CIMP

Peter Gammie

October 13, 2025

Abstract

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

Contents

$\langle proof \rangle \langle proof \rangle$

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

abbreviation (*input*)

$pred\text{-}not :: ('a \Rightarrow bool) \Rightarrow 'a \Rightarrow bool$ (‹¬ -› [40] 40) **where**
$\neg a \equiv \lambda s.\ \neg a\ s$

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

More application specific.

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

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

abbreviation (*input*)
  pred-is-none :: $('a \Rightarrow 'b\ option) \Rightarrow 'a \Rightarrow bool$ (‹*NULL* -› [*40*] *40*) **where**
  $NULL\ a \equiv \lambda s.\ a\ s = None$

abbreviation (*input*)
  pred-empty :: $('a \Rightarrow 'b\ set) \Rightarrow 'a \Rightarrow bool$ (‹*EMPTY* -› [*40*] *40*) **where**
  $EMPTY\ a \equiv \lambda s.\ a\ s = \{\}$

abbreviation (*input*)
  pred-list-null :: $('a \Rightarrow 'b\ list) \Rightarrow 'a \Rightarrow bool$ (‹*LIST'-NULL* -› [*40*] *40*) **where**
  $LIST\text{-}NULL\ a \equiv \lambda s.\ a\ s = []$

abbreviation (*input*)
  pred-list-append :: $('a \Rightarrow 'b\ list) \Rightarrow ('a \Rightarrow 'b\ list) \Rightarrow 'a \Rightarrow 'b\ list$ (**infixr** ‹**@**› *65*) **where**
  $xs\ \mathbf{@}\ ys \equiv \lambda s.\ xs\ s\ @\ ys\ s$

abbreviation (*input*)
  pred-pair :: $('a \Rightarrow 'b) \Rightarrow ('a \Rightarrow 'c) \Rightarrow 'a \Rightarrow 'b \times 'c$ (**infixr** ‹⊗› *60*) **where**
  $a \otimes b \equiv \lambda s.\ (a\ s,\ b\ s)$

abbreviation (*input*)
  pred-singleton :: $('a \Rightarrow 'b) \Rightarrow 'a \Rightarrow 'b\ set$ **where**
  pred-singleton x $\equiv \lambda s.\ \{x\ s\}$

2 Infinite Sequences

Infinite sequences and some operations on them.

We use the customary function-based representation.

type-synonym $'a\ seq = nat \Rightarrow 'a$
type-synonym $'a\ seq\text{-}pred = 'a\ seq \Rightarrow bool$

definition $suffix :: 'a\ seq \Rightarrow nat \Rightarrow 'a\ seq$ (**infixl** ‹$|_s$› *60*) **where**
$\sigma\ |_s\ i \equiv \lambda j.\ \sigma\ (j + i)$

primrec $stake :: nat \Rightarrow 'a\ seq \Rightarrow 'a\ list$ **where**
$stake\ 0\ \sigma = []$
$|\ stake\ (Suc\ n)\ \sigma = \sigma\ 0\ \#\ stake\ n\ (\sigma\ |_s\ 1)$

primrec $shift :: 'a\ list \Rightarrow 'a\ seq \Rightarrow 'a\ seq$ (**infixr** ‹@−› *65*) **where**
$shift\ []\ \sigma = \sigma$
$|\ shift\ (x\ \#\ xs)\ \sigma = (\lambda i.\ case\ i\ of\ 0 \Rightarrow x\ |\ Suc\ i \Rightarrow shift\ xs\ \sigma\ i)$

abbreviation *interval-syn* (‹-'(- → -')› [*69*, *0*, *0*] *70*) **where**
$\sigma(i \rightarrow j) \equiv stake\ j\ (\sigma\ |_s\ i)$

lemma *suffix-eval*: $(\sigma\ |_s\ i)\ j = \sigma\ (j + i)$
⟨*proof*⟩

lemma *suffix-plus*: $\sigma\ |_s\ n\ |_s\ m = \sigma\ |_s\ (m + n)$
⟨*proof*⟩

lemma *suffix-commute*: $((\sigma\ |_s\ n)\ |_s\ m) = ((\sigma\ |_s\ m)\ |_s\ n)$
⟨*proof*⟩

lemma *suffix-plus-com*: $\sigma\ |_s\ m\ |_s\ n = \sigma\ |_s\ (m + n)$
⟨*proof*⟩

lemma *suffix-zero*: $\sigma\ |_s\ 0 = \sigma$
⟨*proof*⟩

lemma *comp-suffix*: $f \circ \sigma\ |_s\ i = (f \circ \sigma)\ |_s\ i$
⟨*proof*⟩

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

lemma *length-stake*[*simp*]: $length\ (stake\ n\ s) = n$
⟨*proof*⟩

lemma *shift-simps*[*simp*]:
$(xs\ @{-}\ \sigma)\ 0 = (if\ xs = []\ then\ \sigma\ 0\ else\ hd\ xs)$
$(xs\ @{-}\ \sigma)\ |_s\ Suc\ 0 = (if\ xs = []\ then\ \sigma\ |_s\ Suc\ 0\ else\ tl\ xs\ @{-}\ \sigma)$
⟨*proof*⟩

lemma *stake-nil*[*simp*]:
$stake\ i\ \sigma = [] \longleftrightarrow i = 0$
⟨*proof*⟩

lemma *stake-shift*:
$stake\ i\ (w\ @{-}\ \sigma) = take\ i\ w\ @\ stake\ (i - length\ w)\ \sigma$

⟨*proof*⟩

lemma *shift-snth-less*[*simp*]:
  assumes $i < length\ xs$
  shows $(xs\ @-\ \sigma)\ i = xs\ !\ i$
⟨*proof*⟩

lemma *shift-snth-ge*[*simp*]:
  assumes $i \geq length\ xs$
  shows $(xs\ @-\ \sigma)\ i = \sigma\ (i - length\ xs)$
⟨*proof*⟩

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

lemma *suffix-shift*:
  $(xs\ @-\ \sigma)\ |_s\ i = drop\ i\ xs\ @-\ (\sigma\ |_s\ i - length\ xs)$
⟨*proof*⟩

lemma *stake-nth*[*simp*]:
  assumes $i < j$
  shows $stake\ j\ s\ !\ i = s\ i$
⟨*proof*⟩

lemma *stake-suffix-id*:
  $stake\ i\ \sigma\ @-\ (\sigma\ |_s\ i) = \sigma$
⟨*proof*⟩

lemma *id-stake-snth-suffix*:
  $\sigma = (stake\ i\ \sigma\ @\ [\sigma\ i])\ @-\ (\sigma\ |_s\ Suc\ i)$
⟨*proof*⟩

lemma *stake-add*[*simp*]:
  $stake\ i\ \sigma\ @\ stake\ j\ (\sigma\ |_s\ i) = stake\ (i + j)\ \sigma$
⟨*proof*⟩

lemma *stake-append*: $stake\ n\ (u\ @-\ s) = take\ (min\ (length\ u)\ n)\ u\ @\ stake\ (n - length\ u)\ s$
⟨*proof*⟩

lemma *stake-shift-stake-shift*:
  $stake\ i\ \sigma\ @-\ stake\ j\ (\sigma\ |_s\ i)\ @-\ \beta = stake\ (i + j)\ \sigma\ @-\ \beta$
⟨*proof*⟩

lemma *stake-suffix-drop*:
  $stake\ i\ (\sigma\ |_s\ j) = drop\ j\ (stake\ (i + j)\ \sigma)$
⟨*proof*⟩

lemma *stake-suffix*:
  assumes $i \leq j$
  shows $stake\ j\ \sigma\ @-\ u\ |_s\ i = \sigma(i \rightarrow j - i)\ @-\ u$
⟨*proof*⟩

