETURNT-refine intro:
bindT-refine[where R'=R])
  done

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 −
  have n c ≤ ⨆ {m a |a. (c, a) ∈ R} if M = SPECT m for c m
  proof (cases c ∈ dom n)
    case True
    with that a obtain a where k: (c,a)∈R n c ≤ m a by blast
    then show ?thesis
      by (intro Sup-upper2) auto
  next
    case False
    then show ?thesis
      by (simp add: domIff)
  qed
  then show ?thesis
    unfolding conc-fun-def by (auto simp add: le-fun-def split: nrest.split)
qed

lemma SPECT-refines-conc-fun:
  assumes a: ⋀m c. (∃ a. (c,a)∈R ∧ n c ≤ m a)
  shows SPECT n ≤ ⇓ R (SPECT m)
  by (rule SPECT-refines-conc-fun') (use a in auto)


lemma conc-fun-br: ⇓ (br α I1) (SPECT (emb I2 t))
```

$= (SPECT\ (emb\ (\lambda x.\ I1\ x \wedge I2\ (\alpha\ x))\ t))$
unfolding *conc-fun-RES*
using *Sup-Some* **by** (*auto simp*: *emb'-def br-def bot-option-def Sup-option-def*)

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*: $A \leq \Downarrow Id\ B \Longrightarrow A \leq B$ **by** *simp*

13.3 ASSERT

lemma *le-R-ASSERTI*: $(\Phi \Longrightarrow M \leq \Downarrow R\ M') \Longrightarrow M \leq \Downarrow R\ (ASSERT\ \Phi \gg\!= (\lambda\text{-}.\ M'))$
by (*auto simp*: *pw-le-iff refine-pw-simps*)

lemma *T-ASSERT*[*vcg-simp-rules*]: *Some* $t \leq lst\ (ASSERT\ \Phi)\ Q \longleftrightarrow$ *Some* $t \leq Q\ () \wedge \Phi$
by (*cases* Φ) *vcg'*

lemma *T-ASSERT-I*: $(\Phi \Longrightarrow$ *Some* $t \leq Q\ ()) \Longrightarrow \Phi \Longrightarrow$ *Some* $t \leq lst\ (ASSERT\ \Phi)\ Q$
by (*simp add*: *T-ASSERT T-RETURNT*)

lemma *T-RESTemb-iff*: *Some* t'
$\leq lst\ (REST\ (emb'\ P\ t))\ Q \longleftrightarrow (\forall x.\ P\ x \longrightarrow$ *Some* $(t' + t\ x) \leq Q\ x\ )$
by (*auto simp*: *emb'-def T-pw mii-alt aux1*)

lemma *T-RESTemb*: $(\bigwedge x.\ P\ x \Longrightarrow$ *Some* $(t' + t\ x) \leq Q\ x)$
$\Longrightarrow$ *Some* $t' \leq lst\ (REST\ (emb'\ P\ t))\ Q$
by (*auto simp*: *T-RESTemb-iff*)

lemma *T-SPEC*: $(\bigwedge x.\ P\ x \Longrightarrow$ *Some* $(t' + t\ x) \leq Q\ x)$
$\Longrightarrow$ *Some* $t' \leq lst\ (SPEC\ P\ t)\ Q$
unfolding *SPEC-REST-emb'-conv*
by (*auto simp*: *T-RESTemb-iff*)

lemma *T-SPECT-I*: (*Some* $(t' + t\ ) \leq Q\ x)$
$\Longrightarrow$ *Some* $t' \leq lst\ (SPECT\ [\ x \mapsto t])\ Q$
by(*auto simp*: *T-pw mii-alt aux1*)

lemma *mm2-map-option*:
  assumes *Some* $(t'+t) \leq$ *mm2* $(Q\ x)$ $(x2\ x)$
  shows *Some* $t' \leq$ *mm2* $(Q\ x)$ (*map-option* $((+)\ t)$ $(x2\ x)$)
proof(*cases Q x*)
  case *None*
  from *assms None* **show** *?thesis*
    by (*auto simp*: *mm2-def split*: *option.splits if-splits*)
next
  case (*Some a*)
  have *arith*: $\neg\ a < b \Longrightarrow t' + t \leq a - b \Longrightarrow a < t + b \Longrightarrow$ *False* **for** *b*
    by (*cases a*; *cases b*; *cases t'*; *cases t*) *auto*
  moreover have *arith'*: $\neg\ a < b \Longrightarrow t' + t \leq a - b \Longrightarrow \neg\ a < t + b \Longrightarrow t' \leq a - (t + b)$ **for** *b*
    by (*cases a*; *cases b*; *cases t'*; *cases t*) *auto*
  ultimately show *?thesis*
    using *Some assms* **by** (*auto simp*: *mm2-def split*: *option.splits if-splits*)
qed

lemma *T-consume*: (*Some* $(t' + t) \leq$ *lst M Q*)
  $\Longrightarrow$ *Some* $t' \leq$ *lst* (*consume M t*) *Q*
  by (*auto intro*!: *mm2-map-option simp*: *consume-def T-pw mii-alt miiFailt split*: *nrest.splits option.splits if-splits*)

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

13.4 VCG splitter

ML ‹
```
structure VCG-Case-Splitter = struct
  fun dest-case ctxt t =
    case strip-comb t of
      (Const (case-comb, -), args) =>
        (case Ctr-Sugar.ctr-sugar-of-case ctxt case-comb of
           NONE => NONE
         | SOME {case-thms, ...} =>
             let
               val lhs = Thm.prop-of (hd (case-thms))
                 |> HOLogic.dest-Trueprop |> HOLogic.dest-eq |> fst;
               val arity = length (snd (strip-comb lhs));
               (*val conv = funpow (length args - arity) Conv.fun-conv
                 (Conv.rewrs-conv (map mk-meta-eq case-thms));*)
             in
               SOME (nth args (arity - 1), case-thms)
             end)
    | - => NONE;

  fun rewrite-with-asm-tac ctxt k =
    Subgoal.FOCUS (fn {context = ctxt', prems, ...} =>
```

```
      Local-Defs.unfold0-tac ctxt' [nth prems k]) ctxt;

  fun split-term-tac ctxt case-term = (
    case dest-case ctxt case-term of
      NONE => no-tac
    | SOME (arg,case-thms) => let
            val stac = asm-full-simp-tac (ctxt |> put-simpset HOL-basic-ss |>
Simplifier.add-simps case-thms)
        in
        (*CHANGED o stac
        ORELSE'*)
        (
          Induct.cases-tac ctxt false [[SOME arg]] NONE []
          THEN-ALL-NEW stac
        )
      end 1


