CIMP

Peter Gammie

March 17, 2025

Abstract

CIMP extends the small imperative language IMP with control non-determinism and constructs for synchronous message passing.

Contents

1 Point-free notation

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

abbreviation (*input*)
pred-K :: $'b \Rightarrow 'a \Rightarrow 'b$ (‹⟨-⟩›) **where**
$\langle f\rangle \equiv \lambda s.\ f$

abbreviation (*input*)
pred-not :: $('a \Rightarrow bool) \Rightarrow 'a \Rightarrow bool$ (‹¬ -› [*40*] *40*) **where**

$\neg a \equiv \lambda s.\ \neg a\ s$

abbreviation (*input*)
pred-conj :: $('a \Rightarrow bool) \Rightarrow ('a \Rightarrow bool) \Rightarrow 'a \Rightarrow bool$ (**infixr** ‹∧› *35*) **where**
$a \wedge b \equiv \lambda s.\ a\ s \wedge b\ s$

abbreviation (*input*)
pred-disj :: $('a \Rightarrow bool) \Rightarrow ('a \Rightarrow bool) \Rightarrow 'a \Rightarrow bool$ (**infixr** ‹∨› *30*) **where**
$a \vee b \equiv \lambda s.\ a\ s \vee b\ s$

abbreviation (*input*)
pred-implies :: $('a \Rightarrow bool) \Rightarrow ('a \Rightarrow bool) \Rightarrow 'a \Rightarrow bool$ (**infixr** ‹⟶› *25*) **where**
$a \longrightarrow b \equiv \lambda s.\ a\ s \longrightarrow b\ s$

abbreviation (*input*)
pred-iff :: $('a \Rightarrow bool) \Rightarrow ('a \Rightarrow bool) \Rightarrow 'a \Rightarrow bool$ (**infixr** ‹⟷› *25*) **where**
$a \longleftrightarrow b \equiv \lambda s.\ a\ s \longleftrightarrow b\ s$

abbreviation (*input*)
pred-eq :: $('a \Rightarrow 'b) \Rightarrow ('a \Rightarrow 'b) \Rightarrow 'a \Rightarrow bool$ (**infix** ‹=› *40*) **where**
$a = b \equiv \lambda s.\ a\ s = b\ s$

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

abbreviation (*input*)
pred-neq :: $('a \Rightarrow 'b) \Rightarrow ('a \Rightarrow 'b) \Rightarrow 'a \Rightarrow bool$ (**infix** ‹≠› *40*) **where**
$a \neq b \equiv \lambda s.\ a\ s \neq b\ s$

abbreviation (*input*)
pred-If :: $('a \Rightarrow bool) \Rightarrow ('a \Rightarrow 'b) \Rightarrow ('a \Rightarrow 'b) \Rightarrow 'a \Rightarrow 'b$ (‹(*If* (-)/ *Then* (-)/ *Else* (-))› [*0*, *0*, *10*] *10*)
where *If P Then x Else y* $\equiv \lambda s.$ *if P s then x s else y s*

abbreviation (*input*)
pred-less :: $('a \Rightarrow 'b::ord) \Rightarrow ('a \Rightarrow 'b) \Rightarrow 'a \Rightarrow bool$ (**infix** ‹<› *40*) **where**
$a < b \equiv \lambda s.\ a\ s < b\ s$

abbreviation (*input*)
pred-le :: $('a \Rightarrow 'b::ord) \Rightarrow ('a \Rightarrow 'b) \Rightarrow 'a \Rightarrow bool$ (**infix** ‹≤› *40*) **where**
$a \leq b \equiv \lambda s.\ a\ s \leq b\ s$

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

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

abbreviation (*input*)
fun-fanout :: $('a \Rightarrow 'b) \Rightarrow ('a \Rightarrow 'c) \Rightarrow 'a \Rightarrow 'b \times 'c$ (**infix** ‹⋈› *35*) **where**
$f \bowtie g \equiv \lambda x.\ (f\ x,\ g\ x)$

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

abbreviation (*input*)
  pred-ex :: ('b ⇒ 'a ⇒ *bool*) ⇒ 'a ⇒ *bool* (**binder** ‹∃ › *10*) **where**
  ∃ *x*. *P x* ≡ λ*s*. ∃ *x*. *P x s*

abbreviation (*input*)
  pred-app :: ('b ⇒ 'a ⇒ 'c) ⇒ ('a ⇒ 'b) ⇒ 'a ⇒ 'c (**infixl** ‹**$**› *100*) **where**
  f **$** *g* ≡ λ*s*. *f* (*g s*) *s*

abbreviation (*input*)
  pred-subseteq :: ('a ⇒ 'b *set*) ⇒ ('a ⇒ 'b *set*) ⇒ 'a ⇒ *bool* (**infix** ‹⊆› *50*) **where**
  A ⊆ *B* ≡ λ*s*. *A s* ⊆ *B s*

abbreviation (*input*)
  pred-union :: ('a ⇒ 'b *set*) ⇒ ('a ⇒ 'b *set*) ⇒ 'a ⇒ 'b *set* (**infixl** ‹∪› *65*) **where**
  a ∪ *b* ≡ λ*s*. *a s* ∪ *b s*

abbreviation (*input*)
  pred-inter :: ('a ⇒ 'b *set*) ⇒ ('a ⇒ 'b *set*) ⇒ 'a ⇒ 'b *set* (**infixl** ‹∩› *65*) **where**
  a ∩ *b* ≡ λ*s*. *a s* ∩ *b s*

More application specific.

abbreviation (*input*)
  pred-conjoin :: ('a ⇒ *bool*) *list* ⇒ 'a ⇒ *bool* **where**
  pred-conjoin xs ≡ *foldr* (∧) *xs* ⟨*True*⟩

abbreviation (*input*)
  pred-disjoin :: ('a ⇒ *bool*) *list* ⇒ 'a ⇒ *bool* **where**
  pred-disjoin xs ≡ *foldr* (∨) *xs* ⟨*False*⟩

abbreviation (*input*)
  pred-is-none :: ('a ⇒ 'b *option*) ⇒ 'a ⇒ *bool* (‹*NULL* -› [*40*] *40*) **where**
  NULL a ≡ λ*s*. *a s* = *None*

abbreviation (*input*)
  pred-empty :: ('a ⇒ 'b *set*) ⇒ 'a ⇒ *bool* (‹*EMPTY* -› [*40*] *40*) **where**
  EMPTY a ≡ λ*s*. *a s* = {}

abbreviation (*input*)
  pred-list-null :: ('a ⇒ 'b *list*) ⇒ 'a ⇒ *bool* (‹*LIST'-NULL* -› [*40*] *40*) **where**
  LIST-NULL a ≡ λ*s*. *a s* = []

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

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

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

2 Infinite Sequences

Infinite sequences and some operations on them.

We use the customary function-based representation.

```
type-synonym 'a seq = nat ⇒ 'a
type-synonym 'a seq-pred = 'a seq ⇒ bool

definition suffix :: 'a seq ⇒ nat ⇒ 'a seq (infixl ‹|s› 60) where
  σ |s i ≡ λj. σ (j + i)

primrec stake :: nat ⇒ 'a seq ⇒ 'a list where
  stake 0 σ = []
| stake (Suc n) σ = σ 0 # stake n (σ |s 1)

primrec shift :: 'a list ⇒ 'a seq ⇒ 'a seq (infixr ‹@−› 65) where
  shift [] σ = σ
| shift (x # xs) σ = (λi. case i of 0 ⇒ x | Suc i ⇒ shift xs σ i)


abbreviation interval-syn (‹-'(- → -')› [69, 0, 0] 70) where
  σ(i → j) ≡ stake j (σ |s i)

lemma suffix-eval: (σ |s i) j = σ (j + i)
unfolding suffix-def by simp

lemma suffix-plus: σ |s n |s m = σ |s (m + n)
unfolding suffix-def by (simp add: add.assoc)

lemma suffix-commute: ((σ |s n) |s m) = ((σ |s m) |s n)
by (simp add: suffix-plus add.commute)

lemma suffix-plus-com: σ |s m |s n = σ |s (m + n)
proof −
  have σ |s n |s m = σ |s (m + n) by (rule suffix-plus)
  then show σ |s m |s n = σ |s (m + n) by (simp add: suffix-commute)
qed

lemma suffix-zero: σ |s 0 = σ
unfolding suffix-def by simp

lemma comp-suffix: f ∘ σ |s i = (f ∘ σ) |s i
unfolding suffix-def comp-def by simp

lemmas suffix-simps[simp] =
  comp-suffix
  suffix-eval
  suffix-plus-com
  suffix-zero

lemma length-stake[simp]: length (stake n s) = n
by (induct n arbitrary: s) auto

lemma shift-simps[simp]:
   (xs @− σ) 0 = (if xs = [] then σ 0 else hd xs)
   (xs @− σ) |s Suc 0 = (if xs = [] then σ |s Suc 0 else tl xs @− σ)
by (induct xs) auto

lemma stake-nil[simp]:
  stake i σ = [] ⟷ i = 0
by (cases i; clarsimp)

lemma stake-shift:
```

stake i (w @− σ) = take i w @ stake (i − length w) σ
by (*induct i arbitrary*: *w*) (*auto simp*: *neq-Nil-conv*)

lemma *shift-snth-less*[*simp*]:
assumes $i < length\ xs$
shows $(xs\ @-\ \sigma)\ i = xs\ !\ i$
using *assms*
proof(*induct i arbitrary*: *xs*)
case (*Suc i xs*) **then show** *?case* **by** (*cases xs*) *simp-all*
qed (*simp add*: *hd-conv-nth nth-tl*)

lemma *shift-snth-ge*[*simp*]:
assumes $i \geq length\ xs$
shows $(xs\ @-\ \sigma)\ i = \sigma\ (i - length\ xs)$
using *assms*
proof(*induct i arbitrary*: *xs*)
case (*Suc i xs*) **then show** *?case* **by** (*cases xs*) *simp-all*
qed *simp*

lemma *shift-snth*:
$(xs\ @-\ \sigma)\ i = (if\ i < length\ xs\ then\ xs\ !\ i\ else\ \sigma\ (i - length\ xs))$
by *simp*

lemma *suffix-shift*:
$(xs\ @-\ \sigma)\ |_s\ i = drop\ i\ xs\ @-\ (\sigma\ |_s\ i - length\ xs)$
proof(*induct i arbitrary*: *xs*)
case (*Suc i xs*) **then show** *?case* **by** (*cases xs*) *simp-all*
qed *simp*

lemma *stake-nth*[*simp*]:
assumes $i < j$
shows $stake\ j\ s\ !\ i = s\ i$
using *assms* **by** (*induct j arbitrary*: *s i*) (*simp-all add*: *nth-Cons′*)

lemma *stake-suffix-id*:
$stake\ i\ \sigma\ @-\ (\sigma\ |_s\ i) = \sigma$
by (*induct i*) (*simp-all add*: *fun-eq-iff shift-snth split*: *nat.splits*)

lemma *id-stake-snth-suffix*:
$\sigma = (stake\ i\ \sigma\ @\ [\sigma\ i])\ @-\ (\sigma\ |_s\ Suc\ i)$
using *stake-suffix-id*
apply (*metis Suc-diff-le append-Nil2 diff-is-0-eq length-stake lessI nat.simps*(*3*) *nat-le-linear shift-snth stake-nil stake-shift take-Suc-conv-app-nth*)
done

lemma *stake-add*[*simp*]:
$stake\ i\ \sigma\ @\ stake\ j\ (\sigma\ |_s\ i) = stake\ (i + j)\ \sigma$
apply (*induct i arbitrary*: *σ*)
apply *simp*
apply *auto*
apply (*metis One-nat-def plus-1-eq-Suc suffix-plus-com*)
done

lemma *stake-append*: $stake\ n\ (u\ @-\ s) = take\ (min\ (length\ u)\ n)\ u\ @\ stake\ (n - length\ u)\ s$
proof (*induct n arbitrary*: *u*)
case (*Suc n*) **then show** *?case*
apply *clarsimp*
apply (*cases u*)

apply *auto*
done
qed *auto*

lemma *stake-shift-stake-shift*:
$stake\ i\ \sigma\ @-\ stake\ j\ (\sigma\ |_s\ i)\ @-\ \beta = stake\ (i + j)\ \sigma\ @-\ \beta$
apply (*induct i arbitrary*: σ)
apply *simp*
apply *auto*
apply (*metis One-nat-def plus-1-eq-Suc suffix-plus-com*)
done

lemma *stake-suffix-drop*:
$stake\ i\ (\sigma\ |_s\ j) = drop\ j\ (stake\ (i + j)\ \sigma)$
by (*metis append-eq-conv-conj length-stake semiring-normalization-rules*(*24*) *stake-add*)

lemma *stake-suffix*:
assumes $i \leq j$
shows $stake\ j\ \sigma\ @-\ u\ |_s\ i = \sigma(i \rightarrow j - i)\ @-\ u$
by (*simp add*: *assms stake-suffix-drop suffix-shift*)

2.1 Decomposing safety and liveness

Famously properties on infinite sequences can be decomposed into *safety* and *liveness* properties Alpern and Schneider (1985); Schneider (1987). See Kindler (1994) for an overview.

definition *safety* :: *'a seq-pred* ⇒ *bool* **where**
$safety\ P \longleftrightarrow (\forall \sigma.\ \neg P\ \sigma \longrightarrow (\exists i.\ \forall \beta.\ \neg P\ (stake\ i\ \sigma\ @-\ \beta)))$

lemma *safety-def2*: — Contraposition gives the customary prefix-closure definition
$safety\ P \longleftrightarrow (\forall \sigma.\ (\forall i.\ \exists \beta.\ P\ (stake\ i\ \sigma\ @-\ \beta)) \longrightarrow P\ \sigma)$
unfolding *safety-def* **by** *blast*

definition *liveness* :: *'a seq-pred* ⇒ *bool* **where**
$liveness\ P \longleftrightarrow (\forall \alpha.\ \exists \sigma.\ P\ (\alpha\ @-\ \sigma))$

lemmas *safetyI* = *iffD2*[*OF safety-def*, *rule-format*]
lemmas *safetyI2* = *iffD2*[*OF safety-def2*, *rule-format*]
lemmas *livenessI* = *iffD2*[*OF liveness-def*, *rule-format*]

lemma *safety-False*:
shows *safety* ($\lambda\sigma.$ *False*)
by (*rule safetyI*) *simp*

lemma *safety-True*:
shows *safety* ($\lambda\sigma.$ *True*)
by (*rule safetyI*) *simp*

lemma *safety-state-prop*:
shows $safety\ (\lambda\sigma.\ P\ (\sigma\ 0))$
by (*rule safetyI*) *auto*

lemma *safety-invariant*:
shows $safety\ (\lambda\sigma.\ \forall i.\ P\ (\sigma\ i))$
apply (*rule safetyI*)
apply *clarsimp*
apply (*metis length-stake lessI shift-snth-less stake-nth*)
done

lemma *safety-transition-relation*:
shows *safety* $(\lambda\sigma.\ \forall\, i.\ (\sigma\ i,\ \sigma\ (i + 1)) \in R)$
apply (*rule safetyI*)
apply *clarsimp*
apply (*metis* (*no-types*, *opaque-lifting*) *Suc-eq-plus1 add.left-neutral add-Suc-right add-diff-cancel-left′ le-add1 list.sel*(*1*) *list.simps*(*3*) *shift-simps*(*1*) *stake.simps*(*2*) *stake-suffix suffix-def*)
done

lemma *safety-conj*:
assumes *safety P*
assumes *safety Q*
shows *safety* $(P \wedge Q)$
using *assms* **unfolding** *safety-def* **by** *blast*

lemma *safety-always-eventually*[*simplified*]:
assumes *safety P*
assumes $\forall\, i.\ \exists\, j \geq i.\ \exists\, \beta.\ P\ (\sigma(0 \to j)\ @-\ \beta)$
shows $P\ \sigma$
using *assms* **unfolding** *safety-def2*
apply −
apply (*drule-tac* $x=\sigma$ **in** *spec*)
apply *clarsimp*
apply (*drule-tac* $x=i$ **in** *spec*)
apply *clarsimp*
apply (*rule-tac* $x=(stake\ j\ \sigma\ @-\ \beta)\ |_s\ i$ **in** *exI*)
apply (*simp add*: *stake-shift-stake-shift stake-suffix*)
done

lemma *safety-disj*:
assumes *safety P*
assumes *safety Q*
shows *safety* $(P \vee Q)$
unfolding *safety-def2* **using** *assms*
by (*metis safety-always-eventually add-diff-cancel-right′ diff-le-self le-add-same-cancel2*)

The decomposition is given by a form of closure.

definition M_p :: *′a seq-pred* ⇒ *′a seq-pred* **where**
$M_p\ P = (\lambda\sigma.\ \forall\, i.\ \exists\, \beta.\ P\ (stake\ i\ \sigma\ @-\ \beta))$

definition *Safe* :: *′a seq-pred* ⇒ *′a seq-pred* **where**
$Safe\ P = (P \vee M_p\ P)$

definition *Live* :: *′a seq-pred* ⇒ *′a seq-pred* **where**
$Live\ P = (P \vee \neg M_p\ P)$

lemma *decomp*:
$P = (Safe\ P \wedge Live\ P)$
unfolding *Safe-def Live-def* **by** *blast*

lemma *safe*:
safety (*Safe P*)
unfolding *Safe-def safety-def* M_p*-def*
apply *clarsimp*
apply (*simp add*: *stake-shift*)
apply (*rule-tac* $x=i$ **in** *exI*)
apply *clarsimp*
apply (*rule-tac* $x=i$ **in** *exI*)
apply *clarsimp*

done

lemma *live*:
  liveness (*Live* P)
proof(*rule livenessI*)
  fix α
  have $(\exists\beta.\ P\ (\alpha\ @-\ \beta)) \vee \neg(\exists\beta.\ P\ (\alpha\ @-\ \beta))$ **by** *blast*
  also have *?this* $\longleftrightarrow (\exists\beta.\ P\ (\alpha\ @-\ \beta) \vee (\forall\gamma.\ \neg P\ (\alpha\ @-\ \gamma)))$ **by** *blast*
  also have $\ldots \longleftrightarrow (\exists\beta.\ P\ (\alpha\ @-\ \beta) \vee (\exists i.\ i = \mathit{length}\ \alpha \wedge (\forall\gamma.\ \neg P\ (\mathit{stake}\ i\ (\alpha\ @-\ \beta)\ @-\ \gamma))))$ **by** (*simp add*: *stake-shift*)
  also have $\ldots \longrightarrow (\exists\beta.\ P\ (\alpha\ @-\ \beta) \vee (\exists i.\ (\forall\gamma.\ \neg P\ (\mathit{stake}\ i\ (\alpha\ @-\ \beta)\ @-\ \gamma))))$ **by** *blast*
  finally have $\exists\beta.\ P\ (\alpha\ @-\ \beta) \vee (\exists i.\ \forall\gamma.\ \neg\ P\ (\mathit{stake}\ i\ (\alpha\ @-\ \beta)\ @-\ \gamma))$ **.**
  then show $\exists\sigma.\ \mathit{Live}\ P\ (\alpha\ @-\ \sigma)$ **unfolding** *Live-def* M_p*-def* **by** *simp*
qed

Sistla (1994) proceeds to give a topological analysis of fairness. An *absolute* liveness property is a liveness property whose complement is stable.

definition *absolute-liveness* :: *'a seq-pred* $\Rightarrow$ *bool* **where** — closed under prepending any finite sequence
  absolute-liveness $P \longleftrightarrow (\exists\sigma.\ P\ \sigma) \wedge (\forall\sigma\ \alpha.\ P\ \sigma \longrightarrow P\ (\alpha\ @-\ \sigma))$

definition *stable* :: *'a seq-pred* $\Rightarrow$ *bool* **where** — closed under suffixes
  stable $P \longleftrightarrow (\exists\sigma.\ P\ \sigma) \wedge (\forall\sigma\ i.\ P\ \sigma \longrightarrow P\ (\sigma\ |_s\ i))$

lemma *absolute-liveness-liveness*:
  assumes *absolute-liveness* P
  shows *liveness* P
using *assms* **unfolding** *absolute-liveness-def liveness-def* **by** *blast*

lemma *stable-absolute-liveness*:
  assumes $P\ \sigma$
  assumes $\neg P\ \sigma'$ — extra hypothesis
  shows *stable* $P \longleftrightarrow$ *absolute-liveness* $(\neg\ P)$
using *assms* **unfolding** *stable-def absolute-liveness-def*
apply *auto*
apply (*metis cancel-comm-monoid-add-class.diff-cancel drop-eq-Nil order-refl shift.simps*(*1*) *suffix-shift suffix-zero*)
apply (*metis stake-suffix-id*)
done

definition *fairness* :: *'a seq-pred* $\Rightarrow$ *bool* **where**
  fairness $P \longleftrightarrow$ *stable* $P \wedge$ *absolute-liveness* P

lemma *fairness-safety*:
  assumes *safety* P
  assumes *fairness* F
  shows $(\forall\sigma.\ F\ \sigma \longrightarrow P\ \sigma) \longleftrightarrow (\forall\sigma.\ P\ \sigma)$
apply *rule*
using *assms*
apply *clarsimp*
unfolding *safety-def fairness-def stable-def absolute-liveness-def*
apply *clarsimp*
apply *blast+*
done

3 Linear Temporal Logic

To talk about liveness we need to consider infinitary behaviour on sequences. Traditionally future-time linear temporal logic (LTL) is used to do this Manna and Pnueli (1991); Owicki and Lamport (1982).