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
$safety\ P \longleftrightarrow (\forall \sigma.\ (\forall i.\ \exists \beta.\ P\ (stake\ i\ \sigma\ @\!-\ \beta)) \longrightarrow P\ \sigma)$
⟨*proof*⟩

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 $safety\ (\lambda\sigma.\ False)$
⟨*proof*⟩

lemma *safety-True*:
shows $safety\ (\lambda\sigma.\ True)$
⟨*proof*⟩

lemma *safety-state-prop*:
shows $safety\ (\lambda\sigma.\ P\ (\sigma\ 0))$
⟨*proof*⟩

lemma *safety-invariant*:
shows $safety\ (\lambda\sigma.\ \forall i.\ P\ (\sigma\ i))$
⟨*proof*⟩

lemma *safety-transition-relation*:
shows $safety\ (\lambda\sigma.\ \forall i.\ (\sigma\ i,\ \sigma\ (i + 1)) \in R)$
⟨*proof*⟩

lemma *safety-conj*:
assumes $safety\ P$
assumes $safety\ Q$
shows $safety\ (P \wedge Q)$
⟨*proof*⟩

lemma *safety-always-eventually*[*simplified*]:
assumes $safety\ P$
assumes $\forall i.\ \exists j \geq i.\ \exists \beta.\ P\ (\sigma(0 \to j)\ @\!-\ \beta)$
shows $P\ \sigma$
⟨*proof*⟩

lemma *safety-disj*:
assumes $safety\ P$
assumes $safety\ Q$
shows $safety\ (P \vee Q)$
⟨*proof*⟩

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* $\Rightarrow$ $'a$ *seq-pred* **where**
  $\mathit{Live}\ P = (P \vee \neg M_p\ P)$

lemma *decomp*:
  $P = (\mathit{Safe}\ P \wedge \mathit{Live}\ P)$
⟨*proof*⟩

lemma *safe*:
  $\mathit{safety}\ (\mathit{Safe}\ P)$
⟨*proof*⟩

lemma *live*:
  $\mathit{liveness}\ (\mathit{Live}\ P)$
⟨*proof*⟩

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
  $\mathit{absolute\text{-}liveness}\ P \longleftrightarrow (\exists\sigma.\ P\ \sigma) \wedge (\forall\sigma\ \alpha.\ P\ \sigma \longrightarrow P\ (\alpha\ @\!-\ \sigma))$

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

lemma *absolute-liveness-liveness*:
  assumes *absolute-liveness* P
  shows *liveness* P
⟨*proof*⟩

lemma *stable-absolute-liveness*:
  assumes $P\ \sigma$
  assumes $\neg P\ \sigma'$ — extra hypothesis
  shows $\mathit{stable}\ P \longleftrightarrow \mathit{absolute\text{-}liveness}\ (\neg\ P)$
⟨*proof*⟩

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

lemma *fairness-safety*:
  assumes *safety* P
  assumes *fairness* F
  shows $(\forall\sigma.\ F\ \sigma \longrightarrow P\ \sigma) \longleftrightarrow (\forall\sigma.\ P\ \sigma)$
⟨*proof*⟩

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

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

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

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

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

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

abbreviation (*input*)
pred-always-imp-syn :: $'a$ *seq-pred* $\Rightarrow$ $'a$ *seq-pred* $\Rightarrow$ $'a$ *seq-pred* (**infixr** ‹$\hookrightarrow$› *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 $\lceil P\rceil\ (\sigma\ |_s\ i) = P\ (\sigma\ i)$
⟨*proof*⟩

lemma *alwaysI*[*intro*]:
assumes $\bigwedge i.\ P\ (\sigma\ |_s\ i)$
shows $(\Box P)\ \sigma$
⟨*proof*⟩

lemma *alwaysD*:
assumes $(\Box P)\ \sigma$
shows $P\ (\sigma\ |_s\ i)$
⟨*proof*⟩

lemma *alwaysE*: $[\![(\Box P)\ \sigma;\ P\ (\sigma\ |_s\ i) \Longrightarrow Q]\!] \Longrightarrow Q$
⟨*proof*⟩

lemma *always-induct*:
assumes $P\ \sigma$
assumes $(\Box(P \longrightarrow \bigcirc P))\ \sigma$
shows $(\Box P)\ \sigma$
⟨*proof*⟩

lemma *seq-comp*:
fixes σ :: $'a$ *seq*
fixes P :: $'b$ *seq-pred*

fixes f :: $'a \Rightarrow 'b$
shows
$(\Box P)\ (f \circ \sigma) \longleftrightarrow (\Box(\lambda\sigma.\ P\ (f \circ \sigma)))\ \sigma$
$(\Diamond P)\ (f \circ \sigma) \longleftrightarrow (\Diamond(\lambda\sigma.\ P\ (f \circ \sigma)))\ \sigma$
$(P\ \mathcal{U}\ Q)\ (f \circ \sigma) \longleftrightarrow ((\lambda\sigma.\ P\ (f \circ \sigma))\ \mathcal{U}\ (\lambda\sigma.\ Q\ (f \circ \sigma)))\ \sigma$
$(P\ \mathcal{W}\ Q)\ (f \circ \sigma) \longleftrightarrow ((\lambda\sigma.\ P\ (f \circ \sigma))\ \mathcal{W}\ (\lambda\sigma.\ Q\ (f \circ \sigma)))\ \sigma$
$\langle proof \rangle$

lemma *nextI*[*intro*]:
assumes $P\ (\sigma\ |_s\ Suc\ 0)$
shows $(\bigcirc P)\ \sigma$
$\langle proof \rangle$

lemma *untilI*[*intro*]:
assumes $Q\ (\sigma\ |_s\ i)$
assumes $\forall\, k<i.\ P\ (\sigma\ |_s\ k)$
shows $(P\ \mathcal{U}\ Q)\ \sigma$
$\langle proof \rangle$

lemma *untilE*:
assumes $(P\ \mathcal{U}\ Q)\ \sigma$
obtains i **where** $Q\ (\sigma\ |_s\ i)$ **and** $\forall\, k<i.\ P\ (\sigma\ |_s\ k)$
$\langle proof \rangle$

lemma *eventuallyI*[*intro*]:
assumes $P\ (\sigma\ |_s\ i)$
shows $(\Diamond P)\ \sigma$
$\langle proof \rangle$

lemma *eventuallyE*[*elim*]:
assumes $(\Diamond P)\ \sigma$
obtains i **where** $P\ (\sigma\ |_s\ i)$
$\langle proof \rangle$