  )

   fun split-tac ctxt = Subgoal.FOCUS-PARAMS (fn {context = ctxt, ...} =>
ALLGOALS (
      SUBGOAL (fn (t, -) => case Logic.strip-imp-concl t of
        @{mpat Trueprop (Some - ≤ lst ?prog -)} => split-term-tac ctxt prog
      | @{mpat Trueprop (progress ?prog)} => split-term-tac ctxt prog
        | @{mpat Trueprop (Case-Labeling.CTXT - - - (valid - - ?prog))} =>
split-term-tac ctxt prog
      | - => no-tac
      ))
    ) ctxt
    THEN-ALL-NEW TRY o Hypsubst.hyp-subst-tac ctxt

  end
›

method-setup vcg-split-case =
 ‹Scan.succeed (fn ctxt => SIMPLE-METHOD' (CHANGED o (VCG-Case-Splitter.split-tac
ctxt)))›
```

13.5 mm3 and emb

```
lemma Some-eq-mm3-Some-conv[vcg-simp-rules]: Some t = mm3 t' (Some t'')
⟷ (t'' ≤ t' ∧ t = enat (t' − t''))
  by (simp add: mm3-def)

lemma Some-eq-mm3-Some-conv': mm3 t' (Some t'') = Some t ⟷ (t'' ≤ t' ∧ t
= enat (t' − t''))
  using Some-eq-mm3-Some-conv by metis
```

lemma *Some-le-emb'-conv*[*vcg-simp-rules*]: *Some* $t \leq$ *emb'* Q *ft* $x \longleftrightarrow Q\ x \wedge t \leq$ *ft* x
by (*auto simp*: *emb'-def*)

lemma *Some-eq-emb'-conv*[*vcg-simp-rules*]: *emb'* Q *tf* $s =$ *Some* $t \longleftrightarrow (Q\ s \wedge t =$ *tf* $s)$
unfolding *emb'-def* **by**(*auto split*: *if-splits*)

13.6 Setup Labeled VCG

context
begin
interpretation *Labeling-Syntax* **.**

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 $C\langle IC, CT, OC: valid\ t\ Q\ (if\ b\ then\ c1\ else\ c2)\rangle$
using *assms*(2−) **unfolding** *LABEL-simps* **by** (*simp add*: *valid-def*)

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 $C\langle IC, CT, OC: t \leq (if\ b\ then\ A\ else\ B)\rangle$
using *assms*(2−) **unfolding** *LABEL-simps* **by** (*simp add*: *valid-def*)

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* $t \leq lst\ (whileIET\ I\ E\ b\ C\ s0)\ Q$
apply(*rule whileIET-rule'*[*THEN T-conseq4*])
subgoal using *assms*(2) **by** *simp*
subgoal using *assms*(3) **by** *simp*
subgoal using *assms*(1) **by** *simp*
subgoal for x **using** *assms*(4) **by** (*cases I x*) (*auto simp*: *Some-eq-mm3-Some-conv'* *split*: *if-splits*)
done

definition *monadic-WHILE* $b\ f\ s \equiv do\ \{$

```
  RECT (λD s. do {
    bv ← b s;
    if bv then do {
      s ← f s;
      D s
    } else do {RETURNT s}
  }) s
}

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)
  unfolding monadic-WHILE-def apply(rule RECT-mono)
   subgoal by (refine-mono)
  using assms by (auto intro!: bindT-mono)

lemma z: inresT A x t ⟹ A ≤ B ⟹ inresT B x t
  by (meson fail-inresT pw-le-iff)

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)
  unfolding monadic-WHILE-def apply(rule RECT-mono)
  subgoal by(refine-mono)
  apply(rule bindT-mono)
   apply fact
  using assms by (auto intro!: bindT-mono dest: z[OF - assms(1)])

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')
  unfolding monadic-WHILE-def apply(rule RECT-refine[OF - assms(1)])
  subgoal by(refine-mono)
  apply(rule bindT-refine'[OF assms(2)])
   subgoal by auto
  by (auto intro!: assms(3) bindT-refine RETURNT-refine)

lemma monadic-WHILE-aux: monadic-WHILE b f s = monadic-WHILEIT (λ-.
True) b f s
  unfolding monadic-WHILEIT-def monadic-WHILE-def
  by simp
```

— No proof
lemma *lst* (*c x*) *Q* = *Some t* ⟹ *Some t* ≤ *lst* (*c x*) *Q′*
apply(*rule T-conseq6*) **oops**

lemma *TbindT-I2*: *tt* ≤ *lst M* (λ*y*. *lst* (*f y*) *Q*) ⟹ *tt* ≤ *lst* (*M* ⤜ *f*) *Q*
by (*simp add*: *T-bindT*)

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*
using *assms* **by** (*cases tt*) (*auto intro*: *T-conseq6*)

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*
using *assms*(*1*)
unfolding *monadic-WHILE-def*
proof (*induction rule*: *RECT-wf-induct*[**where** *R*=*R*])
case *1*
show *?case* **by** *fact*
next
case *2*
then show *?case* **by** *refine-mono*
next
case *step*: (*3 x D r*)
note *IH*[*vcg-rules*] = *step.IH*[*OF - refl*]
note *step.hyps*[*symmetric*, *simp*]
from *step.prems*
show *?case*
apply *simp*
apply (*rule TbindT-I2*)
apply(*rule T-conseq7*)
apply (*rule IS*)
apply *simp*
apply *safe*
subgoal
apply (*rule TbindT-I*)
apply(*rule T-conseq6*[**where** *Q′*=(λ*s′*. *if* (*s′*, *x*) ∈ *R then I s′ else None*)])
subgoal by *simp*
by (*auto split*: *if-splits dest*: *IH*)
subgoal by(*simp add*: *T-RETURNT*)

```
    done
qed

lemma monadic-WHILE-rule'':
  assumes monadic-WHILE bm c s = r
 assumes IS[vcg-rules]: ⋀s t'. I s = Some t'
           ⟹ lst (bm s) (λb. if b then lst (c s)  (λs'. 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
  using assms(1,3)
  unfolding monadic-WHILE-def
proof (induction arbitrary: t rule: RECT-wf-induct[where R=R])
  case 1
  show ?case by fact
next
  case 2
  then show ?case by refine-mono
next
  case step: (3 x D r t')
  note IH[vcg-rules] = step.IH[OF - refl]
  note step.hyps[symmetric, simp]

  from step.prems
  show ?case
    apply simp
    apply (rule TbindT-I)
    apply(rule T-conseq6)
     apply (rule IS) apply simp
    apply simp
    apply safe
    subgoal
      apply (rule TbindT-I)
      apply(rule T-conseq6[where Q'=(λs'. if (s', x) ∈ R then I s' else None)])
       subgoal by simp
      by (auto split: if-splits intro: IH)
    subgoal by vcg'
    done
qed