The following is a straightforward shallow embedding of the now-traditional anchored semantics of LTL Manna and Pnueli (1988). Some of it is adapted from the sophisticated TLA development in the AFP due to Grov and Merz (2011).

Unlike Lamport (2002), include the next operator, which is convenient for stating rules. Sometimes it allows us to ignore the system, i.e. to state rules as temporally valid (LTL-valid) rather than just temporally program valid (LTL-cimp-), in Jackson's terminology.

definition *state-prop* :: $('a \Rightarrow bool) \Rightarrow 'a\ seq\text{-}pred$ (‹⌈-⌉›) **where**
$\lceil P \rceil = (\lambda\sigma.\ P\ (\sigma\ 0))$

definition *next* :: $'a\ seq\text{-}pred \Rightarrow 'a\ seq\text{-}pred$ (‹○-› [80] 80) **where**
$(\bigcirc P) = (\lambda\sigma.\ P\ (\sigma\ |_s\ 1))$

definition *always* :: $'a\ seq\text{-}pred \Rightarrow 'a\ seq\text{-}pred$ (‹□-› [80] 80) **where**
$(\Box P) = (\lambda\sigma.\ \forall\, i.\ P\ (\sigma\ |_s\ i))$

definition *until* :: $'a\ seq\text{-}pred \Rightarrow 'a\ seq\text{-}pred \Rightarrow 'a\ seq\text{-}pred$ (**infixr** ‹𝒰› 30) **where**
$(P\ \mathcal{U}\ Q) = (\lambda\sigma.\ \exists\, i.\ Q\ (\sigma\ |_s\ i) \wedge (\forall\, k{<}i.\ P\ (\sigma\ |_s\ k)))$

definition *eventually* :: $'a\ seq\text{-}pred \Rightarrow 'a\ seq\text{-}pred$ (‹◇-› [80] 80) **where**
$(\Diamond P) = (\langle True \rangle\ \mathcal{U}\ P)$

definition *release* :: $'a\ seq\text{-}pred \Rightarrow 'a\ seq\text{-}pred \Rightarrow 'a\ seq\text{-}pred$ (**infixr** ‹ℛ› 30) **where**
$(P\ \mathcal{R}\ Q) = (\neg(\neg P\ \mathcal{U}\ \neg Q))$

definition *unless* :: $'a\ seq\text{-}pred \Rightarrow 'a\ seq\text{-}pred \Rightarrow 'a\ seq\text{-}pred$ (**infixr** ‹𝒲› 30) **where**
$(P\ \mathcal{W}\ Q) = ((P\ \mathcal{U}\ Q) \vee \Box P)$

abbreviation (*input*)
pred-always-imp-syn :: $'a\ seq\text{-}pred \Rightarrow 'a\ seq\text{-}pred \Rightarrow 'a\ seq\text{-}pred$ (**infixr** ‹↪› 25) **where**
$P \hookrightarrow Q \equiv \Box(P \longrightarrow Q)$

lemmas *defs* =
state-prop-def
always-def
eventually-def
next-def
release-def
unless-def
until-def

lemma *suffix-state-prop*[*simp*]:
shows $\lceil P \rceil\ (\sigma\ |_s\ i) = P\ (\sigma\ i)$
unfolding *defs* **by** *simp*

lemma *alwaysI*[*intro*]:
assumes $\bigwedge i.\ P\ (\sigma\ |_s\ i)$
shows $(\Box P)\ \sigma$
unfolding *defs* **using** *assms* **by** *blast*

lemma *alwaysD*:
assumes $(\Box P)\ \sigma$
shows $P\ (\sigma\ |_s\ i)$
using *assms* **unfolding** *defs* **by** *blast*

lemma *alwaysE*: $[\![(\Box P)\ \sigma;\ P\ (\sigma\ |_s\ i) \Longrightarrow Q]\!] \Longrightarrow Q$
unfolding *defs* **by** *blast*

lemma *always-induct*:
  assumes $P\ \sigma$
  assumes $(\Box(P \longrightarrow \bigcirc P))\ \sigma$
  shows $(\Box P)\ \sigma$
proof(*rule alwaysI*)
  fix *i* **from** *assms* **show** $P\ (\sigma\ |_s\ i)$
    unfolding *defs* **by** (*induct i*) *simp-all*
qed

lemma *seq-comp*:
  fixes σ :: *'a seq*
  fixes P :: *'b seq-pred*
  fixes f :: $'a \Rightarrow 'b$
  shows
    $(\Box P)\ (f \circ \sigma) \longleftrightarrow (\Box(\lambda\sigma.\ P\ (f \circ \sigma)))\ \sigma$
    $(\Diamond P)\ (f \circ \sigma) \longleftrightarrow (\Diamond(\lambda\sigma.\ P\ (f \circ \sigma)))\ \sigma$
    $(P\ \mathcal{U}\ Q)\ (f \circ \sigma) \longleftrightarrow ((\lambda\sigma.\ P\ (f \circ \sigma))\ \mathcal{U}\ (\lambda\sigma.\ Q\ (f \circ \sigma)))\ \sigma$
    $(P\ \mathcal{W}\ Q)\ (f \circ \sigma) \longleftrightarrow ((\lambda\sigma.\ P\ (f \circ \sigma))\ \mathcal{W}\ (\lambda\sigma.\ Q\ (f \circ \sigma)))\ \sigma$
unfolding *defs* **by** *simp-all*

lemma *nextI*[*intro*]:
  assumes $P\ (\sigma\ |_s\ Suc\ 0)$
  shows $(\bigcirc P)\ \sigma$
using *assms* **unfolding** *defs* **by** *simp*

lemma *untilI*[*intro*]:
  assumes $Q\ (\sigma\ |_s\ i)$
  assumes $\forall k{<}i.\ P\ (\sigma\ |_s\ k)$
  shows $(P\ \mathcal{U}\ Q)\ \sigma$
unfolding *defs* **using** *assms* **by** *blast*

lemma *untilE*:
  assumes $(P\ \mathcal{U}\ Q)\ \sigma$
  obtains *i* **where** $Q\ (\sigma\ |_s\ i)$ **and** $\forall k{<}i.\ P\ (\sigma\ |_s\ k)$
using *assms* **unfolding** *until-def* **by** *blast*

lemma *eventuallyI*[*intro*]:
  assumes $P\ (\sigma\ |_s\ i)$
  shows $(\Diamond P)\ \sigma$
unfolding *eventually-def* **using** *assms* **by** *blast*

lemma *eventuallyE*[*elim*]:
  assumes $(\Diamond P)\ \sigma$
  obtains *i* **where** $P\ (\sigma\ |_s\ i)$
using *assms* **unfolding** *defs* **by** (*blast elim*: *untilE*)

lemma *unless-alwaysI*:
  assumes $(\Box\ P)\ \sigma$
  shows $(P\ \mathcal{W}\ Q)\ \sigma$
using *assms* **unfolding** *defs* **by** *blast*

lemma *unless-untilI*:
  assumes $Q\ (\sigma\ |_s\ j)$
  assumes $\bigwedge i.\ i < j \Longrightarrow P\ (\sigma\ |_s\ i)$

shows $(P\ \mathcal{W}\ Q)\ \sigma$
unfolding *defs* **using** *assms* **by** *blast*

lemma *always-imp-refl*[*iff*]:
shows $(P \hookrightarrow P)\ \sigma$
unfolding *defs* **by** *blast*

lemma *always-imp-trans*:
assumes $(P \hookrightarrow Q)\ \sigma$
assumes $(Q \hookrightarrow R)\ \sigma$
shows $(P \hookrightarrow R)\ \sigma$
using *assms* **unfolding** *defs* **by** *blast*

lemma *always-imp-mp*:
assumes $(P \hookrightarrow Q)\ \sigma$
assumes $P\ \sigma$
shows $Q\ \sigma$
using *assms* **unfolding** *defs* **by** (*metis suffix-zero*)

lemma *always-imp-mp-suffix*:
assumes $(P \hookrightarrow Q)\ \sigma$
assumes $P\ (\sigma\ |_s\ i)$
shows $Q\ (\sigma\ |_s\ i)$
using *assms* **unfolding** *defs* **by** *metis*

Some basic facts and equivalences, mostly sanity.

lemma *necessitation*:
$(\bigwedge s.\ P\ s) \Longrightarrow (\Box P)\ \sigma$
$(\bigwedge s.\ P\ s) \Longrightarrow (\Diamond P)\ \sigma$
$(\bigwedge s.\ P\ s) \Longrightarrow (P\ \mathcal{W}\ Q)\ \sigma$
$(\bigwedge s.\ Q\ s) \Longrightarrow (P\ \mathcal{U}\ Q)\ \sigma$
unfolding *defs* **by** *auto*

lemma *cong*:
$(\bigwedge s.\ P\ s = P'\ s) \Longrightarrow \lceil P \rceil = \lceil P' \rceil$
$(\bigwedge \sigma.\ P\ \sigma = P'\ \sigma) \Longrightarrow (\Box P) = (\Box P')$
$(\bigwedge \sigma.\ P\ \sigma = P'\ \sigma) \Longrightarrow (\Diamond P) = (\Diamond P')$
$(\bigwedge \sigma.\ P\ \sigma = P'\ \sigma) \Longrightarrow (\bigcirc P) = (\bigcirc P')$
$[\![\bigwedge \sigma.\ P\ \sigma = P'\ \sigma;\ \bigwedge \sigma.\ Q\ \sigma = Q'\ \sigma]\!] \Longrightarrow (P\ \mathcal{U}\ Q) = (P'\ \mathcal{U}\ Q')$
$[\![\bigwedge \sigma.\ P\ \sigma = P'\ \sigma;\ \bigwedge \sigma.\ Q\ \sigma = Q'\ \sigma]\!] \Longrightarrow (P\ \mathcal{W}\ Q) = (P'\ \mathcal{W}\ Q')$
unfolding *defs* **by** *auto*

lemma *norm*[*simp*]:
$\lceil \langle \mathit{False} \rangle \rceil = \langle \mathit{False} \rangle$
$\lceil \langle \mathit{True} \rangle \rceil = \langle \mathit{True} \rangle$
$(\neg \lceil p \rceil) = \lceil \neg p \rceil$
$(\lceil p \rceil \wedge \lceil q \rceil) = \lceil p \wedge q \rceil$
$(\lceil p \rceil \vee \lceil q \rceil) = \lceil p \vee q \rceil$
$(\lceil p \rceil \longrightarrow \lceil q \rceil) = \lceil p \longrightarrow q \rceil$
$(\lceil p \rceil\ \sigma \wedge \lceil q \rceil\ \sigma) = \lceil p \wedge q \rceil\ \sigma$
$(\lceil p \rceil\ \sigma \vee \lceil q \rceil\ \sigma) = \lceil p \vee q \rceil\ \sigma$
$(\lceil p \rceil\ \sigma \longrightarrow \lceil q \rceil\ \sigma) = \lceil p \longrightarrow q \rceil\ \sigma$

$(\bigcirc \langle \mathit{False} \rangle) = \langle \mathit{False} \rangle$
$(\bigcirc \langle \mathit{True} \rangle) = \langle \mathit{True} \rangle$

$(\Box \langle \mathit{False} \rangle) = \langle \mathit{False} \rangle$
$(\Box \langle \mathit{True} \rangle) = \langle \mathit{True} \rangle$

$(\neg\Box\ P)\ \sigma = (\Diamond\ (\neg\ P))\ \sigma$
$(\Box\Box\ P) = (\Box\ P)$

$(\Diamond\langle False\rangle) = \langle False\rangle$
$(\Diamond\langle True\rangle) = \langle True\rangle$
$(\neg\Diamond\ P) = (\Box\ (\neg\ P))$
$(\Diamond\Diamond\ P) = (\Diamond\ P)$

$(P\ \mathcal{W}\ \langle False\rangle) = (\Box\ P)$

$(\neg(P\ \mathcal{U}\ Q))\ \sigma = (\neg P\ \mathcal{R}\ \neg Q)\ \sigma$
$(\langle False\rangle\ \mathcal{U}\ P) = P$
$(P\ \mathcal{U}\ \langle False\rangle) = \langle False\rangle$
$(P\ \mathcal{U}\ \langle True\rangle) = \langle True\rangle$
$(\langle True\rangle\ \mathcal{U}\ P) = (\Diamond\ P)$
$(P\ \mathcal{U}\ (P\ \mathcal{U}\ Q)) = (P\ \mathcal{U}\ Q)$

$(\neg(P\ \mathcal{R}\ Q))\ \sigma = (\neg P\ \mathcal{U}\ \neg Q)\ \sigma$
$(\langle False\rangle\ \mathcal{R}\ P) = (\Box P)$
$(P\ \mathcal{R}\ \langle False\rangle) = \langle False\rangle$
$(\langle True\rangle\ \mathcal{R}\ P) = P$
$(P\ \mathcal{R}\ \langle True\rangle) = \langle True\rangle$

unfolding *defs*
apply (*auto simp*: *fun-eq-iff*)
apply (*metis suffix-plus suffix-zero*)
apply (*metis suffix-plus suffix-zero*)
  apply *fastforce*
  apply *force*
apply (*metis add.commute add-diff-inverse-nat less-diff-conv2 not-le*)
apply (*metis add.right-neutral not-less0*)
  apply *force*
  apply *fastforce*
done

lemma *always-conj-distrib*: $(\Box(P \wedge Q)) = (\Box P \wedge \Box Q)$
unfolding *defs* **by** *auto*

lemma *eventually-disj-distrib*: $(\Diamond(P \vee Q)) = (\Diamond P \vee \Diamond Q)$
unfolding *defs* **by** *auto*

lemma *always-eventually*[*elim*!]:
  assumes $(\Box P)\ \sigma$
  shows $(\Diamond P)\ \sigma$
using *assms* **unfolding** *defs* **by** *auto*

lemma *eventually-imp-conv-disj*: $(\Diamond(P \longrightarrow Q)) = (\Diamond(\neg P) \vee \Diamond Q)$
unfolding *defs* **by** *auto*

lemma *eventually-imp-distrib*:
  $(\Diamond(P \longrightarrow Q)) = (\Box P \longrightarrow \Diamond Q)$
unfolding *defs* **by** *auto*

lemma *unfold*:
  $(\Box\ P)\ \sigma = (P \wedge \bigcirc\Box P)\ \sigma$
  $(\Diamond\ P)\ \sigma = (P \vee \bigcirc\Diamond P)\ \sigma$
  $(P\ \mathcal{W}\ Q)\ \sigma = (Q \vee (P \wedge \bigcirc(P\ \mathcal{W}\ Q)))\ \sigma$
  $(P\ \mathcal{U}\ Q)\ \sigma = (Q \vee (P \wedge \bigcirc(P\ \mathcal{U}\ Q)))\ \sigma$

$(P\ \mathcal{R}\ Q)\ \sigma = (Q \wedge (P \vee \bigcirc(P\ \mathcal{R}\ Q)))\ \sigma$
unfolding *defs*
apply −
apply (*metis* (*full-types*) *add.commute add-diff-inverse-nat less-one suffix-plus suffix-zero*)
apply (*metis* (*full-types*) *One-nat-def add.right-neutral add-Suc-right lessI less-Suc-eq-0-disj suffix-plus suffix-zero*)

apply *auto*
apply *fastforce*
apply (*metis gr0-conv-Suc nat-neq-iff not-less-eq suffix-zero*)
apply (*metis suffix-zero*)
apply *force*
using *less-Suc-eq-0-disj* **apply** *fastforce*
apply (*metis gr0-conv-Suc nat-neq-iff not-less0 suffix-zero*)

apply *fastforce*
apply (*case-tac i*; *auto*)
apply *force*
using *less-Suc-eq-0-disj* **apply** *force*

apply *force*
using *less-Suc-eq-0-disj* **apply** *fastforce*
apply *fastforce*
apply (*case-tac i*; *auto*)
done

lemma *mono*:
$[\![(\Box P)\ \sigma;\ \bigwedge\sigma.\ P\ \sigma \Longrightarrow\ P'\ \sigma]\!] \Longrightarrow (\Box P')\ \sigma$
$[\![(\Diamond P)\ \sigma;\ \bigwedge\sigma.\ P\ \sigma \Longrightarrow\ P'\ \sigma]\!] \Longrightarrow (\Diamond P')\ \sigma$
$[\![(P\ \mathcal{U}\ Q)\ \sigma;\ \bigwedge\sigma.\ P\ \sigma \Longrightarrow P'\ \sigma;\ \bigwedge\sigma.\ Q\ \sigma \Longrightarrow\ Q'\ \sigma]\!] \Longrightarrow (P'\ \mathcal{U}\ Q')\ \sigma$
$[\![(P\ \mathcal{W}\ Q)\ \sigma;\ \bigwedge\sigma.\ P\ \sigma \Longrightarrow P'\ \sigma;\ \bigwedge\sigma.\ Q\ \sigma \Longrightarrow\ Q'\ \sigma]\!] \Longrightarrow (P'\ \mathcal{W}\ Q')\ \sigma$
unfolding *defs* **by** *force+*

lemma *always-imp-mono*:
$[\![(\Box P)\ \sigma;\ (P \hookrightarrow P')\ \sigma]\!] \Longrightarrow (\Box P')\ \sigma$
$[\![(\Diamond P)\ \sigma;\ (P \hookrightarrow P')\ \sigma]\!] \Longrightarrow (\Diamond P')\ \sigma$
$[\![(P\ \mathcal{U}\ Q)\ \sigma;\ (P \hookrightarrow P')\ \sigma;\ (Q \hookrightarrow Q')\ \sigma]\!] \Longrightarrow (P'\ \mathcal{U}\ Q')\ \sigma$
$[\![(P\ \mathcal{W}\ Q)\ \sigma;\ (P \hookrightarrow P')\ \sigma;\ (Q \hookrightarrow Q')\ \sigma]\!] \Longrightarrow (P'\ \mathcal{W}\ Q')\ \sigma$
unfolding *defs* **by** *force+*

lemma *next-conj-distrib*:
$(\bigcirc(P \wedge Q)) = (\bigcirc P \wedge \bigcirc Q)$
unfolding *defs* **by** *auto*

lemma *next-disj-distrib*:
$(\bigcirc(P \vee Q)) = (\bigcirc P \vee \bigcirc Q)$
unfolding *defs* **by** *auto*

lemma *until-next-distrib*:
$(\bigcirc(P\ \mathcal{U}\ Q)) = (\bigcirc P\ \mathcal{U}\ \bigcirc Q)$
unfolding *defs* **by** (*auto simp*: *fun-eq-iff*)

lemma *until-imp-eventually*:
$((P\ \mathcal{U}\ Q) \longrightarrow \Diamond Q)\ \sigma$
unfolding *defs* **by** *auto*

lemma *until-until-disj*:
assumes $(P\ \mathcal{U}\ Q\ \mathcal{U}\ R)\ \sigma$
shows $((P \vee Q)\ \mathcal{U}\ R)\ \sigma$

using *assms* **unfolding** *defs*
apply *clarsimp*
apply (*metis* (*full-types*) *add-diff-inverse-nat nat-add-left-cancel-less*)
done

lemma *unless-unless-disj*:
  assumes $(P\ \mathcal{W}\ Q\ \mathcal{W}\ R)\ \sigma$
  shows $((P \vee Q)\ \mathcal{W}\ R)\ \sigma$
using *assms* **unfolding** *defs*
apply *auto*
 apply (*metis add.commute add-diff-inverse-nat leI less-diff-conv2*)
apply (*metis add-diff-inverse-nat*)
done