lemma *unless-alwaysI*:
assumes $(\Box\ P)\ \sigma$
shows $(P\ \mathcal{W}\ Q)\ \sigma$
$\langle proof \rangle$

lemma *unless-untilI*:
assumes $Q\ (\sigma\ |_s\ j)$
assumes $\bigwedge i.\ i < j \Longrightarrow P\ (\sigma\ |_s\ i)$
shows $(P\ \mathcal{W}\ Q)\ \sigma$
$\langle proof \rangle$

lemma *always-imp-refl*[*iff*]:
shows $(P \hookrightarrow P)\ \sigma$
$\langle proof \rangle$

lemma *always-imp-trans*:
assumes $(P \hookrightarrow Q)\ \sigma$
assumes $(Q \hookrightarrow R)\ \sigma$
shows $(P \hookrightarrow R)\ \sigma$
$\langle proof \rangle$

lemma *always-imp-mp*:
assumes $(P \hookrightarrow Q)\ \sigma$
assumes $P\ \sigma$

shows $Q\ \sigma$
⟨*proof*⟩

lemma *always-imp-mp-suffix*:
assumes $(P \hookrightarrow Q)\ \sigma$
assumes $P\ (\sigma \mid_s i)$
shows $Q\ (\sigma \mid_s i)$
⟨*proof*⟩

Some basic facts and equivalences, mostly sanity.

lemma *necessitation*:
$(\bigwedge s.\ P\ s) \Longrightarrow (\Box P)\ \sigma$
$(\bigwedge s.\ P\ s) \Longrightarrow (\Diamond P)\ \sigma$
$(\bigwedge s.\ P\ s) \Longrightarrow (P\ \mathcal{W}\ Q)\ \sigma$
$(\bigwedge s.\ Q\ s) \Longrightarrow (P\ \mathcal{U}\ Q)\ \sigma$
⟨*proof*⟩

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

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

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

$(\Box \langle \mathit{False} \rangle) = \langle \mathit{False} \rangle$
$(\Box \langle \mathit{True} \rangle) = \langle \mathit{True} \rangle$
$(\neg \Box\ P)\ \sigma = (\Diamond\ (\neg\ P))\ \sigma$
$(\Box \Box\ P) = (\Box\ P)$

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

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

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

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

$\langle \mathit{proof}\rangle$

lemma *always-conj-distrib*: $(\Box(P \wedge Q)) = (\Box P \wedge \Box Q)$
$\langle \mathit{proof}\rangle$

lemma *eventually-disj-distrib*: $(\Diamond(P \vee Q)) = (\Diamond P \vee \Diamond Q)$
$\langle \mathit{proof}\rangle$

lemma *always-eventually*[*elim*!]:
assumes $(\Box P)\ \sigma$
shows $(\Diamond P)\ \sigma$
$\langle \mathit{proof}\rangle$

lemma *eventually-imp-conv-disj*: $(\Diamond(P \longrightarrow Q)) = (\Diamond(\neg P) \vee \Diamond Q)$
$\langle \mathit{proof}\rangle$

lemma *eventually-imp-distrib*:
$(\Diamond(P \longrightarrow Q)) = (\Box P \longrightarrow \Diamond Q)$
$\langle \mathit{proof}\rangle$

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

lemma *mono*:
$[\![(\Box P)\ \sigma; \bigwedge\sigma.\ P\ \sigma \Longrightarrow\ P'\ \sigma]\!] \Longrightarrow (\Box P')\ \sigma$
$[\![(\Diamond P)\ \sigma; \bigwedge\sigma.\ P\ \sigma \Longrightarrow\ P'\ \sigma]\!] \Longrightarrow (\Diamond P')\ \sigma$
$[\![(P\ \mathcal{U}\ Q)\ \sigma; \bigwedge\sigma.\ P\ \sigma \Longrightarrow P'\ \sigma; \bigwedge\sigma.\ Q\ \sigma \Longrightarrow Q'\ \sigma]\!] \Longrightarrow (P'\ \mathcal{U}\ Q')\ \sigma$
$[\![(P\ \mathcal{W}\ Q)\ \sigma; \bigwedge\sigma.\ P\ \sigma \Longrightarrow P'\ \sigma; \bigwedge\sigma.\ Q\ \sigma \Longrightarrow Q'\ \sigma]\!] \Longrightarrow (P'\ \mathcal{W}\ Q')\ \sigma$
$\langle \mathit{proof}\rangle$

lemma *always-imp-mono*:
$[\![(\Box P)\ \sigma; (P \hookrightarrow P')\ \sigma]\!] \Longrightarrow (\Box P')\ \sigma$
$[\![(\Diamond P)\ \sigma; (P \hookrightarrow P')\ \sigma]\!] \Longrightarrow (\Diamond P')\ \sigma$
$[\![(P\ \mathcal{U}\ Q)\ \sigma; (P \hookrightarrow P')\ \sigma; (Q \hookrightarrow Q')\ \sigma]\!] \Longrightarrow (P'\ \mathcal{U}\ Q')\ \sigma$
$[\![(P\ \mathcal{W}\ Q)\ \sigma; (P \hookrightarrow P')\ \sigma; (Q \hookrightarrow Q')\ \sigma]\!] \Longrightarrow (P'\ \mathcal{W}\ Q')\ \sigma$
$\langle \mathit{proof}\rangle$

lemma *next-conj-distrib*:
$(\bigcirc(P \wedge Q)) = (\bigcirc P \wedge \bigcirc Q)$
$\langle \mathit{proof}\rangle$

lemma *next-disj-distrib*:
$(\bigcirc(P \vee Q)) = (\bigcirc P \vee \bigcirc Q)$
$\langle \mathit{proof}\rangle$

lemma *until-next-distrib*:

$(\bigcirc(P\ \mathcal{U}\ Q)) = (\bigcirc P\ \mathcal{U}\ \bigcirc Q)$
⟨*proof*⟩

lemma *until-imp-eventually*:
$((P\ \mathcal{U}\ Q) \longrightarrow \Diamond Q)\ \sigma$
⟨*proof*⟩

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

lemma *unless-unless-disj*:
assumes $(P\ \mathcal{W}\ Q\ \mathcal{W}\ R)\ \sigma$
shows $((P \vee Q)\ \mathcal{W}\ R)\ \sigma$
⟨*proof*⟩

lemma *until-conj-distrib*:
$((P \wedge Q)\ \mathcal{U}\ R) = ((P\ \mathcal{U}\ R) \wedge (Q\ \mathcal{U}\ R))$
⟨*proof*⟩

lemma *until-disj-distrib*:
$(P\ \mathcal{U}\ (Q \vee R)) = ((P\ \mathcal{U}\ Q) \vee (P\ \mathcal{U}\ R))$
⟨*proof*⟩

lemma *eventually-until*:
$(\Diamond P) = (\neg P\ \mathcal{U}\ P)$
⟨*proof*⟩

lemma *eventually-until-eventually*:
$(\Diamond(P\ \mathcal{U}\ Q)) = (\Diamond Q)$
⟨*proof*⟩

lemma *eventually-unless-until*:
$((P\ \mathcal{W}\ Q) \wedge \Diamond Q) = (P\ \mathcal{U}\ Q)$
⟨*proof*⟩