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) (λs'. 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*
apply(*rule T-conseq4*)
apply(*rule whileT-rule''*[**where** *I*=λ*s*. *mm3 t0* (*I s*)
and *R*=*measure* (*the-enat o the o I*), *OF assms*(*1*)])
subgoal for *s t'*
apply(*cases I s*; *simp*)
subgoal for *ti*
using *IS*[*of s ti*]
apply (*cases c s*) **apply**(*simp*)
subgoal for *M*
using *progress*[*of s*, *THEN progressD*, *of M*]
— TODO: Cleanup
apply(*auto simp*: *T-pw mm3-Some-conv mii-alt mm2-def mm3-def split*: *option.splits if-splits*)
apply *fastforce*
subgoal
by (*metis enat-ord-simps*(*1*) *le-diff-iff le-less-trans option.distinct*(*1*))
subgoal
by (*metis diff-is-0-eq' leI less-option-Some option.simps*(*3*) *zero-enat-def*)

subgoal
by (*smt* (*z3*) *Nat.add-diff-assoc enat-ile enat-ord-code*(*1*) *idiff-enat-enat leI le-add-diff-inverse2 nat-le-iff-add option.simps*(*3*))
subgoal
using *dual-order.trans* **by** *blast*
done
done
done
by *auto*

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

lemma *l*: ~ (*a*::*enat*) < *b* ⟷ *a* ≥ *b* **by** *auto*

lemma *kk*: *a*≥*b* ⟹ (*b*::*enat*) + (*a* − *b*) = *a*
by (*cases a*; *cases b*) *auto*

lemma *Tea*: *Someplus A B = Some t* ⟷ (∃ *a b*. *A = Some a* ∧ *B = Some b* ∧ *t = a + b*)
by (*cases A*; *cases B*) *auto*

lemma *TTT-Some-nofailT*: *lst c Q = Some l* ⟹ *c* ≠ *FAILT*
unfolding *lst-def mii-alt* **by** *auto*

lemma *GRR*: **assumes** *lst* (*SPECT Mf*) *Q* = *Some l*
shows *Mf x* = *None* ∨ (*Q x*≠ *None* ∧ (*Q x*) ≥ *Mf x*)
proof −
from *assms* **have** *None* ∉ {*mii Q* (*SPECT Mf*) *x* |*x*. *True*}
unfolding *lst-def*
unfolding *Inf-option-def* **by** (*auto split*: *if-splits*)
then have *None* ≠ (*case Mf x of None* ⇒ *Some* ∞
| *Some mt* ⇒ *case Q x of None* ⇒ *None*
| *Some rt* ⇒ *if rt* < *mt then None else Some* (*rt* − *mt*))
unfolding *mii-alt mm2-def*
by (*auto*)
then show *?thesis* **by** (*auto split*: *option.splits if-splits*)
qed

lemma *Someplus-None*: *Someplus A B* = *None* ⟷ (*A* = *None* ∨ *B* = *None*)
by (*cases A*; *cases B*) *auto*

lemma *Somemm3*: *A* ≥ *B* ⟹ *mm3 A* (*Some B*) = *Some* (*A* − *B*)
unfolding *mm3-def* **by** *auto*

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*) (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 *I0*: *I s0*
shows *Some t* ≤ *lst r Q*
proof −
— TODO: Cleanup, will be work
show *Some t* ≤ *lst r Q*
apply (*rule monadic-WHILE-rule*′′[**where** *I*=λ*s*. *Someplus* (*Some t*) (*mm3* (*E s0*) ((λ*e*. *if I e*
then Some (*E e*) *else None*) *s*)) **and** *R*=*measure* (*the-enat o the o* (λ*e*. *if I e*
then Some (*E e*) *else None*)), *simplified*])
apply *fact*
subgoal for *s t*′
apply(*auto split*: *if-splits*)
apply(*rule T-conseq4*)
apply(*rule step*)
apply *simp*
apply *auto*
proof (*goal-cases*)
case (*1 b t*′′)
from *1*(*3*) *TTT-Some-nofailT* **obtain** *M* **where** *cs*: *c s* = *SPECT M* **by**

force
{ **assume** *A*: $\bigwedge x.\ M\ x = None$
with *A* **have** *?case* **unfolding** *cs lst-def mii-alt* **by** *simp*
}
moreover
{ **assume** $\exists\, x.\ M\ x \neq None$
then obtain *x* **where** *i*: $M\ x \neq None$ **by** *blast*

let *?T* = *lst* (*c s*) ($\lambda s'.$ *if I s'* $\wedge$ *E s'* $\leq$ *E s then Some* (*enat* (*E s* − *E s'*)) *else None*)

from *GRR*[*OF 1*(*3*)[*unfolded cs*], **where** *x*=*x*]
i **have** (*if I x* $\wedge$ *E x* $\leq$ *E s then Some* (*enat* (*E s* − *E x*)) *else None*) $\neq$ *None* $\wedge$ *M x* $\leq$ (*if I x* $\wedge$ *E x* $\leq$ *E s then Some* (*enat* (*E s* − *E x*)) *else None*)
by *simp*
then have *pf*: *I x E x* $\leq$ *E s M x* $\leq$ *Some* (*enat* (*E s* − *E x*)) **by** (*auto split*: *if-splits*)

then have *M x* < *Some* ∞
using *enat-ord-code*(*4*) *le-less-trans less-option-Some* **by** *blast*

have *Some t''* = *?T* **using** *1*(*3*) **by** *simp*
also have *oo*: *?T* $\leq$ *mm2* (*Some* (*enat* (*E s* − *E x*))) (*M x*)
unfolding *lst-def* **apply**(*rule Inf-lower*) **apply** (*simp add*: *mii-alt cs*) **apply**(*rule exI*[**where** *x*=*x*])
using *pf* **by** *simp*

also from *i* **have** *o*: ... < *Some* ∞ **unfolding** *mm2-def*
apply *auto*
using *fl* **by** *blast*
finally have *tni*: *t''* < ∞ **by** *auto*
then have *tt*: *t'* + *t''* − *t''* = *t'* **apply**(*cases t''*; *cases t'*) **by** *auto*

have *ka*: $\bigwedge x.$ *mii* ($\lambda s'.$ *if I s'* $\wedge$ *E s'* $\leq$ *E s then Some* (*enat* (*E s* − *E s'*)) *else None*) (*c s*) *x* $\geq$ *Some t''* **unfolding** *lst-def*
using *1*(*3*) *T-pw* **by** *fastforce*