lemma *until-conj-distrib*:
  $((P \wedge Q)\ \mathcal{U}\ R) = ((P\ \mathcal{U}\ R) \wedge (Q\ \mathcal{U}\ R))$
unfolding *defs*
apply (*auto simp*: *fun-eq-iff*)
apply (*metis dual-order.strict-trans nat-neq-iff*)
done

lemma *until-disj-distrib*:
  $(P\ \mathcal{U}\ (Q \vee R)) = ((P\ \mathcal{U}\ Q) \vee (P\ \mathcal{U}\ R))$
unfolding *defs* **by** (*auto simp*: *fun-eq-iff*)

lemma *eventually-until*:
  $(\Diamond P) = (\neg P\ \mathcal{U}\ P)$
unfolding *defs*
apply (*auto simp*: *fun-eq-iff*)
apply (*case-tac* P $(x\ |_s\ 0)$)
 apply *blast*
apply (*drule* (*1*) *ex-least-nat-less*)
apply (*metis le-simps*(*2*))
done

lemma *eventually-until-eventually*:
  $(\Diamond(P\ \mathcal{U}\ Q)) = (\Diamond Q)$
unfolding *defs* **by** *force*

lemma *eventually-unless-until*:
  $((P\ \mathcal{W}\ Q) \wedge \Diamond Q) = (P\ \mathcal{U}\ Q)$
unfolding *defs* **by** *force*

lemma *eventually-always-imp-always-eventually*:
  assumes $(\Diamond\Box P)\ \sigma$
  shows $(\Box\Diamond P)\ \sigma$
using *assms* **unfolding** *defs* **by** (*metis suffix-commute*)

lemma *eventually-always-next-stable*:
  assumes $(\Diamond P)\ \sigma$
  assumes $(P \hookrightarrow \bigcirc P)\ \sigma$
  shows $(\Diamond\Box P)\ \sigma$
using *assms* **by** (*metis* (*no-types*) *eventuallyI alwaysD always-induct eventuallyE norm*(*15*))

lemma *next-stable-imp-eventually-always*:
  assumes $(P \hookrightarrow \bigcirc P)\ \sigma$
  shows $(\Diamond P \longrightarrow \Diamond\Box P)\ \sigma$
using *assms eventually-always-next-stable* **by** *blast*

lemma *always-eventually-always*:
  $\Diamond\Box\Diamond P = \Box\Diamond P$
unfolding *defs* **by** (*clarsimp simp*: *fun-eq-iff*) (*metis add.left-commute semiring-normalization-rules*(*25*))

lemma *stable-unless*:
  assumes $(P \hookrightarrow \bigcirc(P \vee Q))\ \sigma$
  shows $(P \hookrightarrow (P\ \mathcal{W}\ Q))\ \sigma$
using *assms* **unfolding** *defs*
apply −
apply (*rule ccontr*)
apply *clarsimp*
apply (*drule* (*1*) *ex-least-nat-less*[**where** $P{=}\lambda j.\ \neg P\ (\sigma\ |_s\ i + j)$ **for** *i*, *simplified*])
apply *clarsimp*
apply (*metis add-Suc-right le-less less-Suc-eq*)
done

lemma *unless-induct*: — Rule WAIT from Manna and Pnueli (1995, Fig 3.3)
  assumes *I*: $(I \hookrightarrow \bigcirc(I \vee R))\ \sigma$
  assumes *P*: $(P \hookrightarrow I \vee R)\ \sigma$
  assumes *Q*: $(I \hookrightarrow Q)\ \sigma$
  shows $(P \hookrightarrow Q\ \mathcal{W}\ R)\ \sigma$
apply (*intro alwaysI impI*)
apply (*erule impE*[*OF alwaysD*[*OF P*]])
apply (*erule disjE*)
 apply (*rule always-imp-mono*(*4*)[**where** *P*=*I* **and** *Q*=*R*])
   apply (*erule mp*[*OF alwaysD*[*OF stable-unless*[*OF I*]]])
  apply (*simp add*: *Q alwaysD*)
 apply *blast*
apply (*simp add*: *unfold*)
done

3.1 Leads-to and leads-to-via

Most of our assertions will be of the form $\lambda s.\ A\ s \longrightarrow (\Diamond C)\ s$ (pronounced "A leads to C") or $\lambda s.\ A\ s \longrightarrow (B\ \mathcal{U}\ C)\ s$ ("A leads to C via B").

Most of these rules are due to Jackson (1998) who used leads-to-via in a sequential setting. Others are due to Manna and Pnueli (1991).

The leads-to-via connective is similar to the "ensures" modality of Chandy and Misra (1989, §3.4.4).

abbreviation (*input*)
  leads-to :: *'a seq-pred* ⇒ *'a seq-pred* ⇒ *'a seq-pred* (**infixr** ‹⇝› *25*) **where**
  $P \leadsto Q \equiv P \hookrightarrow \Diamond Q$

lemma *leads-to-refl*:
  shows $(P \leadsto P)\ \sigma$
by (*metis* (*no-types*, *lifting*) *necessitation*(*1*) *unfold*(*2*))

lemma *leads-to-trans*:
  assumes $(P \leadsto Q)\ \sigma$
  assumes $(Q \leadsto R)\ \sigma$
  shows $(P \leadsto R)\ \sigma$
using *assms* **unfolding** *defs* **by** *clarsimp* (*metis semiring-normalization-rules*(*25*))

lemma *leads-to-eventuallyE*:
  assumes $(P \leadsto Q)\ \sigma$
  assumes $(\Diamond P)\ \sigma$

shows $(\Diamond Q)\ \sigma$
using *assms* **unfolding** *defs* **by** *auto*

lemma *leads-to-mono*:
assumes $(P' \hookrightarrow P)\ \sigma$
assumes $(Q \hookrightarrow Q')\ \sigma$
assumes $(P \leadsto Q)\ \sigma$
shows $(P' \leadsto Q')\ \sigma$
using *assms* **unfolding** *defs* **by** *clarsimp blast*

lemma *leads-to-eventually*:
shows $(P \leadsto Q \longrightarrow \Diamond P \longrightarrow \Diamond Q)\ \sigma$
by (*metis* (*no-types*, *lifting*) *alwaysI unfold*(*2*))

lemma *leads-to-disj*:
assumes $(P \leadsto R)\ \sigma$
assumes $(Q \leadsto R)\ \sigma$
shows $((P \vee Q) \leadsto R)\ \sigma$
using *assms* **unfolding** *defs* **by** *simp*

lemma *leads-to-leads-to-viaE*:
shows $((P \hookrightarrow P\ \mathcal{U}\ Q) \longrightarrow P \leadsto Q)\ \sigma$
unfolding *defs* **by** *clarsimp blast*

lemma *leads-to-via-concl-weaken*:
assumes $(R \hookrightarrow R')\ \sigma$
assumes $(P \hookrightarrow Q\ \mathcal{U}\ R)\ \sigma$
shows $(P \hookrightarrow Q\ \mathcal{U}\ R')\ \sigma$
using *assms* **unfolding** *LTL.defs* **by** *force*

lemma *leads-to-via-trans*:
assumes $(A \hookrightarrow B\ \mathcal{U}\ C)\ \sigma$
assumes $(C \hookrightarrow D\ \mathcal{U}\ E)\ \sigma$
shows $(A \hookrightarrow (B \vee D)\ \mathcal{U}\ E)\ \sigma$
proof(*rule alwaysI*, *rule impI*)
fix *i* **assume** $A\ (\sigma\ |_s\ i)$ **with** *assms* **show** $((B \vee D)\ \mathcal{U}\ E)\ (\sigma\ |_s\ i)$
apply −
apply (*erule alwaysE*[**where** *i*=*i*])
apply *clarsimp*
apply (*erule untilE*)
apply *clarsimp*
apply (*drule* (*1*) *always-imp-mp-suffix*)
apply (*erule untilE*)
apply *clarsimp*
apply (*rule-tac i*=*ia* + *iaa* **in** *untilI*; *simp add*: *ac-simps*)
apply (*metis* (*full-types*) *add.assoc leI le-Suc-ex nat-add-left-cancel-less*)
done
qed

lemma *leads-to-via-disj*: — useful for case distinctions
assumes $(P \hookrightarrow Q\ \mathcal{U}\ R)\ \sigma$
assumes $(P' \hookrightarrow Q'\ \mathcal{U}\ R)\ \sigma$
shows $(P \vee P' \hookrightarrow (Q \vee Q')\ \mathcal{U}\ R)\ \sigma$
using *assms* **unfolding** *defs* **by** (*auto 10 0*)

lemma *leads-to-via-disj'*: — more like a chaining rule
assumes $(A \hookrightarrow B\ \mathcal{U}\ C)\ \sigma$
assumes $(C \hookrightarrow D\ \mathcal{U}\ E)\ \sigma$

```
  shows (A ∨ C ↪ (B ∨ D) 𝒰 E) σ
proof(rule alwaysI, rule impI, erule disjE)
  fix i assume A (σ |s i) with assms show ((B ∨ D) 𝒰 E) (σ |s i)
    apply −
    apply (erule alwaysE[where i=i])
    apply clarsimp
    apply (erule untilE)
    apply clarsimp
    apply (drule (1) always-imp-mp-suffix)
    apply (erule untilE)
    apply clarsimp
    apply (rule-tac i=ia + iaa in untilI; simp add: ac-simps)
    apply (metis (full-types) add.assoc leI le-Suc-ex nat-add-left-cancel-less)
    done
next
  fix i assume C (σ |s i) with assms(2) show ((B ∨ D) 𝒰 E) (σ |s i)
    apply −
    apply (erule alwaysE[where i=i])
    apply (simp add: mono)
    done
qed

lemma leads-to-via-stable-augmentation:
  assumes stable: (P ∧ Q ↪ ○Q) σ
  assumes U: (A ↪ P 𝒰 C) σ
  shows ((A ∧ Q) ↪ P 𝒰 (C ∧ Q)) σ
proof(intro alwaysI impI, elim conjE)
  fix i assume AP: A (σ |s i) Q (σ |s i)
  have Q (σ |s (j + i)) if Q (σ |s i) and ∀k<j. P (σ |s (k + i)) for j
    using that stable by (induct j; force simp: defs)
  with U AP show (P 𝒰 (λσ. C σ ∧ Q σ)) (σ |s i)
    unfolding defs by clarsimp (metis (full-types) add.commute)
qed

lemma leads-to-via-wf:
  assumes wf R
  assumes indhyp: ⋀t. (A ∧ ⌈δ = ⟨t⟩⌉ ↪ B 𝒰 (A ∧ ⌈δ ⊗ ⟨t⟩ ∈ ⟨R⟩⌉ ∨ C)) σ
  shows (A ↪ B 𝒰 C) σ
proof(intro alwaysI impI)
  fix i assume A (σ |s i) with ‹wf R› show (B 𝒰 C) (σ |s i)
  proof(induct δ (σ i) arbitrary: i)
    case (less i) with indhyp[where t=δ (σ i)] show ?case
      apply −
      apply (drule alwaysD[where i=i])
      apply clarsimp
      apply (erule untilE)
      apply (rename-tac j)
       apply (erule disjE; clarsimp)
       apply (drule-tac x=i + j in meta-spec; clarsimp)
       apply (erule untilE; clarsimp)
       apply (rename-tac j k)
       apply (rule-tac i=j + k in untilI)
        apply (simp add: add.assoc)
       apply clarsimp
       apply (metis add.assoc add.commute add-diff-inverse-nat less-diff-conv2 not-le)
      apply auto
      done
  qed
```

qed

The well-founded response rule due to Manna and Pnueli (2010, Fig 1.23: `WELL` (well-founded response)), generalised to an arbitrary set of assertions and sequence predicates.

- *W1* generalised to be contingent.
- *W2* is a well-founded set of assertions that by *W1* includes *P*

lemma *leads-to-wf*:
  fixes *Is* :: (*'a seq-pred* × (*'a* ⇒ *'b*)) *set*
  assumes *wf* (*R* :: *'b rel*)
  assumes *W1*: $(\Box(\exists\,\varphi.\ \lceil\langle\varphi{\in}fst\text{ ‘ }Is\rangle\rceil \wedge (P \longrightarrow \varphi)))\ \sigma$
  assumes *W2*: $\forall(\varphi, \delta){\in}Is.\ \exists(\varphi', \delta'){\in}insert\ (Q, \delta 0)\ Is.\ \forall t.\ (\varphi \wedge \lceil\delta = \langle t\rangle\rceil \leadsto \varphi' \wedge \lceil\delta' \otimes \langle t\rangle \in \langle R\rangle\rceil)\ \sigma$
  shows $(P \leadsto Q)\ \sigma$
proof –
  have $(\varphi \wedge \lceil\delta = \langle t\rangle\rceil \leadsto Q)\ \sigma$ **if** $(\varphi, \delta) \in Is$ **for** $\varphi\ \delta\ t$
    using ‹*wf R*› *that W2*
    apply (*induct t arbitrary*: φ δ)
    unfolding *LTL.defs split-def*
    apply *clarsimp*
    apply (*metis* (*no-types*, *opaque-lifting*) *ab-semigroup-add-class.add-ac*(*1*) *fst-eqD snd-conv surjective-pairing*)
    done
  with *W1* **show** *?thesis*
    apply –
    apply (*rule alwaysI*)
    apply *clarsimp*
    apply (*erule-tac i=i* **in** *alwaysE*)
    apply *clarsimp*
    using *alwaysD suffix-state-prop* **apply** *blast*
    done
qed

3.2 Fairness

A few renderings of weak fairness. van Glabbeek and Höfner (2019) call this "response to insistence" as a generalisation of weak fairness.

definition *weakly-fair* :: *'a seq-pred* ⇒ *'a seq-pred* ⇒ *'a seq-pred* **where**
  weakly-fair enabled taken = $(\Box enabled \hookrightarrow \Diamond taken)$

lemma *weakly-fair-def2*:
  shows *weakly-fair enabled taken* = $\Box(\neg\Box(enabled \wedge \neg taken))$
unfolding *weakly-fair-def* **by** (*metis* (*full-types*) *always-conj-distrib norm*(*18*))

lemma *weakly-fair-def3*:
  shows *weakly-fair enabled taken* = $(\Diamond\Box enabled \longrightarrow \Box\Diamond taken)$
unfolding *weakly-fair-def2*
apply (*clarsimp simp*: *fun-eq-iff*)

unfolding *defs*
apply *auto*
apply (*metis* (*full-types*) *add.left-commute semiring-normalization-rules*(*25*))
done

lemma *weakly-fair-def4*:
  shows *weakly-fair enabled taken* = $\Box\Diamond(enabled \longrightarrow taken)$
using *weakly-fair-def2* **by** *force*

lemma *mp-weakly-fair*:
  assumes *weakly-fair enabled taken* σ
  assumes ($\Box$*enabled*) σ
  shows ($\Diamond$*taken*) σ
using *assms* **unfolding** *weakly-fair-def* **using** *always-imp-mp* **by** *blast*

lemma *always-weakly-fair*:
  shows $\Box$(*weakly-fair enabled taken*) = *weakly-fair enabled taken*
unfolding *weakly-fair-def* **by** *simp*

lemma *eventually-weakly-fair*:
  shows $\Diamond$(*weakly-fair enabled taken*) = *weakly-fair enabled taken*
unfolding *weakly-fair-def2* **by** (*simp add*: *always-eventually-always*)

lemma *weakly-fair-weaken*:
  assumes (*enabled'* $\hookrightarrow$ *enabled*) σ
  assumes (*taken* $\hookrightarrow$ *taken'*) σ
  shows (*weakly-fair enabled taken* $\hookrightarrow$ *weakly-fair enabled' taken'*) σ
using *assms* **unfolding** *weakly-fair-def defs* **by** *simp blast*

lemma *weakly-fair-unless-until*:
  shows (*weakly-fair enabled taken* $\wedge$ (*enabled* $\hookrightarrow$ *enabled* $\mathcal{W}$ *taken*)) = (*enabled* $\hookrightarrow$ *enabled* $\mathcal{U}$ *taken*)
unfolding *defs weakly-fair-def*
apply (*auto simp*: *fun-eq-iff*)
apply (*metis add.right-neutral*)
done

lemma *stable-leads-to-eventually*:
  assumes (*enabled* $\hookrightarrow$ $\bigcirc$(*enabled* $\vee$ *taken*)) σ
  shows (*enabled* $\hookrightarrow$ ($\Box$*enabled* $\vee$ $\Diamond$*taken*)) σ
using *assms* **unfolding** *defs*
apply −
apply (*rule ccontr*)
apply *clarsimp*
apply (*drule* (*1*) *ex-least-nat-less*[**where** P=$\lambda j.$ $\neg$ *enabled* (σ $|_s$ i + j) **for** i, *simplified*])
apply *clarsimp*
apply (*metis add-Suc-right leI less-irrefl-nat*)
done

lemma *weakly-fair-stable-leads-to*:
  assumes (*weakly-fair enabled taken*) σ
  assumes (*enabled* $\hookrightarrow$ $\bigcirc$(*enabled* $\vee$ *taken*)) σ
  shows (*enabled* $\leadsto$ *taken*) σ
using *stable-leads-to-eventually*[*OF assms*(*2*)] *assms*(*1*) **unfolding** *defs weakly-fair-def*
by (*auto simp*: *fun-eq-iff*)

lemma *weakly-fair-stable-leads-to-via*:
  assumes (*weakly-fair enabled taken*) σ
  assumes (*enabled* $\hookrightarrow$ $\bigcirc$(*enabled* $\vee$ *taken*)) σ
  shows (*enabled* $\hookrightarrow$ *enabled* $\mathcal{U}$ *taken*) σ
using *stable-unless*[*OF assms*(*2*)] *assms*(*1*) **by** (*metis* (*mono-tags*) *weakly-fair-unless-until*)

Similarly for strong fairness. van Glabbeek and Höfner (2019) call this "response to persistence" as a generalisation of strong fairness.

definition *strongly-fair* :: *'a seq-pred* $\Rightarrow$ *'a seq-pred* $\Rightarrow$ *'a seq-pred* **where**
  strongly-fair enabled taken = ($\Box\Diamond$*enabled* $\hookrightarrow$ $\Diamond$*taken*)

lemma *strongly-fair-def2*:

strongly-fair enabled taken = $\Box(\neg\Box(\Diamond$*enabled* $\wedge$ $\neg$*taken*))
unfolding *strongly-fair-def* **by** (*metis weakly-fair-def weakly-fair-def2*)

lemma *strongly-fair-def3*:
strongly-fair enabled taken = ($\Box\Diamond$*enabled* $\longrightarrow$ $\Box\Diamond$*taken*)
unfolding *strongly-fair-def2* **by** (*metis* (*full-types*) *always-eventually-always weakly-fair-def2 weakly-fair-def3*)

lemma *always-strongly-fair*:
$\Box$(*strongly-fair enabled taken*) = *strongly-fair enabled taken*
unfolding *strongly-fair-def* **by** *simp*

lemma *eventually-strongly-fair*:
$\Diamond$(*strongly-fair enabled taken*) = *strongly-fair enabled taken*
unfolding *strongly-fair-def2* **by** (*simp add*: *always-eventually-always*)

lemma *strongly-fair-disj-distrib*: — not true for *weakly-fair*
strongly-fair (*enabled1* $\vee$ *enabled2*) *taken* = (*strongly-fair enabled1 taken* $\wedge$ *strongly-fair enabled2 taken*)
unfolding *strongly-fair-def defs*
apply (*auto simp*: *fun-eq-iff*)
apply *blast*
apply *blast*
apply (*metis* (*full-types*) *semiring-normalization-rules*(*25*))
done

lemma *strongly-fair-imp-weakly-fair*:
assumes *strongly-fair enabled taken* σ
shows *weakly-fair enabled taken* σ
using *assms* **unfolding** *strongly-fair-def3 weakly-fair-def3* **by** (*simp add*: *eventually-always-imp-always-eventually*)

lemma *always-enabled-weakly-fair-strongly-fair*:
assumes ($\Box$*enabled*) σ
shows *weakly-fair enabled taken* σ = *strongly-fair enabled taken* σ
using *assms* **by** (*metis strongly-fair-def3 strongly-fair-imp-weakly-fair unfold*(*2*) *weakly-fair-def3*)

3.3 Safety and liveness

Sistla (1994) shows some characterisations of LTL formulas in terms of safety and liveness. Note his ($\mathcal{U}$) is actually ($\mathcal{W}$).