lemma *eventually-always-imp-always-eventually*:
assumes $(\Diamond\Box P)\ \sigma$
shows $(\Box\Diamond P)\ \sigma$
⟨*proof*⟩

lemma *eventually-always-next-stable*:
assumes $(\Diamond P)\ \sigma$
assumes $(P \hookrightarrow \bigcirc P)\ \sigma$
shows $(\Diamond\Box P)\ \sigma$
⟨*proof*⟩

lemma *next-stable-imp-eventually-always*:
assumes $(P \hookrightarrow \bigcirc P)\ \sigma$
shows $(\Diamond P \longrightarrow \Diamond\Box P)\ \sigma$
⟨*proof*⟩

lemma *always-eventually-always*:
$\Diamond\Box\Diamond P = \Box\Diamond P$
⟨*proof*⟩

lemma *stable-unless*:
  assumes $(P \hookrightarrow \bigcirc(P \vee Q))\ \sigma$
  shows $(P \hookrightarrow (P\ \mathcal{W}\ Q))\ \sigma$
⟨*proof*⟩

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 $(P \hookrightarrow Q\ \mathcal{W}\ R)\ \sigma$
⟨*proof*⟩

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** ‹$\leadsto$› *25*) **where**
  $P \leadsto Q \equiv P \hookrightarrow \Diamond Q$

lemma *leads-to-refl*:
  shows $(P \leadsto P)\ \sigma$
⟨*proof*⟩

lemma *leads-to-trans*:
  assumes $(P \leadsto Q)\ \sigma$
  assumes $(Q \leadsto R)\ \sigma$
  shows $(P \leadsto R)\ \sigma$
⟨*proof*⟩

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

lemma *leads-to-mono*:
  assumes $(P' \hookrightarrow P)\ \sigma$
  assumes $(Q \hookrightarrow Q')\ \sigma$
  assumes $(P \leadsto Q)\ \sigma$
  shows $(P' \leadsto Q')\ \sigma$
⟨*proof*⟩

lemma *leads-to-eventually*:
  shows $(P \leadsto Q \longrightarrow \Diamond P \longrightarrow \Diamond Q)\ \sigma$
⟨*proof*⟩

lemma *leads-to-disj*:
  assumes $(P \leadsto R)\ \sigma$
  assumes $(Q \leadsto R)\ \sigma$
  shows $((P \vee Q) \leadsto R)\ \sigma$
⟨*proof*⟩

lemma *leads-to-leads-to-viaE*:

shows $((P \hookrightarrow P\ \mathcal{U}\ Q) \longrightarrow P \rightsquigarrow Q)\ \sigma$
$\langle proof \rangle$

lemma *leads-to-via-concl-weaken*:
assumes $(R \hookrightarrow R')\ \sigma$
assumes $(P \hookrightarrow Q\ \mathcal{U}\ R)\ \sigma$
shows $(P \hookrightarrow Q\ \mathcal{U}\ R')\ \sigma$
$\langle proof \rangle$

lemma *leads-to-via-trans*:
assumes $(A \hookrightarrow B\ \mathcal{U}\ C)\ \sigma$
assumes $(C \hookrightarrow D\ \mathcal{U}\ E)\ \sigma$
shows $(A \hookrightarrow (B \vee D)\ \mathcal{U}\ E)\ \sigma$
$\langle proof \rangle$

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 $(P \vee P' \hookrightarrow (Q \vee Q')\ \mathcal{U}\ R)\ \sigma$
$\langle proof \rangle$

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 $(A \vee C \hookrightarrow (B \vee D)\ \mathcal{U}\ E)\ \sigma$
$\langle proof \rangle$

lemma *leads-to-via-stable-augmentation*:
assumes *stable*: $(P \wedge Q \hookrightarrow \bigcirc Q)\ \sigma$
assumes U: $(A \hookrightarrow P\ \mathcal{U}\ C)\ \sigma$
shows $((A \wedge Q) \hookrightarrow P\ \mathcal{U}\ (C \wedge Q))\ \sigma$
$\langle proof \rangle$

lemma *leads-to-via-wf*:
assumes *wf R*
assumes *indhyp*: $\bigwedge t.\ (A \wedge \lceil \delta = \langle t \rangle \rceil \hookrightarrow B\ \mathcal{U}\ (A \wedge \lceil \delta \otimes \langle t \rangle \in \langle R \rangle \rceil \vee C))\ \sigma$
shows $(A \hookrightarrow B\ \mathcal{U}\ C)\ \sigma$
$\langle proof \rangle$

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 $Is :: ('a\ seq\text{-}pred \times ('a \Rightarrow 'b))\ set$
assumes $wf\ (R :: 'b\ rel)$
assumes $W1$: $(\Box(\exists \varphi.\ \lceil \langle \varphi \in fst\ `\ Is \rangle \rceil \wedge (P \longrightarrow \varphi)))\ \sigma$
assumes $W2$: $\forall (\varphi, \delta) \in Is.\ \exists (\varphi', \delta') \in insert\ (Q, \delta 0)\ Is.\ \forall t.\ (\varphi \wedge \lceil \delta = \langle t \rangle \rceil \rightsquigarrow \varphi' \wedge \lceil \delta' \otimes \langle t \rangle \in \langle R \rangle \rceil)\ \sigma$
shows $(P \rightsquigarrow Q)\ \sigma$
$\langle proof \rangle$

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 $\mathit{weakly\text{-}fair} :: {'a}\ \mathit{seq\text{-}pred} \Rightarrow {'a}\ \mathit{seq\text{-}pred} \Rightarrow {'a}\ \mathit{seq\text{-}pred}$ **where**
$\mathit{weakly\text{-}fair}\ \mathit{enabled}\ \mathit{taken} = (\Box \mathit{enabled} \hookrightarrow \Diamond \mathit{taken})$

lemma *weakly-fair-def2*:
shows $\mathit{weakly\text{-}fair}\ \mathit{enabled}\ \mathit{taken} = \Box(\neg\Box(\mathit{enabled} \wedge \neg \mathit{taken}))$
⟨*proof*⟩

lemma *weakly-fair-def3*:
shows $\mathit{weakly\text{-}fair}\ \mathit{enabled}\ \mathit{taken} = (\Diamond\Box \mathit{enabled} \longrightarrow \Box\Diamond \mathit{taken})$
⟨*proof*⟩

lemma *weakly-fair-def4*:
shows $\mathit{weakly\text{-}fair}\ \mathit{enabled}\ \mathit{taken} = \Box\Diamond(\mathit{enabled} \longrightarrow \mathit{taken})$
⟨*proof*⟩

lemma *mp-weakly-fair*:
assumes $\mathit{weakly\text{-}fair}\ \mathit{enabled}\ \mathit{taken}\ \sigma$
assumes $(\Box \mathit{enabled})\ \sigma$
shows $(\Diamond \mathit{taken})\ \sigma$
⟨*proof*⟩

lemma *always-weakly-fair*:
shows $\Box(\mathit{weakly\text{-}fair}\ \mathit{enabled}\ \mathit{taken}) = \mathit{weakly\text{-}fair}\ \mathit{enabled}\ \mathit{taken}$
⟨*proof*⟩