{ **fix** *x* **assume** *nN*: *M x* $\neq$ *None*
with *progress*[*of s*, *unfolded cs progress-def*] **have** *strict*: *Some 0* < *M x* **by** *auto*
from *ka*[*of x*] *nN* **have** *E x* < *E s* **unfolding** *mii-alt cs progress-def mm2-def*
using *strict less-le zero-enat-def* **by** (*fastforce simp*: *l split*: *if-splits*)
} **note** *strict* = *this*
have *?case*
— TODO: Cleanup
apply(*rule T-conseq5*[**where** *Q'*=($\lambda s'.$ *if I s'* $\wedge$ *E s'* $\leq$ *E s then Some* (*enat* (*E s* − *E s'*)) *else None*)])

```
      using 1(3) apply(auto) [] using 1(2)
      apply (auto simp add: tt  Tea split: if-splits)
      subgoal apply(auto simp add: Some-eq-mm3-Some-conv')
        apply(rule strict) using cs by simp
      subgoal by(simp add: Some-eq-mm3-Some-conv' Somemm3)
      done
  }
  ultimately show ?case by blast
  next
    case (2 x t'')
   then show ?case unfolding mm3-def mm2-def by (auto simp add: l kk split:
if-splits option.splits)
  qed
  subgoal
    using I0 by simp
  done
qed

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)  (λb. 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  i: I s0
shows Some t ≤ lst (monadic-WHILEIE I E bm c s0) Q
  unfolding monadic-WHILEIE-def
  apply(rule neueWhile-rule[OF refl]) by fact+

lemma neueWhile-rule'':
  fixes s0 :: 'a and I :: 'a ⇒ bool and E :: 'a ⇒ nat
  assumes
  step: (⋀s. I s ⟹ Some 0 ≤ lst (bm s) (λb. if b then lst (c s) (λs'. 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
   unfolding monadic-WHILEIE-def apply(rule neueWhile-rule[OF refl, where
I=I and E=E ])
     using assms unfolding G-def H-def by auto
```

lemma *LmonWhileRule*:
fixes *IC CT*
assumes $V\langle(''precondition'', IC, []),(''monwhile'', IC, []) \# CT: I\ s0\rangle$
and $\bigwedge s.\ I\ s \Longrightarrow C\langle Suc\ IC,(''invariant'', Suc\ IC, []) \# (''monwhile'', IC, []) \# CT,OC: valid\ 0\ (\lambda b.\ if\ b\ then\ lst\ (C\ s)\ (\lambda s'.\ if\ I\ s' \wedge E\ s' \leq E\ s\ then\ Some\ (enat\ (E\ s - E\ s'))\ else\ None)\ else\ mm2\ (Q\ s)\ (Someplus\ (Some\ t)\ (mm3\ (E\ s0)\ (Some\ (E\ s)))))\ (bm\ s)\rangle$
and $\bigwedge s.\ V\langle(''progress'', IC, []),(''monwhile'', IC, []) \# CT: progress\ (C\ s)\rangle$
shows $C\langle IC,CT,OC: valid\ t\ Q\ (monadic\text{-}WHILEIE\ I\ E\ bm\ C\ s0)\rangle$
using *assms*(*2*,*3*,*1*) **unfolding** *valid-def LABEL-simps*
by (*rule neueWhile-rule'*)

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

lemma *validD*: $valid\ t\ Q\ M \Longrightarrow Some\ t \leq lst\ M\ Q$ **by**(*simp add*: *valid-def*)

lemma *LABELs-to-concl*:
$C\langle IC,\ CT,\ OC: True\rangle \Longrightarrow C\langle IC,\ CT,\ OC: P\rangle \Longrightarrow P$
$V\langle x,\ ct: True\rangle \Longrightarrow V\langle x,\ ct: P\rangle \Longrightarrow P$
unfolding *LABEL-simps* **.**

lemma *LASSERTRule*:
assumes $V\langle(''ASSERT'', IC, []),CT: \Phi\rangle$
$C\langle Suc\ IC,\ CT,OC: valid\ t\ Q\ (RETURNT\ ())\rangle$
shows $C\langle IC,CT,OC: valid\ t\ Q\ (ASSERT\ \Phi)\rangle$
using *assms* **unfolding** *LABEL-simps* **by** (*simp add*: *valid-def*)

lemma *LbindTRule*:
assumes $C\langle IC,CT,OC: valid\ t\ (\lambda y.\ lst\ (f\ y)\ Q)\ m\rangle$
shows $C\langle IC,CT,OC: valid\ t\ Q\ (bindT\ m\ f)\rangle$
using *assms* **unfolding** *LABEL-simps* **by**(*simp add*: *T-bindT valid-def*)

lemma *LRETURNTRule*:
assumes $C\langle IC,CT,OC: Some\ t \leq Q\ x\rangle$
shows $C\langle IC,CT,OC: valid\ t\ Q\ (RETURNT\ x)\rangle$
using *assms* **unfolding** *LABEL-simps* **by** (*simp add*: *valid-def T-RETURNT*)

lemma *LSELECTRule*:

fixes *IC CT* **defines** $CT' \equiv (''cond'', IC, []) \# CT$
assumes $\bigwedge x.\ P\ x \Longrightarrow C\langle Suc\ IC,(''Some'', IC, []) \# (''SELECT'', IC, []) \# CT,OC1\text{: } valid\ (t+T)\ Q\ (RETURNT\ (Some\ x))\rangle$
and $\forall x.\ \neg\ P\ x \Longrightarrow C\langle Suc\ OC1,(''None'', Suc\ OC1, []) \# (''SELECT'', IC, []) \# CT,OC\text{: } valid\ (t+T)\ Q\ (RETURNT\ None)\rangle$
shows $C\langle IC,CT,OC\text{: } valid\ t\ Q\ (SELECT\ P\ T)\rangle$
using *assms*(2−) **unfolding** *LABEL-simps* **by**(*auto intro*: *T-SELECT T-SPECT-I simp add*: *valid-def T-RETURNT*)

lemma *LRESTembRule*:
assumes $\bigwedge x.\ P\ x \Longrightarrow C\langle IC,CT,OC\text{: } Some\ (t\ +\ T\ x) \leq Q\ x\rangle$
shows $C\langle IC,CT,OC\text{: } valid\ t\ Q\ (REST\ (emb'\ P\ T))\rangle$
using *assms* **unfolding** *LABEL-simps* **by** (*simp add*: *valid-def T-RESTemb*)

lemma *LRESTsingleRule*:
assumes $C\langle IC,CT,OC\text{: } Some\ (t\ +\ t') \leq Q\ x\rangle$
shows $C\langle IC,CT,OC\text{: } valid\ t\ Q\ (REST\ [x\mapsto t'])\rangle$
using *assms* **unfolding** *LABEL-simps* **by** (*simp add*: *valid-def T-pw mii-alt aux1*)

lemma *LTTTinRule*:
assumes $C\langle IC,CT,OC\text{: } valid\ t\ Q\ M\rangle$
shows $C\langle IC,CT,OC\text{: } Some\ t \leq lst\ M\ Q\rangle$
using *assms* **unfolding** *LABEL-simps* **by**(*simp add*: *valid-def*)