See also Chang, Manna, and Pnueli (1992).

lemma *safety-state-prop*:
shows *safety* $\lceil P \rceil$
unfolding *defs* **by** (*rule safety-state-prop*)

lemma *safety-Next*:
assumes *safety P*
shows *safety* ($\bigcirc P$)
using *assms* **unfolding** *defs safety-def*
apply *clarsimp*
apply (*metis* (*mono-tags*) *One-nat-def list.sel*(*3*) *nat.simps*(*3*) *stake.simps*(*2*))
done

lemma *safety-unless*:
assumes *safety P*
assumes *safety Q*
shows *safety* ($P\ \mathcal{W}\ Q$)
proof(*rule safetyI2*)
fix σ **assume** X: $\exists\beta$. ($P\ \mathcal{W}\ Q$) (*stake i* σ @− β) **for** *i*

then show $(P\ \mathcal{W}\ Q)\ \sigma$
proof(*cases* $\forall\, i\ j.\ \exists\, \beta.\ P\ (\sigma(i \to j)\ @\!-\ \beta)$)
case *True*
with ‹*safety P*› **have** $\forall\, i.\ P\ (\sigma\ |_s\ i)$ **unfolding** *safety-def2* **by** *blast*
then show *?thesis* **by** (*blast intro*: *unless-alwaysI*)
next
case *False*
then obtain $k\ k'$ **where** $\forall\, \beta.\ \neg\ P\ (\sigma(k \to k')\ @\!-\ \beta)$ **by** *clarsimp*
then have $\forall\, i\ u.\ k + k' \le i \longrightarrow \neg P\ ((stake\ i\ \sigma\ @\!-\ u)\ |_s\ k)$
by (*metis add.commute diff-add stake-shift-stake-shift stake-suffix-drop suffix-shift*)
then have $\forall\, i\ u.\ k + k' \le i \wedge (P\ \mathcal{W}\ Q)\ (stake\ i\ \sigma\ @\!-\ u) \longrightarrow (\exists\, m{\le}k.\ Q\ ((stake\ i\ \sigma\ @\!-\ u)\ |_s\ m) \wedge (\forall\, p{<}m.\ P\ ((stake\ i\ \sigma\ @\!-\ u)\ |_s\ p)))$
unfolding *defs* **using** *leI* **by** *blast*
then have $\forall\, i\ u.\ k + k' \le i \wedge (P\ \mathcal{W}\ Q)\ (stake\ i\ \sigma\ @\!-\ u) \longrightarrow (\exists\, m{\le}k.\ Q\ (\sigma(m \to i - m)\ @\!-\ u) \wedge (\forall\, p{<}m.\ P\ (\sigma(p \to i - p)\ @\!-\ u)))$
by (*metis stake-suffix add-leE nat-less-le order-trans*)
then have $\forall\, i.\ \exists\, n{\ge}i.\ \exists\, m{\le}k.\ \exists\, u.\ Q\ (\sigma(m \to n - m)\ @\!-\ u) \wedge (\forall\, p{<}m.\ P\ (\sigma(p \to n - p)\ @\!-\ u))$
using *X* **by** (*metis add.commute le-add1*)
then have $\exists\, m{\le}k.\ \forall\, i.\ \exists\, n{\ge}i.\ \exists\, u.\ Q\ (\sigma(m \to n - m)\ @\!-\ u) \wedge (\forall\, p{<}m.\ P\ (\sigma(p \to n - p)\ @\!-\ u))$
by (*simp add*: *always-eventually-pigeonhole*)
with ‹*safety P*› ‹*safety Q*› **show** $(P\ \mathcal{W}\ Q)\ \sigma$
unfolding *defs* **by** (*metis Nat.le-diff-conv2 add-leE safety-always-eventually*)
qed
qed

lemma *safety-always*:
assumes *safety P*
shows *safety* $(\Box P)$
using *assms* **by** (*metis norm*(*20*) *safety-def safety-unless*)

lemma *absolute-liveness-eventually*:
shows *absolute-liveness* $P \longleftrightarrow (\exists\, \sigma.\ P\ \sigma) \wedge P = \Diamond P$
unfolding *absolute-liveness-def defs*
by (*metis cancel-comm-monoid-add-class.diff-cancel drop-eq-Nil order-refl shift.simps*(*1*) *stake-suffix-id suffix-shift suffix-zero*)

lemma *stable-always*:
shows *stable* $P \longleftrightarrow (\exists\, \sigma.\ P\ \sigma) \wedge P = \Box P$
unfolding *stable-def defs* **by** (*metis suffix-zero*)

To show that *weakly-fair* is a *fairness* property requires some constraints on *enabled* and *taken*:

- it is reasonable to assume they are state formulas
- *taken* must be satisfiable

lemma *fairness-weakly-fair*:
assumes $\exists\, s.\ taken\ s$
shows *fairness* (*weakly-fair* ⌈*enabled*⌉ ⌈*taken*⌉)
unfolding *fairness-def stable-def absolute-liveness-def weakly-fair-def*
using *assms*
apply *auto*
apply (*rule-tac* $x{=}\lambda\text{-}\ .s$ **in** *exI*)
apply *fastforce*
apply (*simp add*: *alwaysD*)
apply (*rule-tac* $x{=}\lambda\text{-}\ .s$ **in** *exI*)
apply *fastforce*
apply (*metis* (*full-types*) *absolute-liveness-def absolute-liveness-eventually eventually-weakly-fair weakly-fair-def*)
done

lemma *fairness-strongly-fair*:
  assumes ∃ *s. taken s*
  shows *fairness* (*strongly-fair* ⌈*enabled*⌉ ⌈*taken*⌉)
unfolding *fairness-def stable-def absolute-liveness-def strongly-fair-def*
using *assms*
apply *auto*
   apply (*rule-tac x*=λ- .*s* **in** *exI*)
   apply *fastforce*
  apply (*simp add*: *alwaysD*)
 apply (*rule-tac x*=λ- .*s* **in** *exI*)
 apply *fastforce*
apply (*metis* (*full-types*) *absolute-liveness-def absolute-liveness-eventually eventually-weakly-fair weakly-fair-def*)
done

4 CIMP syntax and semantics

We define a small sequential programming language with synchronous message passing primitives for describing the individual processes. This has the advantage over raw transition systems in that it is programmer-readable, includes sequential composition, supports a program logic and VCG (§5.1), etc. These processes are composed in parallel at the top-level.

CIMP is inspired by IMP, as presented by Winskel (1993) and Nipkow and Klein (2014), and the classical process algebras CCS (Milner 1980, 1989) and CSP (Hoare 1985). Note that the algebraic properties of this language have not been developed.

As we operate in a concurrent setting, we need to provide a small-step semantics (§4.2), which we give in the style of *structural operational semantics* (SOS) as popularised by Plotkin (2004). The semantics of a complete system (§4.3) is presently taken simply to be the states reachable by interleaving the enabled steps of the individual processes, subject to message passing rendezvous. We leave a trace or branching semantics to future work.

This theory contains all the trusted definitions. The soundness of the other theories supervenes upon this one.

4.1 Syntax

Programs are represented using an explicit (deep embedding) of their syntax, as the semantics needs to track the progress of multiple threads of control. Each (atomic) *basic command* (§**??**) is annotated with a *'location*, which we use in our assertions (§4.4). These locations need not be unique, though in practice they likely will be.

Processes maintain *local states* of type *'state*. These can be updated with arbitrary relations of *'state* ⇒ *'state set* with *LocalOp*, and conditions of type *'s* ⇒ *bool* are similarly shallowly embedded. This arrangement allows the end-user to select their own level of atomicity.

The sequential composition operator and control constructs are standard. We add the infinite looping construct *Loop* so we can construct single-state reactive systems; this has implications for fairness assertions.

type-synonym *'s bexp* = *'s* ⇒ *bool*

datatype (*'answer*, *'location*, *'question*, *'state*) *com*
  = *Request* *'location 'state* ⇒ *'question 'answer* ⇒ *'state* ⇒ *'state set* (‹{-} *Request* - -› [*0*, *70*, *70*] *71*)
  | *Response* *'location 'question* ⇒ *'state* ⇒ (*'state* × *'answer*) *set* (‹{-} *Response* -› [*0*, *70*] *71*)
  | *LocalOp* *'location 'state* ⇒ *'state set* (‹{-} *LocalOp* -› [*0*, *70*] *71*)
  | *Cond1* *'location 'state bexp* (*'answer*, *'location*, *'question*, *'state*) *com* (‹{-} *IF* - *THEN* - *FI*› [*0*, *0*, *0*] *71*)
  | *Cond2* *'location 'state bexp* (*'answer*, *'location*, *'question*, *'state*) *com*
        (*'answer*, *'location*, *'question*, *'state*) *com* (‹{-} *IF* -/ *THEN* -/ *ELSE* -/ *FI*› [*0*, *0*, *0*, *0*] *71*)
  | *Loop* (*'answer*, *'location*, *'question*, *'state*) *com* (‹*LOOP DO* -/ *OD*› [*0*] *71*)
  | *While* *'location 'state bexp* (*'answer*, *'location*, *'question*, *'state*) *com* (‹{-} *WHILE* -/ *DO* -/ *OD*› [*0*, *0*, *0*] *71*)
  | *Seq* (*'answer*, *'location*, *'question*, *'state*) *com*
        (*'answer*, *'location*, *'question*, *'state*) *com* (**infixr** ‹;;› *69*)

| *Choose* ('*answer*, '*location*, '*question*, '*state*) *com*
('*answer*, '*location*, '*question*, '*state*) *com* (**infixl** ‹⊕› *68*)

We provide a one-armed conditional as it is the common form and avoids the need to discover a label for an internal *SKIP* and/or trickier proofs about the VCG.

In contrast to classical process algebras, we have local state and distinct request and response actions. These provide an interface to Isabelle/HOL's datatypes that avoids the need for binding (ala the π-calculus of Milner (1989)) or large non-deterministic sums (ala CCS (Milner 1980, §2.8)). Intuitively the requester poses a '*question* with a *Request* command, which upon rendezvous with a responder's *Response* command receives an '*answer*. The '*question* is a deterministic function of the requester's local state, whereas responses can be non-deterministic. Note that CIMP does not provide a notion of channel; these can be modelled by a judicious choice of '*question*.

We also provide a binary external choice operator (⊕) (infix (⊕)). Internal choice can be recovered in combination with local operations (see Milner (1980, §2.3)).

We abbreviate some common commands: *SKIP* is a local operation that does nothing, and the floor brackets simplify deterministic *LocalOp*s. We also adopt some syntax magic from Makarius's *Hoare* and *Multiquote* theories in the Isabelle/HOL distribution.

abbreviation *SKIP-syn* (‹{|-|}/ *SKIP*› [*0*] *71*) **where**
{|*l*|} *SKIP* ≡ {|*l*|} *LocalOp* (λ*s*. {*s*})

abbreviation (*input*) *DetLocalOp* :: '*location* ⇒ ('*state* ⇒ '*state*)
⇒ ('*answer*, '*location*, '*question*, '*state*) *com* (‹{|-|} ⌊-⌋› [*0*, *0*] *71*) **where**
{|*l*|} ⌊*f*⌋ ≡ {|*l*|} *LocalOp* (λ*s*. {*f s*})

syntax
-quote :: '*b* ⇒ ('*a* ⇒ '*b*) (‹«-»› [*0*] *1000*)
-antiquote :: ('*a* ⇒ '*b*) ⇒ '*b* (‹´-› [*1000*] *1000*)
-Assign :: '*location* ⇒ *idt* ⇒ '*b* ⇒ ('*answer*, '*location*, '*question*, '*state*) *com* (‹({|-|} ´- :=/ -)› [*0*, *0*, *70*] *71*)
-NonDetAssign :: '*location* ⇒ *idt* ⇒ '*b set* ⇒ ('*answer*, '*location*, '*question*, '*state*) *com* (‹({|-|} ´- :∈/ -)› [*0*, *0*, *70*] *71*)

abbreviation (*input*) *NonDetAssign* :: '*location* ⇒ (('*val* ⇒ '*val*) ⇒ '*state* ⇒ '*state*) ⇒ ('*state* ⇒ '*val set*)
⇒ ('*answer*, '*location*, '*question*, '*state*) *com* **where**
NonDetAssign l upd es ≡ {|*l*|} *LocalOp* (λ*s*. { *upd* ⟨*e*⟩ *s* |*e*. *e* ∈ *es s* })

translations
{|*l*|} ´*x* := *e* => *CONST DetLocalOp l* «´(*-update-name x* (λ-. *e*))»
{|*l*|} ´*x* :∈ *es* => *CONST NonDetAssign l* (*-update-name x*) «*es*»

parse-translation ‹
```
let
  fun antiquote-tr i (Const (@{syntax-const -antiquote}, -) $
        (t as Const (@{syntax-const -antiquote}, -) $ -)) = skip-antiquote-tr i t
    | antiquote-tr i (Const (@{syntax-const -antiquote}, -) $ t) =
        antiquote-tr i t $ Bound i
    | antiquote-tr i (t $ u) = antiquote-tr i t $ antiquote-tr i u
    | antiquote-tr i (Abs (x, T, t)) = Abs (x, T, antiquote-tr (i + 1) t)
    | antiquote-tr - a = a
  and skip-antiquote-tr i ((c as Const (@{syntax-const -antiquote}, -)) $ t) =
        c $ skip-antiquote-tr i t
    | skip-antiquote-tr i t = antiquote-tr i t;

  fun quote-tr [t] = Abs (s, dummyT, antiquote-tr 0 (Term.incr-boundvars 1 t))
    | quote-tr ts = raise TERM (quote-tr, ts);
in [(@{syntax-const -quote}, K quote-tr)] end
```
›

4.2 Process semantics

Here we define the semantics of a single process's program. We begin by defining the type of externally-visible behaviour:

datatype (*'answer*, *'question*) *seq-label*
  = *sl-Internal* (‹τ›)
  | *sl-Send* *'question* *'answer* (‹«-, -»›)
  | *sl-Receive* *'question* *'answer* (‹»-, -«›)

We define a *labelled transition system* (an LTS) using an execution-stack style of semantics that avoids special treatment of the *SKIP*s introduced by a traditional small step semantics (such as Winskel (1993, Chapter 14)) when a basic command is executed. This was suggested by Thomas Sewell; Pitts (2002) gave a semantics to an ML-like language using this approach.

We record the location of the command that was executed to support fairness constraints.

type-synonym (*'answer*, *'location*, *'question*, *'state*) *local-state*
  = (*'answer*, *'location*, *'question*, *'state*) *com list* × *'location option* × *'state*

inductive
  small-step :: (*'answer*, *'location*, *'question*, *'state*) *local-state*
    ⇒ (*'answer*, *'question*) *seq-label*
    ⇒ (*'answer*, *'location*, *'question*, *'state*) *local-state* ⇒ *bool* (‹- →- -› [55, 0, 56] 55)
where
  ⟦ α = *action s*; $s' \in$ *val* β *s* ⟧ ⟹ (⦃*l*⦄ *Request action val* # *cs*, -, *s*) $\rightarrow_{«\alpha,\ \beta»}$ (*cs*, *Some l*, s')
| (s', β) ∈ *action* α *s* ⟹ (⦃*l*⦄ *Response action* # *cs*, -, *s*) $\rightarrow_{»\alpha,\ \beta«}$ (*cs*, *Some l*, s')

| $s' \in R\ s$ ⟹ (⦃*l*⦄ *LocalOp R* # *cs*, -, *s*) $\rightarrow_\tau$ (*cs*, *Some l*, s')

| *b s* ⟹ (⦃*l*⦄ *IF b THEN c FI* # *cs*, -, *s*) $\rightarrow_\tau$ (*c* # *cs*, *Some l*, *s*)
| ¬*b s* ⟹ (⦃*l*⦄ *IF b THEN c FI* # *cs*, -, *s*) $\rightarrow_\tau$ (*cs*, *Some l*, *s*)

| *b s* ⟹ (⦃*l*⦄ *IF b THEN c1 ELSE c2 FI* # *cs*, -, *s*) $\rightarrow_\tau$ (*c1* # *cs*, *Some l*, *s*)
| ¬*b s* ⟹ (⦃*l*⦄ *IF b THEN c1 ELSE c2 FI* # *cs*, -, *s*) $\rightarrow_\tau$ (*c2* # *cs*, *Some l*, *s*)

| (*c* # *LOOP DO c OD* # *cs*, *s*) $\rightarrow_\alpha$ (cs', s') ⟹ (*LOOP DO c OD* # *cs*, *s*) $\rightarrow_\alpha$ (cs', s')

| *b s* ⟹ (⦃*l*⦄ *WHILE b DO c OD* # *cs*, -, *s*) $\rightarrow_\tau$ (*c* # ⦃*l*⦄ *WHILE b DO c OD* # *cs*, *Some l*, *s*)
| ¬ *b s* ⟹ (⦃*l*⦄ *WHILE b DO c OD* # *cs*, -, *s*) $\rightarrow_\tau$ (*cs*, *Some l*, *s*)

| (*c1* # *c2* # *cs*, *s*) $\rightarrow_\alpha$ (cs', s') ⟹ (*c1*;; *c2* # *cs*, *s*) $\rightarrow_\alpha$ (cs', s')

| *Choose1*: (*c1* # *cs*, *s*) $\rightarrow_\alpha$ (cs', s') ⟹ (*c1* ⊕ *c2* # *cs*, *s*) $\rightarrow_\alpha$ (cs', s')
| *Choose2*: (*c2* # *cs*, *s*) $\rightarrow_\alpha$ (cs', s') ⟹ (*c1* ⊕ *c2* # *cs*, *s*) $\rightarrow_\alpha$ (cs', s')

The following projections operate on local states. These should not appear to the end-user.

abbreviation *cPGM* :: (*'answer*, *'location*, *'question*, *'state*) *local-state* ⇒ (*'answer*, *'location*, *'question*, *'state*) *com list* **where**
  cPGM ≡ *fst*

abbreviation *cTKN* :: (*'answer*, *'location*, *'question*, *'state*) *local-state* ⇒ *'location option* **where**
  cTKN s ≡ *fst* (*snd s*)

abbreviation *cLST* :: (*'answer*, *'location*, *'question*, *'state*) *local-state* ⇒ *'state* **where**
  cLST s ≡ *snd* (*snd s*)

4.3 System steps

A global state maps process names to process' local states. One might hope to allow processes to have distinct types of local state, but there remains no good solution yet in a simply-typed setting; see Schirmer and Wenzel

(2009).

type-synonym $('answer, 'location, 'proc, 'question, 'state)$ *global-state*
$= 'proc \Rightarrow ('answer, 'location, 'question, 'state)$ *local-state*

type-synonym $('proc, 'state)$ *local-states*
$= 'proc \Rightarrow 'state$

An execution step of the overall system is either any enabled internal τ step of any process, or a communication rendezvous between two processes. For the latter to occur, a *Request* action must be enabled in process *p1*, and a *Response* action in (distinct) process *p2*, where the request/response labels α and β (semantically) match.

We also track global communication history here to support assertional reasoning (see §5).

type-synonym $('answer, 'question)$ *event* $= 'question \times 'answer$
type-synonym $('answer, 'question)$ *history* $= ('answer, 'question)$ *event list*

record $('answer, 'location, 'proc, 'question, 'state)$ *system-state* =
GST :: $('answer, 'location, 'proc, 'question, 'state)$ *global-state*
HST :: $('answer, 'question)$ *history*

inductive — This is a predicate of the current state, so the successor state comes first.
system-step :: *'proc set*
$\Rightarrow ('answer, 'location, 'proc, 'question, 'state)$ *system-state*
$\Rightarrow ('answer, 'location, 'proc, 'question, 'state)$ *system-state*
$\Rightarrow$ *bool*
where
LocalStep: $[\![\ GST\ sh\ p \rightarrow_\tau ls';\ GST\ sh' = (GST\ sh)(p := ls');\ HST\ sh' = HST\ sh\ ]\!] \Longrightarrow system\text{-}step\ \{p\}\ sh'\ sh$
| *CommunicationStep*: $[\![\ GST\ sh\ p \rightarrow_{«\alpha,\ \beta»} ls1';\ GST\ sh\ q \rightarrow_{»\alpha,\ \beta«} ls2';\ p \neq q;$
$GST\ sh' = (GST\ sh)(p := ls1',\ q := ls2');\ HST\ sh' = HST\ sh\ @\ [(\alpha,\ \beta)]\ ]\!] \Longrightarrow system\text{-}step$
$\{p,\ q\}\ sh'\ sh$

In classical process algebras matching communication actions yield τ steps, which aids nested parallel composition and the restriction operation (Milner 1980, §2.2). As CIMP does not provide either we do not need to hide communication labels. In CCS/CSP it is not clear how one reasons about the communication history, and it seems that assertional reasoning about these languages is not well developed.