lemma *eventually-weakly-fair*:
shows $\Diamond(\mathit{weakly\text{-}fair}\ \mathit{enabled}\ \mathit{taken}) = \mathit{weakly\text{-}fair}\ \mathit{enabled}\ \mathit{taken}$
⟨*proof*⟩

lemma *weakly-fair-weaken*:
assumes $(\mathit{enabled}' \hookrightarrow \mathit{enabled})\ \sigma$
assumes $(\mathit{taken} \hookrightarrow \mathit{taken}')\ \sigma$
shows $(\mathit{weakly\text{-}fair}\ \mathit{enabled}\ \mathit{taken} \hookrightarrow \mathit{weakly\text{-}fair}\ \mathit{enabled}'\ \mathit{taken}')\ \sigma$
⟨*proof*⟩

lemma *weakly-fair-unless-until*:
shows $(\mathit{weakly\text{-}fair}\ \mathit{enabled}\ \mathit{taken} \wedge (\mathit{enabled} \hookrightarrow \mathit{enabled}\ \mathcal{W}\ \mathit{taken})) = (\mathit{enabled} \hookrightarrow \mathit{enabled}\ \mathcal{U}\ \mathit{taken})$
⟨*proof*⟩

lemma *stable-leads-to-eventually*:
assumes $(\mathit{enabled} \hookrightarrow \bigcirc(\mathit{enabled} \vee \mathit{taken}))\ \sigma$
shows $(\mathit{enabled} \hookrightarrow (\Box \mathit{enabled} \vee \Diamond \mathit{taken}))\ \sigma$
⟨*proof*⟩

lemma *weakly-fair-stable-leads-to*:
assumes $(\mathit{weakly\text{-}fair}\ \mathit{enabled}\ \mathit{taken})\ \sigma$
assumes $(\mathit{enabled} \hookrightarrow \bigcirc(\mathit{enabled} \vee \mathit{taken}))\ \sigma$
shows $(\mathit{enabled} \leadsto \mathit{taken})\ \sigma$
⟨*proof*⟩

lemma *weakly-fair-stable-leads-to-via*:
assumes $(\mathit{weakly\text{-}fair}\ \mathit{enabled}\ \mathit{taken})\ \sigma$
assumes $(\mathit{enabled} \hookrightarrow \bigcirc(\mathit{enabled} \vee \mathit{taken}))\ \sigma$
shows $(\mathit{enabled} \hookrightarrow \mathit{enabled}\ \mathcal{U}\ \mathit{taken})\ \sigma$
⟨*proof*⟩

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 $= \Box(\neg\Box(\Diamond enabled \wedge \neg taken))$
⟨*proof*⟩

lemma *strongly-fair-def3*:
strongly-fair enabled taken $= (\Box\Diamond enabled \longrightarrow \Box\Diamond taken)$
⟨*proof*⟩

lemma *always-strongly-fair*:
$\Box$(*strongly-fair enabled taken*) $=$ *strongly-fair enabled taken*
⟨*proof*⟩

lemma *eventually-strongly-fair*:
$\Diamond$(*strongly-fair enabled taken*) $=$ *strongly-fair enabled taken*
⟨*proof*⟩

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*)
⟨*proof*⟩

lemma *strongly-fair-imp-weakly-fair*:
assumes *strongly-fair enabled taken* σ
shows *weakly-fair enabled taken* σ
⟨*proof*⟩

lemma *always-enabled-weakly-fair-strongly-fair*:
assumes $(\Box enabled)$ σ
shows *weakly-fair enabled taken* σ $=$ *strongly-fair enabled taken* σ
⟨*proof*⟩

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* $\lceil P \rceil$
⟨*proof*⟩

lemma *safety-Next*:
assumes *safety P*
shows *safety* $(\bigcirc P)$
⟨*proof*⟩

lemma *safety-unless*:
assumes *safety P*
assumes *safety Q*
shows *safety* $(P\ \mathcal{W}\ Q)$
⟨*proof*⟩

lemma *safety-always*:
assumes *safety P*
shows *safety* $(\Box P)$
⟨*proof*⟩

lemma *absolute-liveness-eventually*:
  shows *absolute-liveness* $P \longleftrightarrow (\exists\,\sigma.\ P\ \sigma) \wedge P = \Diamond P$
⟨*proof*⟩

lemma *stable-always*:
  shows *stable* $P \longleftrightarrow (\exists\,\sigma.\ P\ \sigma) \wedge P = \Box P$
⟨*proof*⟩

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*⌉)
⟨*proof*⟩

lemma *fairness-strongly-fair*:
  assumes $\exists\, s.$ *taken s*
  shows *fairness* (*strongly-fair* ⌈*enabled*⌉ ⌈*taken*⌉)
⟨*proof*⟩

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* ($\lambda s.$ {*s*})

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

⟨*ML*⟩

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

$\llbracket\ \alpha = action\ s;\ s' \in val\ \beta\ s\ \rrbracket \Longrightarrow (\{\!|l|\!\}\ Request\ action\ val\ \#\ cs,\ \text{-},\ s) \rightarrow_{«\alpha,\ \beta»} (cs,\ Some\ l,\ s')$

$|\ (s',\ \beta) \in action\ \alpha\ s \Longrightarrow (\{\!|l|\!\}\ Response\ action\ \#\ cs,\ \text{-},\ s) \rightarrow_{»\alpha,\ \beta«} (cs,\ Some\ l,\ s')$

$|\ s' \in R\ s \Longrightarrow (\{\!|l|\!\}\ LocalOp\ R\ \#\ cs,\ \text{-},\ s) \rightarrow_{\tau} (cs,\ Some\ l,\ s')$

$|\ b\ s \Longrightarrow (\{\!|l|\!\}\ IF\ b\ THEN\ c\ FI\ \#\ cs,\ \text{-},\ s) \rightarrow_{\tau} (c\ \#\ cs,\ Some\ l,\ s)$

$|\ \neg b\ s \Longrightarrow (\{\!|l|\!\}\ IF\ b\ THEN\ c\ FI\ \#\ cs,\ \text{-},\ s) \rightarrow_{\tau} (cs,\ Some\ l,\ s)$

$|\ b\ s \Longrightarrow (\{\!|l|\!\}\ IF\ b\ THEN\ c1\ ELSE\ c2\ FI\ \#\ cs,\ \text{-},\ s) \rightarrow_{\tau} (c1\ \#\ cs,\ Some\ l,\ s)$

$|\ \neg b\ s \Longrightarrow (\{\!|l|\!\}\ IF\ b\ THEN\ c1\ ELSE\ c2\ FI\ \#\ cs,\ \text{-},\ s) \rightarrow_{\tau} (c2\ \#\ cs,\ Some\ l,\ s)$

$|\ (c\ \#\ LOOP\ DO\ c\ OD\ \#\ cs,\ s) \rightarrow_{\alpha} (cs',\ s') \Longrightarrow (LOOP\ DO\ c\ OD\ \#\ cs,\ s) \rightarrow_{\alpha} (cs',\ s')$