lemma *LfinaltimeRule*:
assumes $V\langle(''time'', IC, []),\ CT\text{: } t \leq t'\ \rangle$
shows $C\langle IC,CT,IC\text{: } Some\ t \leq Some\ t'\rangle$
using *assms* **unfolding** *LABEL-simps* **by** *simp*

lemma *LfinalfuncRule*:
assumes $V\langle(''func'', IC, []),\ CT\text{: } False\ \rangle$
shows $C\langle IC,CT,IC\text{: } Some\ t \leq None\rangle$
using *assms* **unfolding** *LABEL-simps* **by** *simp*

lemma *LembRule*:
assumes $V\langle(''time'', IC, []),\ CT\text{: } t \leq T\ x\ \rangle$
and $V\langle(''func'', IC, []),\ CT\text{: } P\ x\ \rangle$
shows $C\langle IC,CT,IC\text{: } Some\ t \leq emb'\ P\ T\ x\rangle$
using *assms* **unfolding** *LABEL-simps* **by** (*simp add*: *emb'-def*)

lemma *Lmm3Rule*:
assumes $V\langle(''time'', IC, []),\ CT\text{: } Va' \leq Va \wedge t \leq enat\ (Va\ -\ Va')\ \rangle$
and $V\langle(''func'', IC, []),\ CT\text{: } b\ \rangle$
shows $C\langle IC,CT,IC\text{: } Some\ t \leq mm3\ Va\ (if\ b\ then\ Some\ Va'\ else\ None)\ \rangle$
using *assms* **unfolding** *LABEL-simps* **by** (*simp add*: *mm3-def*)

lemma *LinjectRule*:

assumes *Some t* ≤ *lst A Q* ⟹ *Some t* ≤ *lst B Q*
C⟨*IC*,*CT*,*OC*: *valid t Q A*⟩
shows *C*⟨*IC*,*CT*,*OC*: *valid t Q B*⟩
using *assms* **unfolding** *LABEL-simps* **by** (*simp add*: *valid-def*)

lemma *Linject2Rule*:
assumes *A* = *B*
C⟨*IC*,*CT*,*OC*: *valid t Q A*⟩
shows *C*⟨*IC*,*CT*,*OC*: *valid t Q B*⟩
using *assms* **unfolding** *LABEL-simps* **by** (*simp add*: *valid-def*)

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'*)
apply *labeled-VCG* **oops**

lemma
assumes *b c*
shows *do* { *ASSERT b*;
ASSERT c;
RETURNT 1 } ≤ *SPECT* (*emb* (λ*x*. *x*>(*0*::*nat*)) *1*)
proof (*labeled-VCG*, *casify*)
case *ASSERT* **then show** *?case* **by** *fact*
case *ASSERTa* **then show** *?case* **by** *fact*
case *func* **then show** *?case* **by** *simp*
case *time* **then show** *?case* **by** *simp*
qed

lemma *do* {
(*if b then RETURNT* (*1*::*nat*) *else RETURNT 2*)

```
    } ≤ SPECT (emb (λx. x>0) 1)
proof (labeled-VCG, casify)
  case cond {
    case the {
      case time
      then show ?case by simp
    next
      case func
      then show ?case by simp
    }
  next
    case els {
      case func
      then show ?case by simp
    }
  }
qed simp


lemma
  assumes b
  shows do {
      ASSERT b;
      (if b then RETURNT (1::nat) else RETURNT 2)
    } ≤ SPECT (emb (λx. x>0) 1)
proof (labeled-VCG, casify)
  case ASSERT then show ?case by fact
  case cond {
    case the {
      case time
      then show ?case by simp
    next
      case func
      then show ?case by simp
    }
  next
    case els {
      case func
      then show ?case by simp
    }
  }
qed simp

end



lemma SPECT-ub': T≤T' ⟹ SPECT (emb' M' T) ≤ ⇓Id (SPECT (emb' M'
```

T'))
unfolding *emb'-def* **by** (*auto simp*: *pw-le-iff le-funD order-trans refine-pw-simps*)

lemma *REST-single-rule*[*vcg-simp-rules*]: *Some t* ≤ *lst* (*REST* [*x*↦*t'*]) *Q* ⟷ *Some* (*t*+*t'*) ≤ (*Q x*)
by (*simp add*: *T-REST aux1*)

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−
have *P x* ≤ ⨆ {*Q a* |*a*. (*x*, *a*) ∈ *R*} **for** *x*
proof(*cases P x*)
case [*simp*]: (*Some y*)
from *assms*[*of x*, *OF Some*] **obtain** *s'* **where** *s'*: *Some y* ≤ *Q s'* (*x*, *s'*) ∈ *R*
by *blast*+
show *?thesis*
by (*rule order.trans*[**where** *b*=*Q s'*]) (*auto intro*: *s' Sup-upper*)
qed *auto*
then show *?thesis*
by (*auto simp add*: *conc-fun-def le-fun-def*)
qed

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''*))
by (*simp add*: *mm3-def*)

lemma *embtimeI*: *T* ≤ *T'* ⟹ *emb P T* ≤ *emb P T'* **unfolding** *emb'-def* **by** (*auto simp*: *le-fun-def split*: *if-splits*)

lemma *not-cons-is-Nil-conv*[*simp*]: (∀*y ys*. *l* ≠ *y* # *ys*) ⟷ *l*=[] **by** (*cases l*) *auto*

lemma *mm3-Some0-eq*[*simp*]: *mm3 t* (*Some 0*) = *Some t*
by (*auto simp*: *mm3-def*)

lemma *ran-emb'*: $c \in ran\ (emb'\ Q\ t) \longleftrightarrow (\exists\, s'.\ Q\ s' \wedge t\ s' = c)$
by(*auto simp*: *emb'-def ran-def*)

lemma *ran-emb-conv*: $Ex\ Q \Longrightarrow ran\ (emb\ Q\ t) = \{t\}$
by (*auto simp*: *ran-emb'*)

lemma *in-ran-emb-special-case*: $c \in ran\ (emb\ Q\ t) \Longrightarrow c \leq t$
apply (*cases Ex Q*)
subgoal by (*auto simp*: *ran-emb-conv*)
subgoal by (*auto simp*: *emb'-def*)
done

lemma *dom-emb'-eq*[*simp*]: $dom\ (emb'\ Q\ f) = Collect\ Q$
by(*auto simp*: *emb'-def split*: *if-splits*)

lemma *emb-le-emb*: $emb'\ P\ T \leq emb'\ P'\ T' \longleftrightarrow (\forall\, x.\ P\ x \longrightarrow P'\ x \wedge\ T\ x \leq T'\ x)$
unfolding *emb'-def* **by** (*auto simp*: *le-fun-def split*: *if-splits*)