We define predicates over communication histories and system states. These are uncurried to ease composition.

type-synonym $('answer, 'location, 'proc, 'question, 'state)$ *state-pred*
$= ('answer, 'location, 'proc, 'question, 'state)$ *system-state* $\Rightarrow$ *bool*

The *LST* operator (written as a postfix $\downarrow$) projects the local states of the processes from a $('answer, 'location, 'proc, 'question, 'state)$ *system-state*, i.e. it discards control location information.

Conversely the *LSTP* operator lifts predicates over local states into predicates over $('answer, 'location, 'proc, 'question, 'state)$ *system-state*.

Predicates that do not depend on control locations were termed *universal assertions* by Levin and Gries (1981, §3.6).

type-synonym $('proc, 'state)$ *local-state-pred*
$= ('proc, 'state)$ *local-states* $\Rightarrow$ *bool*

definition *LST* :: $('answer, 'location, 'proc, 'question, 'state)$ *system-state*
$\Rightarrow ('proc, 'state)$ *local-states* (‹-↓› [*1000*] *1000*) **where**
$s\downarrow = cLST \circ GST\ s$

abbreviation (*input*) *LSTP* :: $('proc, 'state)$ *local-state-pred*
$\Rightarrow ('answer, 'location, 'proc, 'question, 'state)$ *state-pred* **where**
$LSTP\ P \equiv \lambda s.\ P\ s\downarrow$

4.4 Control predicates

Following Lamport (1980)[1], we define the *at* predicate, which holds of a process when control resides at that location. Due to non-determinism processes can be *at* a set of locations; it is more like "a statement with this location is enabled", which incidentally handles non-unique locations. Lamport's language is deterministic, so he doesn't have this problem. This also allows him to develop a stronger theory about his control predicates.

type-synonym *'location label* = *'location set*

primrec
  atC :: (*'answer*, *'location*, *'question*, *'state*) *com* ⇒ *'location label*
where
  atC (⦃*l*⦄ *Request action val*) = {*l*}
| *atC* (⦃*l*⦄ *Response action*) = {*l*}
| *atC* (⦃*l*⦄ *LocalOp f*) = {*l*}
| *atC* (⦃*l*⦄ *IF - THEN - FI*) = {*l*}
| *atC* (⦃*l*⦄ *IF - THEN - ELSE - FI*) = {*l*}
| *atC* (⦃*l*⦄ *WHILE - DO - OD*) = {*l*}
| *atC* (*LOOP DO c OD*) = *atC c*
| *atC* (*c1*;; *c2*) = *atC c1*
| *atC* (*c1* ⊕ *c2*) = *atC c1* ∪ *atC c2*

primrec *atCs* :: (*'answer*, *'location*, *'question*, *'state*) *com list* ⇒ *'location label* **where**
  atCs [] = {}
| *atCs* (*c* # -) = *atC c*

We provide the following definitions to the end-user.

AT maps process names to a predicate that is true of locations where control for that process resides, and the abbreviation *at* provides a conventional way to use it. The constant *atS* specifies that control for process *p* resides at one of the given locations. This stands in for, and generalises, the *in* predicate of Lamport (1980).

definition *AT* :: (*'answer*, *'location*, *'proc*, *'question*, *'state*) *system-state* ⇒ *'proc* ⇒ *'location label* **where**
  AT s p = *atCs* (*cPGM* (*GST s p*))

abbreviation *at* :: *'proc* ⇒ *'location* ⇒ (*'answer*, *'location*, *'proc*, *'question*, *'state*) *state-pred* **where**
  at p l s ≡ *l* ∈ *AT s p*

definition *atS* :: *'proc* ⇒ *'location set* ⇒ (*'answer*, *'location*, *'proc*, *'question*, *'state*) *state-pred* **where**
  atS p ls s = (∃ *l*∈*ls*. *at p l s*)

definition *atLs* :: *'proc* ⇒ *'location label set* ⇒ (*'answer*, *'location*, *'proc*, *'question*, *'state*) *state-pred* **where**
  atLs p labels s = (*AT s p* ∈ *labels*)

abbreviation (*input*) *atL* :: *'proc* ⇒ *'location label* ⇒ (*'answer*, *'location*, *'proc*, *'question*, *'state*) *state-pred* **where**
  atL p label ≡ *atLs p* {*label*}

definition *atPLs* :: (*'proc* × *'location label*) *set* ⇒ (*'answer*, *'location*, *'proc*, *'question*, *'state*) *state-pred* **where**
  atPLs pls = (∀ *p label*. ⟨(*p*, *label*) ∈ *pls*⟩ ⟶ *atL p label*)

The constant *taken* provides a way of identifying which transition was taken. It is somewhat like Lamport's *after*, but not quite due to the presence of non-determinism here. This does not work well for invariants or preconditions.

definition *taken* :: *'proc* ⇒ *'location* ⇒ (*'answer*, *'location*, *'proc*, *'question*, *'state*) *state-pred* **where**
  taken p l s ⟷ *cTKN* (*GST s p*) = *Some l*

[1] Manna and Pnueli (1995) also develop a theory of locations. I think Lamport attributes control predicates to Owicki in her PhD thesis (under Gries). I did not find a treatment of procedures. Manna and Pnueli (1991) observe that a notation for making assertions over sets of locations reduces clutter significantly.

A process is terminated if it not at any control location.

abbreviation (*input*) *terminated* :: *'proc* ⇒ (*'answer*, *'location*, *'proc*, *'question*, *'state*) *state-pred* **where**
terminated p ≡ *atL p* {}

A complete system consists of one program per process, and a (global) constraint on their initial local states. From these we can construct the set of initial global states and all those reachable by system steps (§4.3).

type-synonym (*'answer*, *'location*, *'proc*, *'question*, *'state*) *programs*
= *'proc* ⇒ (*'answer*, *'location*, *'question*, *'state*) *com*

record (*'answer*, *'location*, *'proc*, *'question*, *'state*) *pre-system* =
PGMs :: (*'answer*, *'location*, *'proc*, *'question*, *'state*) *programs*
INIT :: (*'proc*, *'state*) *local-state-pred*

definition
initial-state :: (*'answer*, *'location*, *'proc*, *'question*, *'state*, *'ext*) *pre-system-ext*
⇒ (*'answer*, *'location*, *'proc*, *'question*, *'state*) *global-state*
⇒ *bool*
where
initial-state sys s = ((∀ *p*. *cPGM* (*s p*) = ⌊*PGMs sys p*⌋ ∧ *cTKN* (*s p*) = *None*) ∧ *INIT sys* (*cLST* ∘ *s*))

We construct infinite runs of a system by allowing stuttering, i.e., arbitrary repetitions of states following Lamport (2002, Chapter 8), by taking the reflexive closure of the *system-step* relation. Therefore terminated programs infinitely repeat their final state (but note our definition of terminated processes in §4.4).

Some accounts define stuttering as the *finite* repetition of states. With or without this constraint *prerun* contains *junk* in the form of unfair runs, where particular processes do not progress.

definition
system-step-reflclp :: (*'answer*, *'location*, *'proc*, *'question*, *'state*) *system-state seq-pred*
where
system-step-reflclp σ ⟷ (λ*sh sh'*. ∃ *pls*. *system-step pls sh' sh*)$^{==}$ (σ *0*) (σ *1*)

definition
prerun :: (*'answer*, *'location*, *'proc*, *'question*, *'state*, *'ext*) *pre-system-ext*
⇒ (*'answer*, *'location*, *'proc*, *'question*, *'state*) *system-state seq-pred*
where
prerun sys = ((λσ. *initial-state sys* (*GST* (σ *0*)) ∧ *HST* (σ *0*) = []) ∧ □*system-step-reflclp*)

definition — state-based invariants only
prerun-valid :: (*'answer*, *'location*, *'proc*, *'question*, *'state*, *'ext*) *pre-system-ext*
⇒ (*'answer*, *'location*, *'proc*, *'question*, *'state*) *state-pred* ⇒ *bool* (‹- $\models_{pre}$ -› [*11*, *0*] *11*)
where
(*sys* $\models_{pre}$ φ) ⟷ (∀ σ. *prerun sys* σ ⟶ (□⌈φ⌉) σ)

A *run* of a system is a *prerun* that satisfies the *FAIR* requirement. Typically this would include *weak fairness* for every transition of every process.

record (*'answer*, *'location*, *'proc*, *'question*, *'state*) *system* =
(*'answer*, *'location*, *'proc*, *'question*, *'state*) *pre-system*
\+ *FAIR* :: (*'answer*, *'location*, *'proc*, *'question*, *'state*) *system-state seq-pred*

definition
run :: (*'answer*, *'location*, *'proc*, *'question*, *'state*) *system*
⇒ (*'answer*, *'location*, *'proc*, *'question*, *'state*) *system-state seq-pred*
where
run sys = (*prerun sys* ∧ *FAIR sys*)

definition
valid :: (*'answer*, *'location*, *'proc*, *'question*, *'state*) *system*
⇒ (*'answer*, *'location*, *'proc*, *'question*, *'state*) *system-state seq-pred* ⇒ *bool* (‹- ⊨ -› [*11*, *0*] *11*)

where
$(sys \models \varphi) \longleftrightarrow (\forall \sigma.\ run\ sys\ \sigma \longrightarrow \varphi\ \sigma)$

5 State-based invariants

We provide a simple-minded verification condition generator (VCG) for this language, providing support for establishing state-based invariants. It is just one way of reasoning about CIMP programs and is proven sound wrt to the CIMP semantics.

Our approach follows Lamport (1980); Lamport and Schneider (1984) (and the later Lamport (2002)) and closely related work by Apt, Francez, and de Roever (1980), Cousot and Cousot (1980) and Levin and Gries (1981), who suggest the incorporation of a history variable. Cousot and Cousot (1980) apparently contains a completeness proof. Lamport mentions that this technique was well-known in the mid-80s when he proposed the use of prophecy variables[2]. See also de Roever, de Boer, Hannemann, Hooman, Lakhnech, Poel, and Zwiers (2001) for an extended discussion of some of this.

declare *small-step.intros*[*intro*]

inductive-cases *small-step-inv*:
$(\{\!\!\{l\}\!\!\}\ Request\ action\ val\ \#\ cs,\ ls) \rightarrow_a s'$
$(\{\!\!\{l\}\!\!\}\ Response\ action\ \#\ cs,\ ls) \rightarrow_a s'$
$(\{\!\!\{l\}\!\!\}\ LocalOp\ R\ \#\ cs,\ ls) \rightarrow_a s'$
$(\{\!\!\{l\}\!\!\}\ IF\ b\ THEN\ c\ FI\ \#\ cs,\ ls) \rightarrow_a s'$
$(\{\!\!\{l\}\!\!\}\ IF\ b\ THEN\ c1\ ELSE\ c2\ FI\ \#\ cs,\ ls) \rightarrow_a s'$
$(\{\!\!\{l\}\!\!\}\ WHILE\ b\ DO\ c\ OD\ \#\ cs,\ ls) \rightarrow_a s'$
$(LOOP\ DO\ c\ OD\ \#\ cs,\ ls) \rightarrow_a s'$

lemma *small-step-stuck*:
$\neg\ ([],\ s) \rightarrow_\alpha c'$
by (*auto elim*: *small-step.cases*)

declare *system-step.intros*[*intro*]

By default we ask the simplifier to rewrite *atS* using ambient *AT* information.

lemma *atS-state-weak-cong*[*cong*]:
$AT\ s\ p = AT\ s'\ p \Longrightarrow atS\ p\ ls\ s \longleftrightarrow atS\ p\ ls\ s'$
by (*auto simp*: *atS-def*)

We provide an incomplete set of basic rules for label sets.

lemma *atS-simps*:
$\neg atS\ p\ \{\}\ s$
$atS\ p\ \{l\}\ s \longleftrightarrow at\ p\ l\ s$
$[\![at\ p\ l\ s;\ l \in ls]\!] \Longrightarrow atS\ p\ ls\ s$
$(\forall\, l.\ at\ p\ l\ s \longrightarrow l \notin ls) \Longrightarrow \neg atS\ p\ ls\ s$
by (*auto simp*: *atS-def*)

lemma *atS-mono*:
$[\![atS\ p\ ls\ s;\ ls \subseteq ls']\!] \Longrightarrow atS\ p\ ls'\ s$
by (*auto simp*: *atS-def*)

lemma *atS-un*:
$atS\ p\ (l \cup l')\ s \longleftrightarrow atS\ p\ l\ s \vee atS\ p\ l'\ s$
by (*auto simp*: *atS-def*)

lemma *atLs-disj-union*[*simp*]:
$(atLs\ p\ label0 \vee atLs\ p\ label1) = atLs\ p\ (label0 \cup label1)$
unfolding *atLs-def* **by** *simp*

[2] https://lamport.azurewebsites.net/pubs/pubs.html

lemma *atLs-insert-disj*:
  $atLs\ p\ (insert\ l\ label0) = (atL\ p\ l \lor atLs\ p\ label0)$
by *simp*

lemma *small-step-terminated*:
  $s \rightarrow_x s' \Longrightarrow atCs\ (fst\ s) = \{\} \Longrightarrow atCs\ (fst\ s') = \{\}$
by (*induct pred*: *small-step*) *auto*

lemma *atC-not-empty*:
  $atC\ c \neq \{\}$
by (*induct c*) *auto*

lemma *atCs-empty*:
  $atCs\ cs = \{\} \longleftrightarrow cs = []$
by (*induct cs*) (*auto simp*: *atC-not-empty*)

lemma *terminated-no-commands*:
  assumes *terminated p sh*
  shows $\exists s.\ GST\ sh\ p = ([], s)$
using *assms* **unfolding** *atLs-def AT-def* **by** (*metis atCs-empty prod.collapse singletonD*)

lemma *terminated-GST-stable*:
  assumes *system-step q sh′ sh*
  assumes *terminated p sh*
  shows $GST\ sh\ p = GST\ sh'\ p$
using *assms* **by** (*auto dest*!: *terminated-no-commands simp*: *small-step-stuck elim*!: *system-step.cases*)

lemma *terminated-stable*:
  assumes *system-step q sh′ sh*
  assumes *terminated p sh*
  shows *terminated p sh′*
using *assms* **unfolding** *atLs-def AT-def*
by (*fastforce split*: *if-splits prod.splits*
    dest: *small-step-terminated*
    elim!: *system-step.cases*)

lemma *system-step-pls-nonempty*:
  assumes *system-step pls sh′ sh*
  shows $pls \neq \{\}$
using *assms* **by** *cases simp-all*

lemma *system-step-no-change*:
  assumes *system-step ps sh′ sh*
  assumes $p \notin ps$
  shows $GST\ sh'\ p = GST\ sh\ p$
using *assms* **by** *cases simp-all*

lemma *initial-stateD*:
  assumes *initial-state sys s*
  shows $AT\ ((\!|GST = s,\ HST = []|\!)) = atC \circ PGMs\ sys \land INIT\ sys\ ((\!|GST = s,\ HST = []|\!))\!\downarrow \land (\forall p\ l.\ \neg taken\ p\ l\ (\!|GST = s,\ HST = []|\!))$
using *assms* **unfolding** *initial-state-def split-def o-def LST-def AT-def taken-def* **by** *simp*

lemma *initial-states-initial*[*iff*]:
  assumes *initial-state sys s*
  shows $at\ p\ l\ ((\!|GST = s,\ HST = []|\!)) \longleftrightarrow l \in atC\ (PGMs\ sys\ p)$
using *assms* **unfolding** *initial-state-def split-def AT-def* **by** *simp*

```
definition
  reachable-state :: ('answer, 'location, 'proc, 'question, 'state, 'ext) pre-system-ext
                    ⇒ ('answer, 'location, 'proc, 'question, 'state) state-pred
where
  reachable-state sys s ⟷ (∃σ i. prerun sys σ ∧ σ i = s)

lemma reachable-stateE:
  assumes reachable-state sys sh
  assumes ⋀σ i. prerun sys σ ⟹ P (σ i)
  shows P sh
using assms unfolding reachable-state-def by blast

lemma prerun-reachable-state:
  assumes prerun sys σ
  shows reachable-state sys (σ i)
using assms unfolding prerun-def LTL.defs system-step-reflclp-def reachable-state-def by auto

lemma reachable-state-induct[consumes 1, case-names init LocalStep CommunicationStep, induct set: reach-
able-state]:
  assumes r: reachable-state sys sh
  assumes i: ⋀s. initial-state sys s ⟹ P (|GST = s, HST = []|)
  assumes l: ⋀sh ls' p. ⟦reachable-state sys sh; P sh; GST sh p →τ ls'⟧ ⟹ P (|GST = (GST sh)(p := ls'), HST
= HST sh|)
  assumes c: ⋀sh ls1' ls2' p1 p2 α β.
                ⟦reachable-state sys sh; P sh;
                GST sh p1 →«α, β» ls1'; GST sh p2 →»α, β« ls2'; p1 ≠ p2 ⟧
                  ⟹ P (|GST = (GST sh)(p1 := ls1', p2 := ls2'), HST = HST sh @ [(α, β)]|)
  shows P sh
using r
proof(rule reachable-stateE)
  fix σ i assume prerun sys σ show P (σ i)
  proof(induct i)
    case 0 from ‹prerun sys σ› show ?case
      unfolding prerun-def by (metis (full-types) i old.unit.exhaust system-state.surjective)
  next
    case (Suc i) with ‹prerun sys σ› show ?case
unfolding prerun-def LTL.defs system-step-reflclp-def reachable-state-def
apply clarsimp
apply (drule-tac x=i in spec)
apply (erule disjE; clarsimp)
apply (erule system-step.cases; clarsimp)
 apply (metis (full-types) ‹prerun sys σ› l old.unit.exhaust prerun-reachable-state system-state.surjective)
apply (metis (full-types) ‹prerun sys σ› c old.unit.exhaust prerun-reachable-state system-state.surjective)
done
  qed
qed

lemma prerun-valid-TrueI:
  shows sys ⊨pre ⟨True⟩
unfolding prerun-valid-def by simp

lemma prerun-valid-conjI:
  assumes sys ⊨pre P
  assumes sys ⊨pre Q
  shows sys ⊨pre P ∧ Q
using assms unfolding prerun-valid-def always-def by simp
```

lemma *valid-prerun-lift*:
  assumes $sys \models_{pre} I$
  shows $sys \models \Box\lceil I \rceil$
using *assms* **unfolding** *prerun-valid-def valid-def run-def* **by** *blast*

lemma *prerun-valid-induct*:
  assumes $\bigwedge\sigma.\ prerun\ sys\ \sigma \Longrightarrow \lceil I \rceil\ \sigma$
  assumes $\bigwedge\sigma.\ prerun\ sys\ \sigma \Longrightarrow (\lceil I \rceil \hookrightarrow (\bigcirc\lceil I \rceil))\ \sigma$
  shows $sys \models_{pre} I$
unfolding *prerun-valid-def* **using** *assms* **by** (*simp add*: *always-induct*)

lemma *prerun-validI*:
  assumes $\bigwedge s.\ reachable\text{-}state\ sys\ s \Longrightarrow I\ s$
  shows $sys \models_{pre} I$
unfolding *prerun-valid-def* **using** *assms* **by** (*simp add*: *alwaysI prerun-reachable-state*)

lemma *prerun-validE*:
  assumes *reachable-state sys s*
  assumes $sys \models_{pre} I$
  shows *I s*
using *assms* **unfolding** *prerun-valid-def*
by (*metis alwaysE reachable-stateE suffix-state-prop*)

5.0.1 Relating reachable states to the initial programs

To usefully reason about the control locations presumably embedded in the single global invariant, we need to link the programs we have in reachable state *s* to the programs in the initial states. The *fragments* function decomposes the program into statements that can be directly executed (§**??**). We also compute the locations we could be at after executing that statement as a function of the process's local state.