$|\ b\ s \Longrightarrow (\{\!|l|\!\}\ WHILE\ b\ DO\ c\ OD\ \#\ cs,\ \text{-},\ s) \rightarrow_{\tau} (c\ \#\ \{\!|l|\!\}\ WHILE\ b\ DO\ c\ OD\ \#\ cs,\ Some\ l,\ s)$

$|\ \neg\ b\ s \Longrightarrow (\{\!|l|\!\}\ WHILE\ b\ DO\ c\ OD\ \#\ cs,\ \text{-},\ s) \rightarrow_{\tau} (cs,\ Some\ l,\ s)$

$|\ (c1\ \#\ c2\ \#\ cs,\ s) \rightarrow_{\alpha} (cs',\ s') \Longrightarrow (c1;;\ c2\ \#\ cs,\ s) \rightarrow_{\alpha} (cs',\ s')$

$|\ Choose1\!:\ (c1\ \#\ cs,\ s) \rightarrow_{\alpha} (cs',\ s') \Longrightarrow (c1 \oplus c2\ \#\ cs,\ s) \rightarrow_{\alpha} (cs',\ s')$

$|\ Choose2\!:\ (c2\ \#\ cs,\ s) \rightarrow_{\alpha} (cs',\ s') \Longrightarrow (c1 \oplus c2\ \#\ cs,\ s) \rightarrow_{\alpha} (cs',\ s')$

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* ⇒ ('*answer*, '*location*, '*question*, '*state*) *local-state*

type-synonym ('*proc*, '*state*) *local-states*
  = '*proc* ⇒ '*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 :: $'proc\ set$
    $\Rightarrow ('answer, 'location, 'proc, 'question, 'state)\ system\text{-}state$
    $\Rightarrow ('answer, 'location, 'proc, 'question, 'state)\ system\text{-}state$
    $\Rightarrow bool$
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*
  $= ('answer, 'location, 'proc, 'question, 'state)\ system\text{-}state \Rightarrow bool$

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*
  $= ('proc, 'state)\ local\text{-}states \Rightarrow bool$

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

abbreviation (*input*) *LSTP* :: $('proc, 'state)\ local\text{-}state\text{-}pred$
    $\Rightarrow ('answer, 'location, 'proc, 'question, 'state)\ state\text{-}pred$ **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 $'location\ label = 'location\ set$

[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.

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 :: ('$answer$, '$location$, '$proc$, '$question$, '$state$) $system\text{-}state$ ⇒ '$proc$ ⇒ '$location$ $label$ **where**
  AT s p = $atCs$ ($cPGM$ (GST s p))

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

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

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

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

definition $atPLs$ :: ('$proc$ × '$location$ $label$) set ⇒ ('$answer$, '$location$, '$proc$, '$question$, '$state$) $state\text{-}pred$ **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 $taken$ :: '$proc$ ⇒ '$location$ ⇒ ('$answer$, '$location$, '$proc$, '$question$, '$state$) $state\text{-}pred$ **where**
  $taken$ p l s ⟷ $cTKN$ (GST s p) = $Some$ l

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

abbreviation ($input$) $terminated$ :: '$proc$ ⇒ ('$answer$, '$location$, '$proc$, '$question$, '$state$) $state\text{-}pred$ **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*
    $\Rightarrow ({'}answer, {'}location, {'}proc, {'}question, {'}state)$ *global-state*
    $\Rightarrow$ *bool*
where
  $\textit{initial-state sys s} = ((\forall p.\ \textit{cPGM}\ (s\ p) = [\textit{PGMs sys p}] \wedge \textit{cTKN}\ (s\ p) = \textit{None}) \wedge \textit{INIT sys}\ (\textit{cLST} \circ s))$

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
  $\textit{system-step-reflclp}\ \sigma \longleftrightarrow (\lambda sh\ sh'.\ \exists pls.\ \textit{system-step pls}\ sh'\ sh)^{==}\ (\sigma\ 0)\ (\sigma\ 1)$

definition
  prerun :: $({'}answer, {'}location, {'}proc, {'}question, {'}state, {'}ext)$ *pre-system-ext*
    $\Rightarrow ({'}answer, {'}location, {'}proc, {'}question, {'}state)$ *system-state seq-pred*
where
  $\textit{prerun sys} = ((\lambda\sigma.\ \textit{initial-state sys}\ (\textit{GST}\ (\sigma\ 0)) \wedge \textit{HST}\ (\sigma\ 0) = []) \wedge \Box \textit{system-step-reflclp})$

definition — state-based invariants only
  prerun-valid :: $({'}answer, {'}location, {'}proc, {'}question, {'}state, {'}ext)$ *pre-system-ext*
    $\Rightarrow ({'}answer, {'}location, {'}proc, {'}question, {'}state)$ *state-pred* $\Rightarrow$ *bool* $(\langle\text{-} \models_{pre} \text{-}\rangle\ [11, 0]\ 11)$
where
  $(\textit{sys} \models_{pre} \varphi) \longleftrightarrow (\forall \sigma.\ \textit{prerun sys}\ \sigma \longrightarrow (\Box\lceil\varphi\rceil)\ \sigma)$

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*
    $\Rightarrow ({'}answer, {'}location, {'}proc, {'}question, {'}state)$ *system-state seq-pred*
where
  $\textit{run sys} = (\textit{prerun sys} \wedge \textit{FAIR sys})$

definition
  valid :: $({'}answer, {'}location, {'}proc, {'}question, {'}state)$ *system*
    $\Rightarrow ({'}answer, {'}location, {'}proc, {'}question, {'}state)$ *system-state seq-pred* $\Rightarrow$ *bool* $(\langle\text{-} \models \text{-}\rangle\ [11, 0]\ 11)$
where
  $(\textit{sys} \models \varphi) \longleftrightarrow (\forall \sigma.\ \textit{run sys}\ \sigma \longrightarrow \varphi\ \sigma)$

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

lemma *small-step-stuck*:
$\neg\ ([],\ s) \rightarrow_\alpha c'$
⟨*proof*⟩

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

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

lemma *atS-state-weak-cong*[*cong*]:
$AT\ s\ p = AT\ s'\ p \Longrightarrow atS\ p\ ls\ s \longleftrightarrow atS\ p\ ls\ s'$
⟨*proof*⟩

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

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

lemma *atS-mono*:
$[\![atS\ p\ ls\ s;\ ls \subseteq ls']\!] \Longrightarrow atS\ p\ ls'\ s$
⟨*proof*⟩

lemma *atS-un*:
$atS\ p\ (l \cup l')\ s \longleftrightarrow atS\ p\ l\ s \lor atS\ p\ l'\ s$
⟨*proof*⟩

lemma *atLs-disj-union*[*simp*]:
$(atLs\ p\ label0 \lor atLs\ p\ label1) = atLs\ p\ (label0 \cup label1)$
⟨*proof*⟩

lemma *atLs-insert-disj*:
$atLs\ p\ (insert\ l\ label0) = (atL\ p\ l \lor atLs\ p\ label0)$
⟨*proof*⟩

lemma *small-step-terminated*:
$s \rightarrow_x s' \Longrightarrow atCs\ (fst\ s) = \{\} \Longrightarrow atCs\ (fst\ s') = \{\}$
⟨*proof*⟩

lemma *atC-not-empty*:

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

$atC\ c \neq \{\}$
⟨*proof*⟩

lemma *atCs-empty*:
$atCs\ cs = \{\} \longleftrightarrow cs = []$
⟨*proof*⟩

lemma *terminated-no-commands*:
assumes *terminated p sh*
shows $\exists\, s.\ GST\ sh\ p = ([], s)$
⟨*proof*⟩

lemma *terminated-GST-stable*:
assumes *system-step q sh′ sh*
assumes *terminated p sh*
shows $GST\ sh\ p = GST\ sh'\ p$
⟨*proof*⟩

lemma *terminated-stable*:
assumes *system-step q sh′ sh*
assumes *terminated p sh*
shows *terminated p sh′*
⟨*proof*⟩

lemma *system-step-pls-nonempty*:
assumes *system-step pls sh′ sh*
shows $pls \neq \{\}$
⟨*proof*⟩

lemma *system-step-no-change*:
assumes *system-step ps sh′ sh*
assumes $p \notin ps$
shows $GST\ sh'\ p = GST\ sh\ p$
⟨*proof*⟩

lemma *initial-stateD*:
assumes *initial-state sys s*
shows $AT\ (\llparenthesis GST = s,\ HST = [] \rrparenthesis) = atC \circ PGMs\ sys \wedge INIT\ sys\ (\llparenthesis GST = s,\ HST = [] \rrparenthesis)\!\downarrow \wedge (\forall\, p\ l.\ \neg taken\ p\ l\ (\llparenthesis GST = s,\ HST = [] \rrparenthesis))$
⟨*proof*⟩

lemma *initial-states-initial*[*iff*]:
assumes *initial-state sys s*
shows $at\ p\ l\ (\llparenthesis GST = s,\ HST = [] \rrparenthesis) \longleftrightarrow l \in atC\ (PGMs\ sys\ p)$
⟨*proof*⟩

definition
reachable-state :: (′*answer*, ′*location*, ′*proc*, ′*question*, ′*state*, ′*ext*) *pre-system-ext*
⇒ (′*answer*, ′*location*, ′*proc*, ′*question*, ′*state*) *state-pred*
where
$reachable\text{-}state\ sys\ s \longleftrightarrow (\exists\, \sigma\ i.\ prerun\ sys\ \sigma \wedge \sigma\ i = s)$

lemma *reachable-stateE*:
assumes *reachable-state sys sh*
assumes $\bigwedge \sigma\ i.\ prerun\ sys\ \sigma \Longrightarrow P\ (\sigma\ i)$
shows *P sh*
⟨*proof*⟩

lemma *prerun-reachable-state*:
  assumes *prerun sys* σ
  shows *reachable-state sys* $(\sigma\ i)$
$\langle proof \rangle$

lemma *reachable-state-induct*[*consumes 1*, *case-names init LocalStep CommunicationStep*, *induct set*: *reachable-state*]:
  assumes *r*: *reachable-state sys sh*
  assumes *i*: $\bigwedge s.\ \textit{initial-state sys s} \Longrightarrow P\ (\!|GST = s,\ HST = []|\!)$
  assumes *l*: $\bigwedge sh\ ls'\ p.\ [\![\textit{reachable-state sys sh};\ P\ sh;\ GST\ sh\ p \rightarrow_{\tau}\ ls']\!] \Longrightarrow P\ (\!|GST = (GST\ sh)(p := ls'),\ HST = HST\ sh|\!)$
  assumes *c*: $\bigwedge sh\ ls1'\ ls2'\ p1\ p2\ \alpha\ \beta.$
$[\![\textit{reachable-state sys sh};\ P\ sh;$
$GST\ sh\ p1 \rightarrow_{«\alpha,\ \beta»}\ ls1';\ GST\ sh\ p2 \rightarrow_{»\alpha,\ \beta«}\ ls2';\ p1 \neq p2\ ]\!]$
$\Longrightarrow P\ (\!|GST = (GST\ sh)(p1 := ls1',\ p2 := ls2'),\ HST = HST\ sh\ @\ [(\alpha,\ \beta)]|\!)$
  shows *P sh*
$\langle proof \rangle$

lemma *prerun-valid-TrueI*:
  shows $sys \models_{pre} \langle True \rangle$
$\langle proof \rangle$

lemma *prerun-valid-conjI*:
  assumes $sys \models_{pre} P$
  assumes $sys \models_{pre} Q$
  shows $sys \models_{pre} P \land Q$
$\langle proof \rangle$

lemma *valid-prerun-lift*:
  assumes $sys \models_{pre} I$
  shows $sys \models \Box\lceil I \rceil$
$\langle proof \rangle$

lemma *prerun-valid-induct*:
  assumes $\bigwedge\sigma.\ \textit{prerun sys}\ \sigma \Longrightarrow \lceil I \rceil\ \sigma$
  assumes $\bigwedge\sigma.\ \textit{prerun sys}\ \sigma \Longrightarrow (\lceil I \rceil \hookrightarrow (\bigcirc\lceil I \rceil))\ \sigma$
  shows $sys \models_{pre} I$
$\langle proof \rangle$

lemma *prerun-validI*:
  assumes $\bigwedge s.\ \textit{reachable-state sys s} \Longrightarrow I\ s$
  shows $sys \models_{pre} I$
$\langle proof \rangle$

lemma *prerun-validE*:
  assumes *reachable-state sys s*
  assumes $sys \models_{pre} I$
  shows *I s*
$\langle proof \rangle$

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*
  $= 'state \Rightarrow 'location\ set$

fun *lconst* :: $'location\ set \Rightarrow ('answer, 'location, 'question, 'state)\ loc\text{-}comp$ **where**
  $lconst\ lp\ s = lp$

definition *lcond* :: $'location\ set \Rightarrow 'location\ set \Rightarrow 'state\ bexp$
    $\Rightarrow ('answer, 'location, 'question, 'state)\ loc\text{-}comp$ **where**
  $lcond\ lp\ lp'\ b\ s = (if\ b\ s\ then\ lp\ else\ lp')$

lemma *lcond-split*:
  $Q\ (lcond\ lp\ lp'\ b\ s) \longleftrightarrow (b\ s \longrightarrow Q\ lp) \wedge (\neg b\ s \longrightarrow Q\ lp')$
$\langle proof \rangle$

lemma *lcond-split-asm*:
  $Q\ (lcond\ lp\ lp'\ b\ s) \longleftrightarrow \neg\ ((b\ s \wedge \neg Q\ lp) \vee (\neg b\ s \wedge \neg\ Q\ lp'))$
$\langle proof \rangle$