13.10 VCG for monadic programs

13.10.1 old

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

lemma *SS*[*vcg-simp-rules*]: $Some\ t = Some\ t' \longleftrightarrow t = t'$ **by** *simp*
lemma *SS'*: $(if\ b\ then\ Some\ t\ else\ None) = Some\ t' \longleftrightarrow (b \wedge t = t')$ **by** *simp*

term (*case s of* (*a*,*b*) ⇒ *M a b*)
lemma *case-T*[*vcg-rules*]: $(\bigwedge a\ b.\ s = (a,\ b) \Longrightarrow t \leq lst\ Q\ (M\ a\ b)) \Longrightarrow t\ \leq lst\ Q\ (case\ s\ of\ (a,b) \Rightarrow M\ a\ b)$
by (*simp add*: *split-def*)

13.10.2 new setup

named-theorems *vcg-rules'*
lemma *if-T*[*vcg-rules'*]: $(b \Longrightarrow t \leq lst\ Ma\ Q) \Longrightarrow (\neg b \Longrightarrow t \leq lst\ Mb\ Q) \Longrightarrow t \leq lst\ (if\ b\ then\ Ma\ else\ Mb)\ Q$
by (*simp add*: *split-def*)
lemma *RETURNT-T-I*[*vcg-rules'*]: $t \leq Q\ x \Longrightarrow t\ \leq lst\ (RETURNT\ x)\ Q$
by (*simp add*: *T-RETURNT*)

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*)
apply(*rule T-specifies-I*)
apply(*vcg'*‹−›) **unfolding** *option.case*
oops

thm *option.case*

13.11 setup for *refine-vcg*

lemma *If-refine*[*refine*]: *b* = *b'* ⟹
(*b* ⟹ *b'* ⟹ *S1* ≤ ⇓ *R S1'*) ⟹
(¬ *b* ⟹ ¬ *b'* ⟹ *S2* ≤ ⇓ *R S2'*) ⟹ (*if b then S1 else S2*) ≤ ⇓ *R* (*if b' then S1' else S2'*)
by *auto*

lemma *Case-option-refine*[*refine*]: (*x*,*x'*)∈ ⟨*S*⟩*option-rel* ⟹
(⋀*y y'*. (*y*,*y'*)∈*S* ⟹ *S2 y* ≤ ⇓ *R* (*S2' y'*)) ⟹ *S1* ≤ ⇓ *R S1'*
⟹ (*case x of None* ⇒ *S1* | *Some y* ⇒ *S2 y*) ≤ ⇓ *R* (*case x' of None* ⇒ *S1'* | *Some y'* ⇒ *S2' y'*)
by(*auto split*: *option.split*)

— Check names
lemma *conc-fun-Id-refined*[*refine0*]: ⋀*S*. *S* ≤ ⇓ *Id S* **by** *simp*
lemma *conc-fun-ASSERT-refine*[*refine0*]: Φ ⟹ (Φ ⟹ *S* ≤ ⇓ *R S'*) ⟹ *ASSERT* Φ ≫= (λ-. *S*) ≤ ⇓ *R S'*
by *auto*
declare *le-R-ASSERTI* [*refine0*]

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

— Check name
lemma *SPECT-refine-vcg-cons*[*refine-vcg-cons*]: *m* ≤ *SPECT* Φ ⟹ (⋀*x*. Φ *x* ≤ Ψ *x*) ⟹ *m* ≤ *SPECT* Ψ

by (*metis dual-order.trans le-funI nres-order-simps*(*2*))

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

13.12 Relators

definition *nrest-rel* **where**
nres-rel-def-internal: *nrest-rel R* ≡ {(*c*,*a*). *c* ≤ ⇓*R a*}

lemma *nrest-rel-def*: ⟨*R*⟩*nrest-rel* ≡ {(*c*,*a*). *c* ≤ ⇓*R a*}
by (*simp add*: *nres-rel-def-internal relAPP-def*)

lemma *nrest-relD*: (*c*,*a*)∈⟨*R*⟩*nrest-rel* ⟹ *c* ≤⇓*R a* **by** (*simp add*: *nrest-rel-def*)
lemma *nrest-relI*: *c* ≤⇓*R a* ⟹ (*c*,*a*)∈⟨*R*⟩*nrest-rel* **by** (*simp add*: *nrest-rel-def*)

lemma *nrest-rel-comp*: ⟨*A*⟩*nrest-rel O* ⟨*B*⟩*nrest-rel* = ⟨*A O B*⟩*nrest-rel*
by (*auto simp*: *nrest-rel-def conc-fun-chain*[*symmetric*] *conc-trans*)

lemma *pw-nrest-rel-iff*: (*a*,*b*)∈⟨*A*⟩*nrest-rel* ⟷ *nofailT* (⇓ *A b*) ⟶ *nofailT a* ∧
(∀ *x t*. *inresT a x t* ⟶ *inresT* (⇓ *A b*) *x t*)
by (*simp add*: *pw-le-iff nrest-rel-def*)

lemma *param-FAIL*[*param*]: (*FAILT*,*FAILT*) ∈ ⟨*R*⟩*nrest-rel*
by (*auto simp*: *nrest-rel-def*)

lemma *param-RETURN*[*param*]:
(*RETURNT*,*RETURNT*) ∈ *R* → ⟨*R*⟩*nrest-rel*
by (*auto simp*: *nrest-rel-def RETURNT-refine*)

lemma *param-bind*[*param*]:
(*bindT*,*bindT*) ∈ ⟨*Ra*⟩*nrest-rel* → (*Ra*→⟨*Rb*⟩*nrest-rel*) → ⟨*Rb*⟩*nrest-rel*
by (*auto simp*: *nrest-rel-def intro*: *bindT-refine dest*: *fun-relD*)

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*:
$it \subseteq S \Longrightarrow x \in it \Longrightarrow S - (it - \{x\}) = insert\ x\ (S - it)$ **by** *auto*

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* ($\langle FOREACH_{OC}{}^{-,-}\rangle$) **where** *FOREACHoci R* Φ *S c f* $\sigma 0$ *inittime body-time* $\equiv$ *do* {
ASSERT (*finite S*);
xs $\leftarrow$ *SPECT* (*emb* ($\lambda xs.$ *distinct xs* $\wedge$ *S* = *set xs* $\wedge$ *sorted-wrt R xs*) *inittime*);
(-,σ) $\leftarrow$ *whileIET*
($\lambda(it,\sigma).\ \exists xs'.$ *xs* = *xs'* @ *it* $\wedge$ Φ (*set it*) σ) ($\lambda(it,-).$ *length it* $*$ *body-time*) (*FOREACH-cond c*) (*FOREACH-body f*) (*xs*,$\sigma 0$);
RETURNT σ }