Eliding the bodies of *IF* and *WHILE* statements yields smaller (but equivalent) proof obligations.

type-synonym (*'answer*, *'location*, *'question*, *'state*) *loc-comp*
  = *'state* ⇒ *'location set*

fun *lconst* :: *'location set* ⇒ (*'answer*, *'location*, *'question*, *'state*) *loc-comp* **where**
  lconst lp s = *lp*

definition *lcond* :: *'location set* ⇒ *'location set* ⇒ *'state bexp*
    ⇒ (*'answer*, *'location*, *'question*, *'state*) *loc-comp* **where**
  lcond lp lp' b s = (*if b s then lp else lp'*)

lemma *lcond-split*:
  $Q\ (lcond\ lp\ lp'\ b\ s) \longleftrightarrow (b\ s \longrightarrow Q\ lp) \wedge (\neg b\ s \longrightarrow Q\ lp')$
unfolding *lcond-def* **by** (*simp split*: *if-splits*)

lemma *lcond-split-asm*:
  $Q\ (lcond\ lp\ lp'\ b\ s) \longleftrightarrow \neg\ ((b\ s \wedge \neg Q\ lp) \vee (\neg b\ s \wedge \neg\ Q\ lp'))$
unfolding *lcond-def* **by** (*simp split*: *if-splits*)

lemmas *lcond-splits* = *lcond-split lcond-split-asm*

fun
  fragments :: (*'answer*, *'location*, *'question*, *'state*) *com*
      ⇒ *'location set*
      ⇒ ((*'answer*, *'location*, *'question*, *'state*) *com*
       × (*'answer*, *'location*, *'question*, *'state*) *loc-comp*) *set*
where
  fragments ({|*l*|} *IF b THEN c FI*) *aft*

= { ({|l|} IF b THEN c′ FI, lcond (atC c) aft b) |c′. True }
∪ fragments c aft
| fragments ({|l|} IF b THEN c1 ELSE c2 FI) aft
= { ({|l|} IF b THEN c1′ ELSE c2′ FI, lcond (atC c1) (atC c2) b) |c1′ c2′. True }
∪ fragments c1 aft ∪ fragments c2 aft
| fragments (LOOP DO c OD) aft = fragments c (atC c)
| fragments ({|l|} WHILE b DO c OD) aft
= fragments c {l} ∪ { ({|l|} WHILE b DO c′ OD, lcond (atC c) aft b) |c′. True }
| fragments (c1;; c2) aft = fragments c1 (atC c2) ∪ fragments c2 aft
| fragments (c1 ⊕ c2) aft = fragments c1 aft ∪ fragments c2 aft
| fragments c aft = { (c, lconst aft) }

fun
fragmentsL :: ('answer, 'location, 'question, 'state) com list
⇒ (('answer, 'location, 'question, 'state) com
× ('answer, 'location, 'question, 'state) loc-comp) set
where
fragmentsL [] = {}
| fragmentsL [c] = fragments c {}
| fragmentsL (c # c′ # cs) = fragments c (atC c′) ∪ fragmentsL (c′ # cs)

abbreviation
fragmentsLS :: ('answer, 'location, 'question, 'state) local-state
⇒ (('answer, 'location, 'question, 'state) com
× ('answer, 'location, 'question, 'state) loc-comp) set
where
fragmentsLS s ≡ fragmentsL (cPGM s)

We show that taking system steps preserves fragments.

lemma small-step-fragmentsLS:
assumes $s \rightarrow_\alpha s'$
shows fragmentsLS s′ ⊆ fragmentsLS s
using assms **by** induct (case-tac [!] cs, auto)

lemma reachable-state-fragmentsLS:
assumes reachable-state sys sh
shows fragmentsLS (GST sh p) ⊆ fragments (PGMs sys p) {}
using assms
by (induct rule: reachable-state-induct)
(auto simp: initial-state-def dest: subsetD[OF small-step-fragmentsLS])

inductive
basic-com :: ('answer, 'location, 'question, 'state) com ⇒ bool
where
basic-com ({|l|} Request action val)
| basic-com ({|l|} Response action)
| basic-com ({|l|} LocalOp R)
| basic-com ({|l|} IF b THEN c FI)
| basic-com ({|l|} IF b THEN c1 ELSE c2 FI)
| basic-com ({|l|} WHILE b DO c OD)

lemma fragments-basic-com:
assumes (c′, aft′) ∈ fragments c aft
shows basic-com c′
using assms **by** (induct c arbitrary: aft) (auto intro: basic-com.intros)

lemma fragmentsL-basic-com:
assumes (c′, aft′) ∈ fragmentsL cs

shows *basic-com c'*
using *assms*
apply (*induct cs*)
 apply *simp*
apply (*case-tac cs*)
 apply (*auto simp*: *fragments-basic-com*)
done

To reason about system transitions we need to identify which basic statement gets executed next. To that end we factor out the recursive cases of the *small-step* semantics into *contexts*, which isolate the *basic-com* commands with immediate externally-visible behaviour. Note that non-determinism means that more than one *basic-com* can be enabled at a time.

The representation of evaluation contexts follows Berghofer (2012). This style of operational semantics was originated by Felleisen and Hieb (1992).

type-synonym (*'answer*, *'location*, *'question*, *'state*) *ctxt*
  = ((*'answer*, *'location*, *'question*, *'state*) *com* ⇒ (*'answer*, *'location*, *'question*, *'state*) *com*)
   × ((*'answer*, *'location*, *'question*, *'state*) *com* ⇒ (*'answer*, *'location*, *'question*, *'state*) *com list*)

inductive-set
  ctxt :: (*'answer*, *'location*, *'question*, *'state*) *ctxt set*
where
  C-Hole: (*id*, ⟨[]⟩) ∈ *ctxt*
| *C-Loop*: (*E*, *fctxt*) ∈ *ctxt* ⟹ (λ*c1*. *LOOP DO E c1 OD*, λ*c1*. *fctxt c1* @ [*LOOP DO E c1 OD*]) ∈ *ctxt*
| *C-Seq*: (*E*, *fctxt*) ∈ *ctxt* ⟹ (λ*c1*. *E c1*;; *c2*, λ*c1*. *fctxt c1* @ [*c2*]) ∈ *ctxt*
| *C-Choose1*: (*E*, *fctxt*) ∈ *ctxt* ⟹ (λ*c1*. *E c1* ⊕ *c2*, *fctxt*) ∈ *ctxt*
| *C-Choose2*: (*E*, *fctxt*) ∈ *ctxt* ⟹ (λ*c2*. *c1* ⊕ *E c2*, *fctxt*) ∈ *ctxt*

We can decompose a small step into a context and a *basic-com*.

fun
  decompose-com :: (*'answer*, *'location*, *'question*, *'state*) *com*
      ⇒ ((*'answer*, *'location*, *'question*, *'state*) *com*
       × (*'answer*, *'location*, *'question*, *'state*) *ctxt*) *set*
where
  decompose-com (*LOOP DO c1 OD*) = { (*c*, λ*t*. *LOOP DO ictxt t OD*, λ*t*. *fctxt t* @ [*LOOP DO ictxt t OD*]) |*c fctxt ictxt*. (*c*, *ictxt*, *fctxt*) ∈ *decompose-com c1* }
| *decompose-com* (*c1*;; *c2*) = { (*c*, λ*t*. *ictxt t*;; *c2*, λ*t*. *fctxt t* @ [*c2*]) |*c fctxt ictxt*. (*c*, *ictxt*, *fctxt*) ∈ *decompose-com c1* }
| *decompose-com* (*c1* ⊕ *c2*) = { (*c*, λ*t*. *ictxt t* ⊕ *c2*, *fctxt*) |*c fctxt ictxt*. (*c*, *ictxt*, *fctxt*) ∈ *decompose-com c1* }
      ∪ { (*c*, λ*t*. *c1* ⊕ *ictxt t*, *fctxt*) |*c fctxt ictxt*. (*c*, *ictxt*, *fctxt*) ∈ *decompose-com c2* }
| *decompose-com c* = {(*c*, *id*, ⟨[]⟩)}

definition
  decomposeLS :: (*'answer*, *'location*, *'question*, *'state*) *local-state*
    ⇒ ((*'answer*, *'location*, *'question*, *'state*) *com*
     × ((*'answer*, *'location*, *'question*, *'state*) *com* ⇒ (*'answer*, *'location*, *'question*, *'state*) *com*)
     × ((*'answer*, *'location*, *'question*, *'state*) *com* ⇒ (*'answer*, *'location*, *'question*, *'state*) *com list*)) *set*
where
  decomposeLS s = (*case cPGM s of c* # - ⇒ *decompose-com c* | - ⇒ {})

lemma *ctxt-inj*:
  assumes (*E*, *fctxt*) ∈ *ctxt*
  assumes *E x* = *E y*
  shows *x* = *y*
using *assms* **by** (*induct set*: *ctxt*) *auto*

lemma *decompose-com-non-empty*: *decompose-com c* ≠ {}
by (*induct c*) *auto*

```
lemma decompose-com-basic-com:
  assumes (c′, ctxts) ∈ decompose-com c
  shows basic-com c′
using assms by (induct c arbitrary: c′ ctxts) (auto intro: basic-com.intros)

lemma decomposeLS-basic-com:
  assumes (c′, ctxts) ∈ decomposeLS s
  shows basic-com c′
using assms unfolding decomposeLS-def by (simp add: decompose-com-basic-com split: list.splits)

lemma decompose-com-ctxt:
  assumes (c′, ctxts) ∈ decompose-com c
  shows ctxts ∈ ctxt
using assms by (induct c arbitrary: c′ ctxts) (auto intro: ctxt.intros)

lemma decompose-com-ictxt:
  assumes (c′, ictxt, fctxt) ∈ decompose-com c
  shows ictxt c′ = c
using assms by (induct c arbitrary: c′ ictxt fctxt) auto

lemma decompose-com-small-step:
  assumes as: (c′ # fctxt c′ @ cs, s) →α s′
  assumes ds: (c′, ictxt, fctxt) ∈ decompose-com c
  shows (c # cs, s) →α s′
using decompose-com-ctxt[OF ds] as decompose-com-ictxt[OF ds]
by (induct ictxt fctxt arbitrary: c cs)
   (cases s′, fastforce simp: fun-eq-iff dest: ctxt-inj)+

theorem context-decompose:
  s →α s′ ⟷ (∃(c, ictxt, fctxt) ∈ decomposeLS s.
                 cPGM s = ictxt c # tl (cPGM s)
               ∧ (c # fctxt c @ tl (cPGM s), cTKN s, cLST s) →α s′
               ∧ (∀ l∈atC c. cTKN s′ = Some l)) (is ?lhs = ?rhs)
proof(rule iffI)
  assume ?lhs then show ?rhs
  unfolding decomposeLS-def
  proof(induct rule: small-step.induct)
    case (Choose1 c1 cs s α cs′ s′ c2) then show ?case
      apply clarsimp
      apply (rename-tac c ictxt fctxt)
      apply (rule-tac x=(c, λt. ictxt t ⊕ c2, fctxt) in bexI)
      apply auto
      done
  next
    case (Choose2 c2 cs s α cs′ s′ c1) then show ?case
      apply clarsimp
      apply (rename-tac c ictxt fctxt)
      apply (rule-tac x=(c, λt. c1 ⊕ ictxt t, fctxt) in bexI)
      apply auto
      done
  qed fastforce+
next
  assume ?rhs then show ?lhs
    unfolding decomposeLS-def
    by (cases s) (auto split: list.splits dest: decompose-com-small-step)
qed
```

While we only use this result left-to-right (to decompose a small step into a basic one), this equivalence shows that we lose no information in doing so.

Decomposing a compound command preserves *fragments* too.

fun
  loc-compC :: (*'answer*, *'location*, *'question*, *'state*) *com*
    ⇒ (*'answer*, *'location*, *'question*, *'state*) *com list*
    ⇒ (*'answer*, *'location*, *'question*, *'state*) *loc-comp*
where
  loc-compC (⦃*l*⦄ *IF b THEN c FI*) *cs* = *lcond* (*atC c*) (*atCs cs*) *b*
| *loc-compC* (⦃*l*⦄ *IF b THEN c1 ELSE c2 FI*) *cs* = *lcond* (*atC c1*) (*atC c2*) *b*
| *loc-compC* (*LOOP DO c OD*) *cs* = *lconst* (*atC c*)
| *loc-compC* (⦃*l*⦄ *WHILE b DO c OD*) *cs* = *lcond* (*atC c*) (*atCs cs*) *b*
| *loc-compC c cs* = *lconst* (*atCs cs*)

lemma *decompose-fragments*:
  assumes (*c*, *ictxt*, *fctxt*) ∈ *decompose-com c0*
  shows (*c*, *loc-compC c* (*fctxt c* @ *cs*)) ∈ *fragments c0* (*atCs cs*)
using *assms*
proof(*induct c0 arbitrary*: *c ictxt fctxt cs*)
  case (*Loop c01 c ictxt fctxt cs*)
  from *Loop.prems Loop.hyps*(*1*)[**where** *cs*=*ictxt c* # *cs*] **show** *?case* **by** (*auto simp*: *decompose-com-ictxt*)
next
  case (*Seq c01 c02 c ictxt fctxt cs*)
  from *Seq.prems Seq.hyps*(*1*)[**where** *cs*=*c02* # *cs*] **show** *?case* **by** *auto*
qed *auto*

lemma *at-decompose*:
  assumes (*c*, *ictxt*, *fctxt*) ∈ *decompose-com c0*
  shows *atC c* ⊆ *atC c0*
using *assms* **by** (*induct c0 arbitrary*: *c ictxt fctxt*; *fastforce*)

lemma *at-decomposeLS*:
  assumes (*c*, *ictxt*, *fctxt*) ∈ *decomposeLS s*
  shows *atC c* ⊆ *atCs* (*cPGM s*)
using *assms* **unfolding** *decomposeLS-def* **by** (*auto simp*: *at-decompose split*: *list.splits*)

lemma *decomposeLS-fragmentsLS*:
  assumes (*c*, *ictxt*, *fctxt*) ∈ *decomposeLS s*
  shows (*c*, *loc-compC c* (*fctxt c* @ *tl* (*cPGM s*))) ∈ *fragmentsLS s*
using *assms*
proof(*cases cPGM s*)
  case (*Cons d ds*)
  with *assms decompose-fragments*[**where** *cs*=*ds*] **show** *?thesis*
    by (*cases ds*) (*auto simp*: *decomposeLS-def*)
qed (*simp add*: *decomposeLS-def*)

lemma *small-step-loc-compC*:
  assumes *basic-com c*
  assumes (*c* # *cs*, *ls*) $\rightarrow_\alpha$ *ls'*
  shows *loc-compC c cs* (*snd ls*) = *atCs* (*cPGM ls'*)
using *assms* **by** (*fastforce elim*: *basic-com.cases elim*!: *small-step-inv split*: *lcond-splits*)

The headline result allows us to constrain the initial and final states of a given small step in terms of the original programs, provided the initial state is reachable.

theorem *decompose-small-step*:
  assumes *GST sh p* $\rightarrow_\alpha$ *ps'*
  assumes *reachable-state sys sh*
  obtains *c cs aft*

where $(c,\ aft) \in fragments\ (PGMs\ sys\ p)\ \{\}$
and $atC\ c \subseteq atCs\ (cPGM\ (GST\ sh\ p))$
and $aft\ (cLST\ (GST\ sh\ p)) = atCs\ (cPGM\ ps')$
and $(c\ \#\ cs,\ cTKN\ (GST\ sh\ p),\ cLST\ (GST\ sh\ p)) \rightarrow_{\alpha} ps'$
and $\forall\, l{\in}atC\ c.\ cTKN\ ps' = Some\ l$
using *assms*
apply –
apply (*frule iffD1*[*OF context-decompose*])
apply *clarsimp*
apply (*frule decomposeLS-fragmentsLS*)
apply (*frule at-decomposeLS*)
apply (*frule* (*1*) *subsetD*[*OF reachable-state-fragmentsLS*])
apply (*frule decomposeLS-basic-com*)
apply (*frule* (*1*) *small-step-loc-compC*)
apply *simp*
done

Reasoning by induction over the reachable states with *decompose-small-step* is quite tedious. We provide a very simple VCG that generates friendlier local proof obligations in §5.1.

5.1 Simple-minded Hoare Logic/VCG for CIMP

We do not develop a proper Hoare logic or full VCG for CIMP: this machinery merely packages up the subgoals that arise from induction over the reachable states (§5). This is somewhat in the spirit of Ridge (2009).

Note that this approach is not compositional: it consults the original system to find matching communicating pairs, and *aft* tracks the labels of possible successor statements. More serious Hoare logics are provided by Cousot and Cousot (1989); Lamport (1980); Lamport and Schneider (1984).

Intuitively we need to discharge a proof obligation for either *Request*s or *Response*s but not both. Here we choose to focus on *Request*s as we expect to have more local information available about these.

inductive
$vcg :: ('answer,\ 'location,\ 'proc,\ 'question,\ 'state)\ programs$
$\Rightarrow\ 'proc$
$\Rightarrow ('answer,\ 'location,\ 'question,\ 'state)\ loc\text{-}comp$
$\Rightarrow ('answer,\ 'location,\ 'proc,\ 'question,\ 'state)\ state\text{-}pred$
$\Rightarrow ('answer,\ 'location,\ 'question,\ 'state)\ com$
$\Rightarrow ('answer,\ 'location,\ 'proc,\ 'question,\ 'state)\ state\text{-}pred$
$\Rightarrow bool$ (‹-, -, - ⊢/ ⦃-⦄/ -/ ⦃-⦄› [*11,0,0,0,0,0*] *11*)
where
$[\![\ \bigwedge aft'\ action'\ s\ ps'\ p's'\ l'\ \beta\ s'\ p'.$
$[\![\ pre\ s;\ (\{\!|l'|\!\}\ Response\ action',\ aft') \in fragments\ (coms\ p')\ \{\};\ p \neq p';$
$ps' \in val\ \beta\ (s{\downarrow}\ p);\ (p's',\ \beta) \in action'\ (action\ (s{\downarrow}\ p))\ (s{\downarrow}\ p');$
$at\ p\ l\ s;\ at\ p'\ l'\ s;$
$AT\ s' = (AT\ s)(p := aft\ (s{\downarrow}\ p),\ p' := aft'\ (s{\downarrow}\ p'));$
$s'{\downarrow} = s{\downarrow}(p := ps',\ p' := p's');$
$taken\ p\ l\ s';$
$HST\ s' = HST\ s\ @\ [(action\ (s{\downarrow}\ p),\ \beta)];$
$\forall\, p''{\in}{-}\{p,p'\}.\ GST\ s'\ p'' = GST\ s\ p''$
$]\!] \Longrightarrow post\ s'$
$]\!] \Longrightarrow coms,\ p,\ aft \vdash \{\!|pre|\!\}\ \{\!|l|\!\}\ Request\ action\ val\ \{\!|post|\!\}$
$|\ [\![\ \bigwedge s\ ps'\ s'.$
$[\![\ pre\ s;\ ps' \in f\ (s{\downarrow}\ p);$
$at\ p\ l\ s;$
$AT\ s' = (AT\ s)(p := aft\ (s{\downarrow}\ p));$
$s'{\downarrow} = s{\downarrow}(p := ps');$
$taken\ p\ l\ s';$
$HST\ s' = HST\ s;$
$\forall\, p''{\in}{-}\{p\}.\ GST\ s'\ p'' = GST\ s\ p''$