Foreach with continuation condition and annotated invariant:

definition *FOREACHci* ($\langle FOREACH_{C}{}^{-}\rangle$) **where** *FOREACHci* $\equiv$ *FOREACHoci* (λ- -. *True*)

13.15 Proof Rules

lemma *FOREACHoci-rule*:
assumes *IP*:
$\bigwedge x\ it\ \sigma.\ [\![\ c\ \sigma;\ x \in it;\ it \subseteq S;\ I\ it\ \sigma;\ \forall y \in it - \{x\}.\ R\ x\ y;$
$\forall y \in S - it.\ R\ y\ x\ ]\!] \Longrightarrow f\ x\ \sigma \leq SPECT\ (emb\ (I\ (it - \{x\}))\ (enat\ body\text{-}time))$
assumes *FIN*: *finite S*
assumes *I0*: *I S* $\sigma 0$
assumes *II1*: $\bigwedge \sigma.\ [\![I\ \{\}\ \sigma]\!] \Longrightarrow P\ \sigma$
assumes *II2*: $\bigwedge it\ \sigma.\ [\![\ it \neq \{\};\ it \subseteq S;\ I\ it\ \sigma;\ \neg c\ \sigma;$
$\forall x \in it.\ \forall y \in S - it.\ R\ y\ x\ ]\!] \Longrightarrow P\ \sigma$
assumes *time-ub*: *inittime* + *enat* ((*card S*) $*$ *body-time*) $\leq$ *enat overall-time*

assumes *progress-f*[*progress-rules*]: $\bigwedge a\ b.\ progress\ (f\ a\ b)$
shows *FOREACHoci R I S c f σ0 inittime body-time* ≤ *SPECT* (*emb P* (*enat overall-time*))
unfolding *FOREACHoci-def*
apply(*rule T-specifies-I*)
unfolding *FOREACH-body-def FOREACH-cond-def*
apply(*vcg'*‹−› *rules*: *IP*[*THEN T-specifies-rev*, *THEN T-conseq4*])

prefer *5* **apply** *auto* []
subgoal using *I0* **by** *blast*
subgoal by *blast*
subgoal by *simp*
subgoal by *auto*
subgoal by (*metis distinct-append hd-Cons-tl remove1-tl set-remove1-eq sorted-wrt.simps*(*2*) *sorted-wrt-append*)
subgoal by (*metis DiffD1 DiffD2 UnE list.set-sel*(*1*) *set-append sorted-wrt-append*)

subgoal apply (*auto simp*: *Some-le-mm3-Some-conv Some-le-emb'-conv Some-eq-emb'-conv diff-mult-distrib*)
subgoal by (*metis append.assoc append.simps*(*2*) *append-Nil hd-Cons-tl*)
subgoal by (*metis remove1-tl set-remove1-eq*)
done
subgoal using *time-ub II1* **apply** (*auto simp*: *Some-le-mm3-Some-conv Some-le-emb'-conv Some-eq-emb'-conv distinct-card*)
subgoal by (*metis DiffD1 DiffD2 II2 Un-iff Un-upper2 sorted-wrt-append*)
subgoal by (*metis DiffD1 DiffD2 II2 Un-iff sorted-wrt-append sup-ge2*)
subgoal by (*metis add-mono diff-le-self dual-order.trans enat-ord-simps*(*1*) *length-append order-mono-setup.refl*)
done
subgoal by (*fact FIN*)
done

lemma *FOREACHci-rule* :
assumes *IP*:
$\bigwedge x\ it\ \sigma.\ [\![\ x{\in}it;\ it{\subseteq}S;\ I\ it\ \sigma;\ c\ \sigma\ ]\!] \Longrightarrow f\ x\ \sigma \leq SPECT\ (emb\ (I\ (it{-}\{x\}))\ (enat\ body\text{-}time))$
assumes *II1*: $\bigwedge\sigma.\ [\![I\ \{\}\ \sigma]\!] \Longrightarrow P\ \sigma$
assumes *II2*: $\bigwedge it\ \sigma.\ [\![\ it{\neq}\{\};\ it{\subseteq}S;\ I\ it\ \sigma;\ \neg c\ \sigma\ ]\!] \Longrightarrow P\ \sigma$
assumes *FIN*: *finite S*
assumes *I0*: *I S σ0*
assumes *progress-f*: $\bigwedge a\ b.\ progress\ (f\ a\ b)$
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*))
unfolding *FOREACHci-def*
by (*rule FOREACHoci-rule*) (*simp-all add*: *assms*)

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*)
unfolding *nfoldli-def* **by** (*subst RECT-unfold*, *refine-mono*, *auto split*: *nat.split*)+

lemma *param-nfoldli*[*param*]:
shows (*nfoldli*,*nfoldli*) ∈
⟨*Ra*⟩*list-rel* → (*Rb*→*Id*) → (*Ra*→*Rb*→⟨*Rb*⟩*nrest-rel*) → *Rb* → ⟨*Rb*⟩*nrest-rel*
proof (*intro fun-relI*, *goal-cases*)
case (*1 l l′ c c′ f f′ s s′*)
thus *?case*
apply (*induct arbitrary*: *s s′*)
apply (*simp only*: *nfoldli-simps True-implies-equals*)
apply *parametricity*
apply (*simp only*: *nfoldli-simps True-implies-equals*)
apply (*parametricity*)
done
qed

lemma *nfoldli-no-ctd*[*simp*]: ¬*ctd s* ⟹ *nfoldli l ctd f s* = *RETURNT s*
by (*cases l*) *auto*

lemma *nfoldli-append*: *nfoldli* (*l1*@*l2*) *ctd f s* = *nfoldli l1 ctd f s* ≫= *nfoldli l2 ctd f*
by (*induction l1 arbitrary*: *s*) *simp-all*

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*
using *assms* **by** (*induction l arbitrary*: *s*) (*auto simp*: *pw-eq-iff refine-pw-simps*)

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 ⟹ I lr l σ ⟹ nfoldliIE I body-time l c f σ ≤ SPECT* (*emb P* (*body-time ∗ length l*))
proof (*induct l arbitrary*: *lr σ*)
case *Nil*
then show *?case* **by** (*auto simp*: *nfoldliIE-def RETURNT-def le-fun-def Some-le-emb′-conv dest*!: *FC*)
next
case (*Cons a l*)
show *?case*
proof(*cases c σ*)
case *True*
have *f a σ ≫= nfoldli l c f ≤ SPECT* (*emb P* (*enat* (*body-time + body-time ∗ length l*)))
apply (*rule T-specifies-I*)
apply (*vcg′‹−› rules*: *IS*[*THEN T-specifies-rev* , *THEN T-conseq4*]
Cons(*1*)[*unfolded nfoldliIE-def*, *THEN T-specifies-rev* , *THEN T-conseq4*])
using *True Cons*(*2*,*3*) **by** (*auto simp add*: *Some-eq-emb′-conv Some-le-emb′-conv*)
with *True* **show** *?thesis*
by (*auto simp add*: *nfoldliIE-def*)
next
case *False*
hence *P σ*
by (*rule FNC*[*OF Cons*(*2*)[*symmetric*] *Cons*(*3*)])
with *False* **show** *?thesis*
by (*auto simp add*: *nfoldliIE-def RETURNT-def le-fun-def Some-le-emb′-conv*)
qed
qed

lemma *nfoldliIE-rule*:
assumes *I0*: *I* [] *l0 σ0*
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*
assumes *T*: (*body-time ∗ length l0*) *≤ t*
shows *nfoldliIE I body-time l0 c f σ0 ≤ SPECT* (*emb P t*)
proof −

```
  have nfoldliIE I body-time l0 c f σ0 ≤ SPECT (emb P (body-time * length l0))
    by (rule nfoldliIE-rule'[where lr=[]]) (use assms in auto)
  also have ... ≤ SPECT (emb P t)
    by (rule SPECT-ub) (use T in ‹auto simp: le-fun-def›)
  finally show ?thesis .
qed


lemma nfoldli-refine[refine]:
  assumes (li, l) ∈ ⟨S⟩list-rel
    and (si, s) ∈ R
    and CR: (ci, c) ∈ R → bool-rel
    and [refine]: ⋀xi x si s. ⟦ (xi,x)∈S; (si,s)∈R; c s ⟧ ⟹ fi xi si ≤ ⇓R (f x s)
  shows nfoldli li ci fi si ≤ ⇓ R (nfoldli l c f s)
  using assms(1,2)
proof (induction arbitrary: si s rule: list-rel-induct)
  case Nil thus ?case by (simp add: RETURNT-refine)
next
  case (Cons xi x li l)
  note [refine] = Cons

  show ?case
    apply (simp add: RETURNT-refine  split del: if-split)
    apply refine-rcg
    using CR Cons.prems by (auto simp: RETURNT-refine  dest: fun-relD)
qed

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

lemma nfoldliIE'-rule:
  assumes
⋀x l1 l2 σ.
    l0 = l1 @ x # l2 ⟹
   I l1 (x # l2) σ ⟹ Some 0 ≤ lst (f x σ) (emb (I (l1 @ [x]) l2) (enat body-time))
I [] l0 σ0
(⋀σ. I l0 [] σ ⟹ Some (t + enat (body-time * length l0)) ≤ Q σ)
shows Some t ≤ lst (nfoldliIE' I body-time l0 f σ0) Q
  unfolding nfoldliIE'-def
  apply(rule nfoldliIE-rule[where P=I l0 [] and c=λ-. True and t=body-time *
length l0,
        THEN T-specifies-rev, THEN T-conseq4])
      apply fact
     subgoal apply(rule T-specifies-I) using assms(1) by auto
    subgoal by auto
   apply simp
  apply simp
  subgoal
    unfolding Some-eq-emb'-conv
    using assms(3) by auto
```

done

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*, *σ*) $\gg=$
(λ(-, *σ*). *RETURNT σ*))
proof (*induct l arbitrary*: *σ*)
case *Nil* **thus** *?case*
unfolding *whileIET-def*
by (*subst whileT-unfold*) (*auto simp*: *FOREACH-cond-def*)
next
case (*Cons x ls*)
show *?case*
proof (*cases c σ*)
case *False* **thus** *?thesis*
unfolding *whileIET-def* **by** (*subst whileT-unfold*) (*simp add*: *FOREACH-cond-def*)
next
case [*simp*]: *True*
from *Cons* **show** *?thesis*
unfolding *whileIET-def* **by** (*subst whileT-unfold*)
(*auto simp add*: *FOREACH-cond-def FOREACH-body-def intro*: *bindT-mono*)
qed
qed

lemma *rr*: *l0 = l1 @ a* $\Longrightarrow$ *a* $\neq$ [] $\Longrightarrow$ *l0 = l1 @ hd a # tl a* **by** *auto*

lemma *nfoldli-rule*:
assumes *I0*: *I* [] *l0 σ0*
assumes *IS*: $\bigwedge$*x l1 l2 σ*. ⟦ *l0=l1@x#l2*; *I l1* (*x#l2*) *σ*; *c σ* ⟧ $\Longrightarrow$ *f x σ* $\leq$ *SPECT* (*emb* (*I* (*l1@*[*x*]) *l2*) (*enat body-time*))
assumes *FNC*: $\bigwedge$*l1 l2 σ*. ⟦ *l0=l1@l2*; *I l1 l2 σ*; ¬*c σ* ⟧ $\Longrightarrow$ *P σ*
assumes *FC*: $\bigwedge$*σ*. ⟦ *I l0* [] *σ*; *c σ* ⟧ $\Longrightarrow$ *P σ*
assumes *progressf*: $\bigwedge$*x y*. *progress* (*f x y*)
shows *nfoldli l0 c f σ0* $\leq$ *SPECT* (*emb P* (*body-time* $*$ *length l0*))
apply (*rule order-trans*[*OF nfoldli-while*[
where *I*=λ(*l2*,*σ*). ∃ *l1*. *l0=l1@l2* ∧ *I l1 l2 σ* **and** *E*=λ(*l2*,*σ*). (*length l2*) $*$ *body-time*]])
unfolding *FOREACH-cond-def FOREACH-body-def*
apply(*rule T-specifies-I*)
apply(*vcg′-step* ‹*clarsimp*›)
apply *simp* **subgoal using** *I0* **by** *auto*
apply *simp* **subgoal**
apply(*vcg′-step* ‹*clarsimp*›)
apply (*elim exE conjE*)
subgoal for *a b l1*
apply(*vcg′-step* ‹*clarsimp*› *rules*: *IS*[*THEN T-specifies-rev* , *THEN T-conseq4*

```
])
          apply(rule rr)
           apply simp-all
     by(auto simp add: Some-le-mm3-Some-conv emb-eq-Some-conv left-diff-distrib'
                intro:  exI[where x=l1@[hd a]])
      done
  subgoal
    supply progressf [progress-rules] by (progress ‹clarsimp›)
  subgoal
    apply(vcg' ‹clarsimp›)
    subgoal for a σ
      apply(cases c σ)
       using FC FNC by(auto simp add: Some-le-emb'-conv  mult.commute)
    done
  done

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 (λl. 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
  }
  unfolding OP-def FOREACHoci-def LIST-FOREACH-def it-to-sorted-list-def
  by simp



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.