$\rrbracket \Longrightarrow$ *post s'*
$\rrbracket \Longrightarrow$ *coms, p, aft* $\vdash$ $\{\!|pre|\!\}$ $\{\!|l|\!\}$ *LocalOp f* $\{\!|post|\!\}$
| $\llbracket \bigwedge s\ s'.$
$\llbracket$ *pre s*;
at p l s;
AT s' = (*AT s*)(*p* := *aft* (*s*↓ *p*));
s'↓ = *s*↓;
taken p l s';
HST s' = *HST s*;
$\forall p''{\in}{-}\{p\}.$ *GST s' p''* = *GST s p''*
$\rrbracket \Longrightarrow$ *post s'*
$\rrbracket \Longrightarrow$ *coms, p, aft* $\vdash$ $\{\!|pre|\!\}$ $\{\!|l|\!\}$ *IF b THEN t FI* $\{\!|post|\!\}$
| $\llbracket \bigwedge s\ s'.$
$\llbracket$ *pre s*;
at p l s;
AT s' = (*AT s*)(*p* := *aft* (*s*↓ *p*));
s'↓ = *s*↓;
taken p l s';
HST s' = *HST s*;
$\forall p''{\in}{-}\{p\}.$ *GST s' p''* = *GST s p''*
$\rrbracket \Longrightarrow$ *post s'*
$\rrbracket \Longrightarrow$ *coms, p, aft* $\vdash$ $\{\!|pre|\!\}$ $\{\!|l|\!\}$ *IF b THEN t ELSE e FI* $\{\!|post|\!\}$
| $\llbracket \bigwedge s\ s'.$
$\llbracket$ *pre s*;
at p l s;
AT s' = (*AT s*)(*p* := *aft* (*s*↓ *p*));
s'↓ = *s*↓;
taken p l s';
HST s' = *HST s*;
$\forall p''{\in}{-}\{p\}.$ *GST s' p''* = *GST s p''*
$\rrbracket \Longrightarrow$ *post s'*
$\rrbracket \Longrightarrow$ *coms, p, aft* $\vdash$ $\{\!|pre|\!\}$ $\{\!|l|\!\}$ *WHILE b DO c OD* $\{\!|post|\!\}$
— There are no proof obligations for the following commands, but including them makes some basic rules hold (§5.1.1):
| *coms, p, aft* $\vdash$ $\{\!|pre|\!\}$ $\{\!|l|\!\}$ *Response action* $\{\!|post|\!\}$
| *coms, p, aft* $\vdash$ $\{\!|pre|\!\}$ *c1* ;; *c2* $\{\!|post|\!\}$
| *coms, p, aft* $\vdash$ $\{\!|pre|\!\}$ *LOOP DO c OD* $\{\!|post|\!\}$
| *coms, p, aft* $\vdash$ $\{\!|pre|\!\}$ *c1* $\oplus$ *c2* $\{\!|post|\!\}$

We abbreviate invariance with one-sided validity syntax.

abbreviation *valid-inv* (‹-, -, - ⊢/ {|-|}/ -› [*11,0,0,0,0*] *11*) **where**
coms, p, aft $\vdash$ $\{\!|I|\!\}$ *c* $\equiv$ *coms, p, aft* $\vdash$ $\{\!|I|\!\}$ *c* $\{\!|I|\!\}$

inductive-cases *vcg-inv*:
coms, p, aft $\vdash$ $\{\!|pre|\!\}$ $\{\!|l|\!\}$ *Request action val* $\{\!|post|\!\}$
coms, p, aft $\vdash$ $\{\!|pre|\!\}$ $\{\!|l|\!\}$ *LocalOp f* $\{\!|post|\!\}$
coms, p, aft $\vdash$ $\{\!|pre|\!\}$ $\{\!|l|\!\}$ *IF b THEN t FI* $\{\!|post|\!\}$
coms, p, aft $\vdash$ $\{\!|pre|\!\}$ $\{\!|l|\!\}$ *IF b THEN t ELSE e FI* $\{\!|post|\!\}$
coms, p, aft $\vdash$ $\{\!|pre|\!\}$ $\{\!|l|\!\}$ *WHILE b DO c OD* $\{\!|post|\!\}$
coms, p, aft $\vdash$ $\{\!|pre|\!\}$ *LOOP DO c OD* $\{\!|post|\!\}$
coms, p, aft $\vdash$ $\{\!|pre|\!\}$ $\{\!|l|\!\}$ *Response action* $\{\!|post|\!\}$
coms, p, aft $\vdash$ $\{\!|pre|\!\}$ *c1* ;; *c2* $\{\!|post|\!\}$
coms, p, aft $\vdash$ $\{\!|pre|\!\}$ *Choose c1 c2* $\{\!|post|\!\}$

We tweak *fragments* by omitting *Response*s, yielding fewer obligations

fun
vcg-fragments' :: (*'answer*, *'location*, *'question*, *'state*) *com*
$\Rightarrow$ *'location set*

⇒ (('answer, 'location, 'question, 'state) com
  × ('answer, 'location, 'question, 'state) loc-comp) set
where
  vcg-fragments' (⦃l⦄ *Response action*) *aft* = {}
| *vcg-fragments'* (⦃l⦄ *IF b THEN c FI*) *aft*
    = *vcg-fragments' c aft*
    ∪ { (⦃l⦄ *IF b THEN c' FI*, *lcond* (*atC c*) *aft b*) |*c'*. *True* }
| *vcg-fragments'* (⦃l⦄ *IF b THEN c1 ELSE c2 FI*) *aft*
    = *vcg-fragments' c2 aft* ∪ *vcg-fragments' c1 aft*
    ∪ { (⦃l⦄ *IF b THEN c1' ELSE c2' FI*, *lcond* (*atC c1*) (*atC c2*) *b*) |*c1' c2'*. *True* }
| *vcg-fragments'* (*LOOP DO c OD*) *aft* = *vcg-fragments' c* (*atC c*)
| *vcg-fragments'* (⦃l⦄ *WHILE b DO c OD*) *aft*
    = *vcg-fragments' c* {*l*} ∪ { (⦃l⦄ *WHILE b DO c' OD*, *lcond* (*atC c*) *aft b*) |*c'*. *True* }
| *vcg-fragments'* (*c1* ;; *c2*) *aft* = *vcg-fragments' c2 aft* ∪ *vcg-fragments' c1* (*atC c2*)
| *vcg-fragments'* (*c1* ⊕ *c2*) *aft* = *vcg-fragments' c1 aft* ∪ *vcg-fragments' c2 aft*
| *vcg-fragments' c aft* = {(*c*, *lconst aft*)}

abbreviation
  vcg-fragments :: ('answer, 'location, 'question, 'state) com
    ⇒ (('answer, 'location, 'question, 'state) com
      × ('answer, 'location, 'question, 'state) loc-comp) set
where
  vcg-fragments c ≡ *vcg-fragments' c* {}

fun *isResponse* :: ('answer, 'location, 'question, 'state) com ⇒ bool **where**
  isResponse (⦃l⦄ *Response action*) ⟷ *True*
| *isResponse* - ⟷ *False*

lemma *fragments-vcg-fragments'*:
  ⟦ (*c*, *aft*) ∈ *fragments c' aft'*; ¬*isResponse c* ⟧ ⟹ (*c*, *aft*) ∈ *vcg-fragments' c' aft'*
by (*induct c' arbitrary*: *aft'*) *auto*

lemma *vcg-fragments'-fragments*:
  vcg-fragments' c' aft' ⊆ *fragments c' aft'*
by (*induct c' arbitrary*: *aft'*) (*auto 10 0*)

lemma *VCG-step*:
  assumes *V*: ⋀*p*. ∀(*c*, *aft*) ∈ *vcg-fragments* (*PGMs sys p*). *PGMs sys*, *p*, *aft* ⊢ ⦃*pre*⦄ *c* ⦃*post*⦄
  assumes *S*: *system-step p sh' sh*
  assumes *R*: *reachable-state sys sh*
  assumes *P*: *pre sh*
  shows *post sh'*
using *S*
proof *cases*
  case *LocalStep* **with** *P* **show** *?thesis*
    apply −
    apply (*erule decompose-small-step*[*OF - R*])
    apply (*frule fragments-basic-com*)
    apply (*erule basic-com.cases*)
    apply (*fastforce dest*!: *fragments-vcg-fragments' V*[*rule-format*]
        elim: *vcg-inv elim*!: *small-step-inv*
        simp: *LST-def AT-def taken-def fun-eq-iff*)+
    done
next
  case *CommunicationStep* **with** *P* **show** *?thesis*
    apply −
    apply (*erule decompose-small-step*[*OF - R*])
    apply (*erule decompose-small-step*[*OF - R*])

```
    subgoal for c cs aft c' cs' aft'
    apply (frule fragments-basic-com[where c'=c])
    apply (frule fragments-basic-com[where c'=c'])
    apply (elim basic-com.cases; clarsimp elim!: small-step-inv)
    apply (drule fragments-vcg-fragments')
    apply (fastforce dest!: V[rule-format]
                     elim: vcg-inv elim!: small-step-inv
                     simp: LST-def AT-def taken-def fun-eq-iff)+
    done
    done
qed
```

The user sees the conclusion of V for each element of *vcg-fragments*.

```
lemma VCG-step-inv-stable:
  assumes V: ⋀p. ∀(c, aft) ∈ vcg-fragments (PGMs sys p). PGMs sys, p, aft ⊢ ⦃I⦄ c
  assumes prerun sys σ
  shows (⌈I⌉ ↪ ○⌈I⌉) σ
apply (rule alwaysI)
apply clarsimp
apply (rule nextI)
apply clarsimp
using assms(2) unfolding prerun-def
apply clarsimp
apply (erule-tac i=i in alwaysE)
unfolding system-step-reflclp-def
apply clarsimp
apply (erule disjE; clarsimp)
using VCG-step[where pre=I and post=I] V assms(2) prerun-reachable-state
apply blast
done

lemma VCG:
  assumes I: ∀s. initial-state sys s ⟶ I (⦇GST = s, HST = []⦈)
  assumes V: ⋀p. ∀(c, aft) ∈ vcg-fragments (PGMs sys p). PGMs sys, p, aft ⊢ ⦃I⦄ c
  shows sys ⊨pre I
apply (rule prerun-valid-induct)
 apply (clarsimp simp: prerun-def state-prop-def)
 apply (metis (full-types) I old.unit.exhaust system-state.surjective)
using VCG-step-inv-stable[OF V] apply blast
done

lemmas VCG-valid = valid-prerun-lift[OF VCG, of sys I] for sys I
```

5.1.1 VCG rules

We can develop some (but not all) of the familiar Hoare rules; see Lamport (1980) and the seL4/l4.verified lemma buckets for inspiration. We avoid many of the issues Lamport mentions as we only treat basic (atomic) commands.

```
context
  fixes coms :: ('answer, 'location, 'proc, 'question, 'state) programs
  fixes p :: 'proc
  fixes aft :: ('answer, 'location, 'question, 'state) loc-comp
begin

abbreviation
  valid-syn :: ('answer, 'location, 'proc, 'question, 'state) state-pred
           ⇒ ('answer, 'location, 'question, 'state) com
           ⇒ ('answer, 'location, 'proc, 'question, 'state) state-pred ⇒ bool where
  valid-syn P c Q ≡ coms, p, aft ⊢ ⦃P⦄ c ⦃Q⦄
```

notation *valid-syn* (‹⦃-⦄/ -/ ⦃-⦄›)

abbreviation
valid-inv-syn :: (*'answer*, *'location*, *'proc*, *'question*, *'state*) *state-pred*
⇒ (*'answer*, *'location*, *'question*, *'state*) *com* ⇒ *bool* **where**
valid-inv-syn P c ≡ ⦃P⦄ c ⦃P⦄
notation *valid-inv-syn* (‹⦃-⦄/ -›)

lemma *vcg-True*:
⦃P⦄ c ⦃⟨*True*⟩⦄
by (*cases c*) (*fastforce elim*!: *vcg-inv intro*: *vcg.intros*)+

lemma *vcg-conj*:
⟦ ⦃I⦄ c ⦃Q⦄; ⦃I⦄ c ⦃R⦄ ⟧ ⟹ ⦃I⦄ c ⦃$Q \wedge R$⦄
by (*cases c*) (*fastforce elim*!: *vcg-inv intro*: *vcg.intros*)+

lemma *vcg-pre-imp*:
⟦ ⋀*s*. *P s* ⟹ *Q s*; ⦃Q⦄ c ⦃R⦄ ⟧ ⟹ ⦃P⦄ c ⦃R⦄
by (*cases c*) (*fastforce elim*!: *vcg-inv intro*: *vcg.intros*)+

lemmas *vcg-pre* = *vcg-pre-imp*[*rotated*]

lemma *vcg-post-imp*:
⟦ ⋀*s*. *Q s* ⟹ *R s*; ⦃P⦄ c ⦃Q⦄ ⟧ ⟹ ⦃P⦄ c ⦃R⦄
by (*cases c*) (*fastforce elim*!: *vcg-inv intro*: *vcg.intros*)+

lemma *vcg-prop*[*intro*]:
⦃⟨P⟩⦄ c
by (*cases c*) (*fastforce intro*: *vcg.intros*)+

lemma *vcg-drop-imp*:
assumes ⦃P⦄ c ⦃Q⦄
shows ⦃P⦄ c ⦃$R \longrightarrow Q$⦄
using *assms*
by (*cases c*) (*fastforce elim*!: *vcg-inv intro*: *vcg.intros*)+

lemma *vcg-conj-lift*:
assumes *x*: ⦃P⦄ c ⦃Q⦄
assumes *y*: ⦃P'⦄ c ⦃Q'⦄
shows ⦃$P \wedge P'$⦄ c ⦃$Q \wedge Q'$⦄
apply (*rule vcg-conj*)
apply (*rule vcg-pre*[*OF x*], *simp*)
apply (*rule vcg-pre*[*OF y*], *simp*)
done

lemma *vcg-disj-lift*:
assumes *x*: ⦃P⦄ c ⦃Q⦄
assumes *y*: ⦃P'⦄ c ⦃Q'⦄
shows ⦃$P \vee P'$⦄ c ⦃$Q \vee Q'$⦄
using *assms*
by (*cases c*) (*fastforce elim*!: *vcg-inv intro*: *vcg.intros*)+

lemma *vcg-imp-lift*:
assumes ⦃P'⦄ c ⦃$\neg P$⦄
assumes ⦃Q'⦄ c ⦃Q⦄
shows ⦃$P' \vee Q'$⦄ c ⦃$P \longrightarrow Q$⦄
by (*simp only*: *imp-conv-disj vcg-disj-lift*[*OF assms*])

lemma *vcg-ex-lift*:
  assumes $\bigwedge x.\ \{\!|P\ x|\!\}\ c\ \{\!|Q\ x|\!\}$
  shows $\{\!|\lambda s.\ \exists\, x.\ P\ x\ s|\!\}\ c\ \{\!|\lambda s.\ \exists\, x.\ Q\ x\ s|\!\}$
using *assms*
by (*cases c*) (*fastforce elim*!: *vcg-inv intro*: *vcg.intros*)+

lemma *vcg-all-lift*:
  assumes $\bigwedge x.\ \{\!|P\ x|\!\}\ c\ \{\!|Q\ x|\!\}$
  shows $\{\!|\lambda s.\ \forall\, x.\ P\ x\ s|\!\}\ c\ \{\!|\lambda s.\ \forall\, x.\ Q\ x\ s|\!\}$
using *assms*
by (*cases c*) (*fastforce elim*!: *vcg-inv intro*: *vcg.intros*)+

lemma *vcg-name-pre-state*:
  assumes $\bigwedge s.\ P\ s \Longrightarrow \{\!|(=)\ s|\!\}\ c\ \{\!|Q|\!\}$
  shows $\{\!|P|\!\}\ c\ \{\!|Q|\!\}$
using *assms*
by (*cases c*) (*fastforce elim*!: *vcg-inv intro*: *vcg.intros*)+

lemma *vcg-lift-comp*:
  assumes *f*: $\bigwedge P.\ \{\!|\lambda s.\ P\ (f\ s :: 'a :: type)|\!\}\ c$
  assumes *P*: $\bigwedge x.\ \{\!|Q\ x|\!\}\ c\ \{\!|P\ x|\!\}$
  shows $\{\!|\lambda s.\ Q\ (f\ s)\ s|\!\}\ c\ \{\!|\lambda s.\ P\ (f\ s)\ s|\!\}$
apply (*rule vcg-name-pre-state*)
apply (*rename-tac s*)
apply (*rule vcg-pre*)
 apply (*rule vcg-post-imp*[*rotated*])
  apply (*rule vcg-conj-lift*)
   apply (*rule-tac x*=*f s* **in** *P*)
  apply (*rule-tac P*=λ*fs*. *fs* = *f s* **in** *f*)
 apply *simp*
apply *simp*
done

5.1.2 Cheap non-interference rules

These rules magically construct VCG lifting rules from the easier to prove *eq-imp* facts. We don't actually use these in the GC, but we do derive *fun-upd* equations using the same mechanism. Thanks to Thomas Sewell for the requisite syntax magic.

As these *eq-imp* facts do not usefully compose, we make the definition asymmetric (i.e., *g* does not get a bundle of parameters).

Note that these are effectively parametricity rules.

definition *eq-imp* :: $('a \Rightarrow 'b \Rightarrow 'c) \Rightarrow ('b \Rightarrow 'e) \Rightarrow bool$ **where**
  $eq\text{-}imp\ f\ g \equiv (\forall\, s\ s'.\ (\forall\, x.\ f\ x\ s = f\ x\ s') \longrightarrow (g\ s = g\ s'))$

lemma *eq-impD*:
  $[\![\ eq\text{-}imp\ f\ g;\ \forall\, x.\ f\ x\ s = f\ x\ s'\ ]\!] \Longrightarrow g\ s = g\ s'$
by (*simp add*: *eq-imp-def*)

lemma *eq-imp-vcg*:
  assumes *g*: *eq-imp f g*
  assumes *f*: $\forall\, x\ P.\ \{\!|P \circ (f\ x)|\!\}\ c$
  shows $\{\!|P \circ g|\!\}\ c$
apply (*rule vcg-name-pre-state*)
apply (*rename-tac s*)
apply (*rule vcg-pre*)
 apply (*rule vcg-post-imp*[*rotated*])
  apply (*rule vcg-all-lift*[**where** *'a*=*'a*])

```
  apply (rule-tac x=x and P=λfs. fs = f x s in f[rule-format])
 apply simp
 apply (frule eq-impD[where f=f, OF g])
 apply simp
apply simp
done

lemma eq-imp-vcg-LST:
  assumes g: eq-imp f g
  assumes f: ∀ x P. {|P ∘ (f x) ∘ LST|} c
  shows {|P ∘ g ∘ LST|} c
apply (rule vcg-name-pre-state)
apply (rename-tac s)
apply (rule vcg-pre)
 apply (rule vcg-post-imp[rotated])
  apply (rule vcg-all-lift[where 'a='a])
  apply (rule-tac x=x and P=λfs. fs = f x s↓ in f[rule-format])
 apply simp
 apply (frule eq-impD[where f=f, OF g])
 apply simp
apply simp
done

lemma eq-imp-fun-upd:
  assumes g: eq-imp f g
  assumes f: ∀ x. f x (s(fld := val)) = f x s
  shows g (s(fld := val)) = g s
apply (rule eq-impD[OF g])
apply (rule f)
done

lemma curry-forall-eq:
  (∀ f. P f) = (∀ f. P (case-prod f))
by (metis case-prod-curry)

lemma pres-tuple-vcg:
  (∀ P. {|P ∘ (λs. (f s, g s))|} c)
    ⟷ ((∀ P. {|P ∘ f|} c) ∧ (∀ P. {|P ∘ g|} c))
apply (simp add: curry-forall-eq o-def)
apply safe
  apply fast
 apply fast
apply (rename-tac P)
apply (rule-tac f=f and P=λfs s. P fs (g s) in vcg-lift-comp; simp)
done

lemma pres-tuple-vcg-LST:
  (∀ P. {|P ∘ (λs. (f s, g s)) ∘ LST|} c)
    ⟷ ((∀ P. {|P ∘ f ∘ LST|} c) ∧ (∀ P. {|P ∘ g ∘ LST|} c))
apply (simp add: curry-forall-eq o-def)
apply safe
  apply fast
 apply fast
apply (rename-tac P)
apply (rule-tac f=λs. f s↓ and P=λfs s. P fs (g s) for g in vcg-lift-comp; simp)
done

no-notation valid-syn (‹{|-|}/ -/ {|-|}›)
```

end

6 One locale per process

A sketch of what we're doing in *ConcurrentGC*, for quicker testing.

FIXME write some lemmas that further exercise the generated thms.

locale *P1*
begin

definition *com* :: (*unit*, *string*, *unit*, *nat*) *com* **where**
  com = ⦃''A''⦄ *WHILE* ((<) *0*) *DO* ⦃''B''⦄ ⌊λs. s − 1⌋ *OD*

intern-com *com-def*
print-theorems

locset-definition *loop* = {*B*}
print-theorems
thm *locset-cache*

definition *assertion* = *atS False loop*

end

thm *locset-cache*

locale *P2*
begin

thm *locset-cache*

definition *com* :: (*unit*, *string*, *unit*, *nat*) *com* **where**
  com = ⦃''C''⦄ *WHILE* ((<) *0*) *DO* ⦃''A''⦄ ⌊*Suc*⌋ *OD*

intern-com *com-def*
locset-definition *loop* = {*A*}
print-theorems

end

thm *locset-cache*

primrec *coms* :: *bool* ⇒ (*unit*, *string*, *unit*, *nat*) *com* **where**
  coms False = *P1.com*
| *coms True* = *P2.com*

7 Example: a one-place buffer

To demonstrate the CIMP reasoning infrastructure, we treat the trivial one-place buffer example of Lamport and Schneider (1984, §3.3). Note that the semantics for our language is different to Lamport and Schneider's, who treated a historical variant of CSP (i.e., not the one in Hoare (1985)).

We introduce some syntax for fixed-topology (static channel-based) scenarios.

abbreviation
  rcv-syn :: *'location* ⇒ *'channel* ⇒ (*'val* ⇒ *'state* ⇒ *'state*)

$\Rightarrow$ *(unit, 'location, 'channel × 'val, 'state) com* (‹{|-|}/ -▷-› [0,0,81] 81)
where
{|l|} ch▷f ≡ {|l|} Response (λq s. if fst q = ch then {(f (snd q) s, ())} else {})

abbreviation
snd-syn :: 'location ⇒ 'channel ⇒ ('state ⇒ 'val)
⇒ *(unit, 'location, 'channel × 'val, 'state) com* (‹{|-|}/ -◁-› [0,0,81] 81)
where
{|l|} ch◁f ≡ {|l|} Request (λs. (ch, f s)) (λans s. {s})

These definitions largely follow Lamport and Schneider (1984). We have three processes communicating over two channels. We enumerate program locations.

datatype *ex-chname* = ξ12 | ξ23
type-synonym *ex-val* = *nat*
type-synonym *ex-ch* = *ex-chname* × *ex-val*
datatype *ex-loc* = r12 | r23 | s23 | s12
datatype *ex-proc* = p1 | p2 | p3

type-synonym *ex-pgm* = *(unit, ex-loc, ex-ch, ex-val) com*
type-synonym *ex-pred* = *(unit, ex-loc, ex-proc, ex-ch, ex-val) state-pred*
type-synonym *ex-state* = *(unit, ex-loc, ex-proc, ex-ch, ex-val) system-state*
type-synonym *ex-sys* = *(unit, ex-loc, ex-proc, ex-ch, ex-val) system*
type-synonym *ex-history* = *(ex-ch × unit) list*

We further specialise these for our particular example.

primrec
ex-coms :: *ex-proc* ⇒ *ex-pgm*
where
ex-coms p1 = {|s12|} ξ12◁id
| *ex-coms p2* = LOOP DO {|r12|} ξ12▷(λv -. v) ;; {|s23|} ξ23◁id OD
| *ex-coms p3* = {|r23|} ξ23▷(λv -. v)

Each process starts with an arbitrary initial local state.

abbreviation *ex-init* :: (*ex-proc* ⇒ *ex-val*) ⇒ *bool* **where**
ex-init ≡ ⟨True⟩

abbreviation *sys* :: *ex-sys* **where**
sys ≡ (|PGMs = ex-coms, INIT = ex-init, FAIR = ⟨True⟩|)

The following adapts Kai Engelhardt's, from his notes titled *Proving an Asynchronous Message Passing Program Correct*, 2011. The history variable tracks the causality of the system, which I feel is missing in Lamport's treatment. We tack on Lamport's invariant so we can establish *Etern-pred*.

abbreviation
filter-on-channel :: *ex-chname* ⇒ *ex-state* ⇒ *ex-val list* (‹⌊-› [100] 101)
where
⌊ch ≡ map (snd ∘ fst) ∘ filter ((=) ch ∘ fst ∘ fst) ∘ HST

definition *IL* :: *ex-pred* **where**
IL = *pred-conjoin* [
at p1 s12 ⟶ LIST-NULL ⌊ξ12
, terminated p1 ⟶ ⌊ξ12 = (λs. [s↓ p1])
, at p2 r12 ⟶ ⌊ξ12 = ⌊ξ23
, at p2 s23 ⟶ ⌊ξ12 = ⌊ξ23 @ (λs. [s↓ p2]) ∧ (λs. s↓ p1 = s↓ p2)
, at p3 r23 ⟶ LIST-NULL ⌊ξ23
, terminated p3 ⟶ ⌊ξ23 = (λs. [s↓ p2]) ∧ (λs. s↓ p1 = s↓ p3)
]

If *p3* terminates, then it has *p1*'s value. This is stronger than Lamport and Schneider's as we don't ask that the first process has also terminated.

definition *Etern-pred* :: *ex-pred* **where**
Etern-pred = (*terminated p3* ⟶ (λ*s*. *s*↓ *p1* = *s*↓ *p3*))

Proofs from here down.

lemma *correct-system*:
assumes *IL sh*
shows *Etern-pred sh*
using *assms* **unfolding** *Etern-pred-def IL-def* **by** *simp*

lemma *IL-p1*: *ex-coms*, *p1*, *lconst* {} ⊢ ⦃*IL*⦄ ⦃*s12*⦄ *ξ12*◁(λ*s*. *s*)
apply (*rule vcg.intros*)
apply (*rename-tac p′*)
apply (*case-tac p′*; *clarsimp simp*: *IL-def atLs-def*)
done

lemma *IL-p2*: *ex-coms*, *p2*, *lconst* {*r12*} ⊢ ⦃*IL*⦄ ⦃*s23*⦄ *ξ23*◁(λ*s*. *s*)
apply (*rule vcg.intros*)
apply (*rename-tac p′*)
apply (*case-tac p′*; *clarsimp simp*: *IL-def*)
done

lemma *IL*: *sys* $\models_{pre}$ *IL*
apply (*rule VCG*)
apply (*clarsimp simp*: *IL-def atLs-def dest*!: *initial-stateD*)
apply (*rename-tac p*)
apply (*case-tac p*; *clarsimp simp*: *IL-p1 IL-p2*)
done

lemma *IL-valid*: *sys* ⊨ □⌈*IL*⌉
by (*rule valid-prerun-lift*[*OF IL*])

8 Example: an unbounded buffer

This is more literally Kai Engelhardt's example from his notes titled *Proving an Asynchronous Message Passing Program Correct*, 2011.

datatype *ex-chname* = *ξ12* | *ξ23*
type-synonym *ex-val* = *nat*
type-synonym *ex-ls* = *ex-val list*
type-synonym *ex-ch* = *ex-chname* × *ex-val*
datatype *ex-loc* = *c1* | *r12* | *r23* | *s23* | *s12*
datatype *ex-proc* = *p1* | *p2* | *p3*

type-synonym *ex-pgm* = (*unit*, *ex-loc*, *ex-ch*, *ex-ls*) *com*
type-synonym *ex-pred* = (*unit*, *ex-loc*, *ex-proc*, *ex-ch*, *ex-ls*) *state-pred*
type-synonym *ex-state* = (*unit*, *ex-loc*, *ex-proc*, *ex-ch*, *ex-ls*) *system-state*
type-synonym *ex-sys* = (*unit*, *ex-loc*, *ex-proc*, *ex-ch*, *ex-ls*) *system*
type-synonym *ex-history* = (*ex-ch* × *unit*) *list*

The local state for the producer process contains all values produced; consider that ghost state.

abbreviation (*input*) *snoc* :: ′*a* ⇒ ′*a list* ⇒ ′*a list* **where** *snoc x xs* ≡ *xs* @ [*x*]

primrec *ex-coms* :: *ex-proc* ⇒ *ex-pgm* **where**
ex-coms p1 = *LOOP DO* ⦃*c1*⦄ *LocalOp* (λ*xs*. {*snoc x xs* |*x*. *True*}) ;; ⦃*s12*⦄ *ξ12*◁(*last*, *id*) *OD*
| *ex-coms p2* = *LOOP DO* ⦃*r12*⦄ *ξ12*▷*snoc*
⊕ ⦃*c1*⦄ *IF* (λ*s*. *length s* > *0*) *THEN* ⦃*s23*⦄ *ξ12*◁(*hd*, *tl*) *FI*
OD

| *ex-coms p3* = *LOOP DO* ⦃*r23*⦄ *ξ23*▹*snoc OD*

abbreviation *ex-init* :: (*ex-proc* ⇒ *ex-ls*) ⇒ *bool* **where**
ex-init s ≡ ∀ *p*. *s p* = []

abbreviation *sys* :: *ex-sys* **where**
sys ≡ (|*PGMs* = *ex-coms*, *INIT* = *ex-init*, *FAIR* = ⟨*True*⟩|)

abbreviation
filter-on-channel :: *ex-chname* ⇒ *ex-state* ⇒ *ex-val list* (‹⌊-› [*100*] *101*)
where
⌊*ch* ≡ *map* (*snd* ∘ *fst*) ∘ *filter* ((=) *ch* ∘ *fst* ∘ *fst*) ∘ *HST*

definition *I-pred* :: *ex-pred* **where**
I-pred = *pred-conjoin* [
    at p1 c1 ⟶ ⌊*ξ12* = (λ*s*. *s*↓ *p1*)
  , *at p1 s12* ⟶ (λ*s*. *length* (*s*↓ *p1*) > *0* ∧ *butlast* (*s*↓ *p1*) = (⌊*ξ12*) *s*)
  , ⌊*ξ12* ≤ (λ*s*. *s*↓ *p1*)
  , ⌊*ξ12* = ⌊*ξ23* @ (λ*s*. *s*↓ *p2*)
  , *at p2 s23* ⟶ (λ*s*. *length* (*s*↓ *p2*) > *0*)
  , (λ*s*. *s*↓ *p3*) = ⌊*ξ23*
  ]

The local state of *p3* is some prefix of the local state of *p1*.

definition *Etern-pred* :: *ex-pred* **where**
Etern-pred ≡ λ*s*. *s*↓ *p3* ≤ *s*↓ *p1*

lemma *correct-system*:
assumes *I-pred s*
shows *Etern-pred s*
using *assms* **unfolding** *Etern-pred-def I-pred-def less-eq-list-def prefix-def* **by** *clarsimp*

lemma *p1-c1*[*simplified*, *intro*]:
ex-coms, *p1*, *lconst* {*s12*} ⊢ ⦃*I-pred*⦄ ⦃*c1*⦄ *LocalOp* (λ*xs*. { *snoc x xs* |*x*. *True* })
apply (*rule vcg.intros*)
apply (*clarsimp simp*: *I-pred-def atS-def*)
done

lemma *p1-s12*[*simplified*, *intro*]:
ex-coms, *p1*, *lconst* {*c1*} ⊢ ⦃*I-pred*⦄ ⦃*s12*⦄ *ξ12*◃(*last*, *id*)
apply (*rule vcg.intros*)
apply (*rename-tac p'*)
apply (*case-tac p'*; *clarsimp*)
apply (*clarsimp simp*: *I-pred-def atS-def*)
apply (*metis Prefix-Order.prefix-snoc append.assoc append-butlast-last-id*)
done

lemma *p2-s23*[*simplified*, *intro*]:
ex-coms, *p2*, *lconst* {*c1*, *r12*} ⊢ ⦃*I-pred*⦄ ⦃*s23*⦄ *ξ12*◃(*hd*, *tl*)
apply (*rule vcg.intros*)
apply (*rename-tac p'*)
apply (*case-tac p'*; *clarsimp*)
done

lemma *p2-pi4*[*intro*]:
ex-coms, *p2*, *lcond* {*s23*} {*c1*, *r12*} (λ*s*. *s* ≠ []) ⊢ ⦃*I-pred*⦄ ⦃*c1*⦄ *IF* (λ*s*. *s* ≠ []) *THEN c' FI*
apply (*rule vcg.intros*)
apply (*clarsimp simp*: *I-pred-def atS-def split*: *lcond-splits*)

done

lemma *I*: *sys* $\models_{pre}$ *I-pred*
apply (*rule VCG*)
apply (*clarsimp dest*!: *initial-stateD simp*: *I-pred-def atS-def*)
apply (*rename-tac p*)
apply (*case-tac p*; *auto*)
done

lemma *I-valid*: *sys* $\models \Box\lceil$*I-pred*$\rceil$
by (*rule valid-prerun-lift*[*OF I*])

9 Concluding remarks

Previously Nipkow and Prensa Nieto (1999); Prensa Nieto (2002, 2003)[3] have developed the classical Owicki/Gries and Rely-Guarantee paradigms for the verification of shared-variable concurrent programs in Isabelle/HOL. These have been used to show the correctness of a garbage collector (Prensa Nieto and Esparza 2000).

We instead use synchronous message passing, which is significantly less explored. de Boer, de Roever, and Hannemann (1999); **?** provide compositional systems for *terminating* systems. We have instead adopted Lamport's paradigm of a single global invariant and local proof obligations as the systems we have in mind are tightly coupled and it is not obvious that the proofs would be easier on a decomposed system; see **?**, §1.6.6 for a concurring opinion.

Unlike the generic sequential program verification framework Simpl (Schirmer 2004), we do not support function calls, or a sophisticated account of state spaces. Moreover we do no meta-theory beyond showing the simple VCG is sound (§5.1).

References

B. Alpern and F. B. Schneider. Defining liveness. *Information Processing Letters*, 21(4):181–185, 1985. doi: 10.1016/0020-0190(85)90056-0.

K. R. Apt, N. Francez, and W. P. de Roever. A proof system for communicating sequential processes. *ACM Transactions on Programming Languages and Systems*, 2(3):359–385, 1980. doi: 10.1145/357103.357110.

S. Berghofer. A solution to the PoplMark challenge using de Bruijn indices in Isabelle/HOL. *J. Autom. Reasoning*, 49(3):303–326, 2012.

K. M. Chandy and J. Misra. *Parallel program design - a foundation.* Addison-Wesley, 1989. ISBN 978-0-201-05866-6.

E. Y. Chang, Z. Manna, and A. Pnueli. Characterization of temporal property classes. In *ICALP'1992*, volume 623 of *LNCS*, pages 474–486. Springer, 1992. doi: 10.1007/3-540-55719-9_97.

P. Cousot and R. Cousot. Semantic analysis of Communicating Sequential Processes (shortened version). In *ICALP*, volume 85 of *LNCS*, pages 119–133. Springer, 1980.

P. Cousot and R. Cousot. A language independent proof of the soundness and completeness of generalized Hoare logic. *Information and Computation*, 80(2):165–191, February 1989.

F. S. de Boer, W. P. de Roever, and U. Hannemann. The semantic foundations of a compositional proof method for synchronously communicating processes. In M. Kutylowski, L. Pacholski, and T. Wierzbicki, editors, *MFCS*, volume 1672 of *LNCS*, pages 343–353. Springer, 1999.

W. P. de Roever, F. S. de Boer, U. Hannemann, J. Hooman, Y. Lakhnech, M. Poel, and J. Zwiers. *Concurrency Verification: Introduction to Compositional and Noncompositional Methods*, volume 54 of *Cambridge Tracts in Theoretical Computer Science*. Cambridge University Press, 2001.

[3]The theories are in `$ISABELLE/src/HOL/Hoare_Parallel`.

M. Felleisen and R. Hieb. The revised report on the syntactic theories of sequential control and state. *Theoretical Computer Science*, 103(2):235–271, 1992. doi: 10.1016/0304-3975(92)90014-7.

G. Grov and S. Merz. A definitional encoding of tla* in isabelle/hol. *Archive of Formal Proofs*, November 2011. ISSN 2150-914x. http://isa-afp.org/entries/TLA, Formal proof development.

C.A.R. Hoare. *Communicating Sequential Processes.* International Series In Computer Science. Prentice-Hall, 1985. URL http://www.usingcsp.com/.

P. B. Jackson. Verifying a garbage collection algorithm. In *TPHOLs*, volume 1479 of *LNCS*, pages 225–244. Springer, 1998. doi: 10.1007/BFb0055139.

E. Kindler. Safety and liveness properties: A survey. *Bulletin of the European Association for Theoretical Computer Science*, 53(30):268–272, 6 1994. URL http://citeseerx.ist.psu.edu/viewdoc/download?doi=10.1.1.43.8206&rep=rep1&type=pdf.

L. Lamport. The “Hoare Logic” of concurrent programs. *Acta Informatica*, 14:21–37, 1980.

L. Lamport. *Specifying Systems, The TLA+ Language and Tools for Hardware and Software Engineers.* Addison-Wesley, 2002.

L. Lamport and F. B. Schneider. The “Hoare Logic” of CSP, and all that. *ACM Transactions on Programming Languages and Systems*, 6(2):281–296, 1984.

G. Levin and D. Gries. A proof technique for communicating sequential processes. *Acta Inf.*, 15:281–302, 1981.

Z. Manna and A. Pnueli. The anchored version of the temporal framework. In J. W. de Bakker, W. P. de Roever, and G. Rozenberg, editors, *Linear Time, Branching Time and Partial Order in Logics and Models for Concurrency, School/Workshop, Noordwijkerhout, The Netherlands, May 30 - June 3, 1988, Proceedings*, volume 354 of *LNCS*, pages 201–284. Springer, 1988. doi: 10.1007/BFb0013024.

Z. Manna and A. Pnueli. Tools and rules for the practicing verifier. In R. F. Rashid, editor, *CMU Computer Science: A 25th Anniversary Commemorative*, pages 121–156. ACM Press and Addison-Wesley, 1991. Also Technical Report STAN-CS-90-1321.

Z. Manna and A. Pnueli. *Temporal verification of reactive systems - Safety.* Springer, 1995.

Z. Manna and A. Pnueli. Temporal verification of reactive systems: Response. In *Time for Verification, Essays in Memory of Amir Pnueli*, volume 6200 of *LNCS*, pages 279–361. Springer, 2010. doi: 10.1007/978-3-642-13754-9_13.

R. Milner. *A Calculus of Communicating Systems.* Springer, 1980.

R. Milner. *Communication and Concurrency.* Prentice-Hall, Englewood Cliffs, NJ, 1989.

T. Nipkow and G. Klein. *Concrete Semantics: A Proof Assistant Approach.* Springer, 2014. URL http://www.in.tum.de/~nipkow/Concrete-Semantics/.

T. Nipkow and L. Prensa Nieto. Owicki/Gries in Isabelle/HOL. In J.-P. Finance, editor, *FASE*, volume 1577 of *LNCS*, pages 188–203. Springer, 1999.

S. S. Owicki and L. Lamport. Proving liveness properties of concurrent programs. *ACM Transactions on Programming Languages and Systems*, 4(3):455–495, 1982. doi: 10.1145/357172.357178.

A. M. Pitts. Operational semantics and program equivalence. In G. Barthe, P. Dybjer, and J. Saraiva, editors, *Applied Semantics, Advanced Lectures*, volume 2395 of *LNCS*, pages 378–412. Springer, 2002. International Summer School, APPSEM 2000, Caminha, Portugal, September 9–15, 2000.

G. D. Plotkin. The origins of structural operational semantics. *Journal of Logic and Algebraic Programming*, 60-61:3–15, 2004.

L. Prensa Nieto. *Verification of Parallel Programs with the Owicki-Gries and Rely-Guarantee Methods in Isabelle/HOL.* PhD thesis, Technische Universität München, 2002.

L. Prensa Nieto. The Rely-Guarantee method in Isabelle/HOL. In P. Degano, editor, *ESOP'2003*, volume 2618 of *LNCS*, pages 348–362. Springer, 2003.

L. Prensa Nieto and J. Esparza. Verifying single and multi-mutator garbage collectors with Owicki/Gries in Isabelle/HOL. In M. Nielsen and B. Rovan, editors, *Mathematical Foundations of Computer Science (MFCS 2000)*, volume 1893 of *LNCS*, pages 619–628. Springer, 2000.

T. Ridge. Verifying distributed systems: the operational approach. In *POPL'2009*, pages 429–440. ACM, 2009. doi: 10.1145/1480881.1480934.

N. Schirmer. A verification environment for sequential imperative programs in isabelle/hol. In F. Baader and A. Voronkov, editors, *LPAR*, volume 3452 of *LNCS*, pages 398–414. Springer, 2004.

N. Schirmer and M. Wenzel. State spaces - the locale way. *Electr. Notes Theor. Comput. Sci.*, 254:161–179, 2009.

F. B. Schneider. Decomposing properties into safety and liveness using predicate logic. Technical Report 87-874, Department of Computer Science, Cornell University, October 1987.

A. P. Sistla. Safety, liveness and fairness in temporal logic. *Formal Aspects of Computing*, 6(5):495–512, 1994. doi: 10.1007/BF01211865.

R. J. van Glabbeek and P. Höfner. Progress, justness and fairness. *ACM Computing Surveys*, 2019. URL http://arxiv.org/abs/1810.07414v1.

G. Winskel. *The Formal Semantics of Programming Languages.* MIT Press, Cambridge, MA, 1993.