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

fun
  fragments :: $('answer, 'location, 'question, 'state)\ com$
    $\Rightarrow 'location\ set$
    $\Rightarrow (\ ('answer, 'location, 'question, 'state)\ com$
    $\times ('answer, 'location, 'question, 'state)\ loc\text{-}comp\ )\ set$
where
  $fragments\ (\{\!|l|\!\}\ IF\ b\ THEN\ c\ FI)\ aft$
    $= \{\ (\{\!|l|\!\}\ IF\ b\ THEN\ c'\ FI,\ lcond\ (atC\ c)\ aft\ b)\ |c'.\ True\ \}$
    $\cup\ fragments\ c\ aft$
$|\ fragments\ (\{\!|l|\!\}\ IF\ b\ THEN\ c1\ ELSE\ c2\ FI)\ aft$
    $= \{\ (\{\!|l|\!\}\ IF\ b\ THEN\ c1'\ ELSE\ c2'\ FI,\ lcond\ (atC\ c1)\ (atC\ c2)\ b)\ |c1'\ c2'.\ True\ \}$
    $\cup\ fragments\ c1\ aft \cup fragments\ c2\ aft$
$|\ fragments\ (LOOP\ DO\ c\ OD)\ aft = fragments\ c\ (atC\ c)$
$|\ fragments\ (\{\!|l|\!\}\ WHILE\ b\ DO\ c\ OD)\ aft$
    $=\ fragments\ c\ \{l\} \cup \{\ (\{\!|l|\!\}\ WHILE\ b\ DO\ c'\ OD,\ lcond\ (atC\ c)\ aft\ b)\ |c'.\ True\ \}$
$|\ fragments\ (c1;;\ c2)\ aft = fragments\ c1\ (atC\ c2) \cup fragments\ c2\ aft$
$|\ fragments\ (c1\ \oplus\ c2)\ aft = fragments\ c1\ aft \cup fragments\ c2\ aft$
$|\ fragments\ c\ aft = \{\ (c,\ lconst\ aft)\ \}$

fun
  fragmentsL :: $('answer, 'location, 'question, 'state)\ com\ list$
    $\Rightarrow (\ ('answer, 'location, 'question, 'state)\ com$
    $\times ('answer, 'location, 'question, 'state)\ loc\text{-}comp\ )\ set$
where
  $fragmentsL\ [] = \{\}$
$|\ fragmentsL\ [c] = fragments\ c\ \{\}$
$|\ fragmentsL\ (c\ \#\ c'\ \#\ cs) = fragments\ c\ (atC\ c') \cup fragmentsL\ (c'\ \#\ cs)$

abbreviation
  fragmentsLS :: $('answer, 'location, 'question, 'state)\ local\text{-}state$
    $\Rightarrow (\ ('answer, 'location, 'question, 'state)\ com$
    $\times ('answer, 'location, 'question, 'state)\ loc\text{-}comp\ )\ set$
where
  $fragmentsLS\ s \equiv fragmentsL\ (cPGM\ s)$

We show that taking system steps preserves fragments.

lemma *small-step-fragmentsLS*:
  assumes $s \rightarrow_\alpha s'$

shows $\mathit{fragmentsLS}\ s' \subseteq \mathit{fragmentsLS}\ s$
$\langle proof \rangle$

lemma *reachable-state-fragmentsLS*:
assumes *reachable-state sys sh*
shows $\mathit{fragmentsLS}\ (\mathit{GST}\ sh\ p) \subseteq \mathit{fragments}\ (\mathit{PGMs}\ sys\ p)\ \{\}$
$\langle proof \rangle$

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

lemma *fragments-basic-com*:
assumes $(c', \mathit{aft}') \in \mathit{fragments}\ c\ \mathit{aft}$
shows $\mathit{basic\text{-}com}\ c'$
$\langle proof \rangle$

lemma *fragmentsL-basic-com*:
assumes $(c', \mathit{aft}') \in \mathit{fragmentsL}\ cs$
shows $\mathit{basic\text{-}com}\ c'$
$\langle proof \rangle$

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 $('\mathit{answer}, '\mathit{location}, '\mathit{question}, '\mathit{state})\ \mathit{ctxt}$
$= (('\mathit{answer}, '\mathit{location}, '\mathit{question}, '\mathit{state})\ \mathit{com} \Rightarrow ('\mathit{answer}, '\mathit{location}, '\mathit{question}, '\mathit{state})\ \mathit{com})$
$\times (('\mathit{answer}, '\mathit{location}, '\mathit{question}, '\mathit{state})\ \mathit{com} \Rightarrow ('\mathit{answer}, '\mathit{location}, '\mathit{question}, '\mathit{state})\ \mathit{com\ list})$

inductive-set
$\mathit{ctxt} :: ('\mathit{answer}, '\mathit{location}, '\mathit{question}, '\mathit{state})\ \mathit{ctxt\ set}$
where
$\mathit{C\text{-}Hole}: (\mathit{id}, \langle[]\rangle) \in \mathit{ctxt}$
$|\ \mathit{C\text{-}Loop}: (E, \mathit{fctxt}) \in \mathit{ctxt} \Longrightarrow (\lambda c1.\ \mathit{LOOP\ DO}\ E\ c1\ \mathit{OD}, \lambda c1.\ \mathit{fctxt}\ c1\ @\ [\mathit{LOOP\ DO}\ E\ c1\ \mathit{OD}]) \in \mathit{ctxt}$
$|\ \mathit{C\text{-}Seq}: (E, \mathit{fctxt}) \in \mathit{ctxt} \Longrightarrow (\lambda c1.\ E\ c1;;\ c2, \lambda c1.\ \mathit{fctxt}\ c1\ @\ [c2]) \in \mathit{ctxt}$
$|\ \mathit{C\text{-}Choose1}: (E, \mathit{fctxt}) \in \mathit{ctxt} \Longrightarrow (\lambda c1.\ E\ c1 \oplus c2, \mathit{fctxt}) \in \mathit{ctxt}$
$|\ \mathit{C\text{-}Choose2}: (E, \mathit{fctxt}) \in \mathit{ctxt} \Longrightarrow (\lambda c2.\ c1 \oplus E\ c2, \mathit{fctxt}) \in \mathit{ctxt}$

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

fun
$\mathit{decompose\text{-}com} :: ('\mathit{answer}, '\mathit{location}, '\mathit{question}, '\mathit{state})\ \mathit{com}$
$\Rightarrow (\ ('\mathit{answer}, '\mathit{location}, '\mathit{question}, '\mathit{state})\ \mathit{com}$
$\times ('\mathit{answer}, '\mathit{location}, '\mathit{question}, '\mathit{state})\ \mathit{ctxt}\ )\ \mathit{set}$
where
$\mathit{decompose\text{-}com}\ (\mathit{LOOP\ DO}\ c1\ \mathit{OD}) = \{\ (c, \lambda t.\ \mathit{LOOP\ DO\ ictxt}\ t\ \mathit{OD}, \lambda t.\ \mathit{fctxt}\ t\ @\ [\mathit{LOOP\ DO\ ictxt}\ t\ \mathit{OD}])\ |c\ \mathit{fctxt\ ictxt}.\ (c, \mathit{ictxt}, \mathit{fctxt}) \in \mathit{decompose\text{-}com}\ c1\ \}$
$|\ \mathit{decompose\text{-}com}\ (c1;;\ c2) = \{\ (c, \lambda t.\ \mathit{ictxt}\ t;;\ c2, \lambda t.\ \mathit{fctxt}\ t\ @\ [c2])\ |c\ \mathit{fctxt\ ictxt}.\ (c, \mathit{ictxt}, \mathit{fctxt}) \in \mathit{decompose\text{-}com}\ c1\ \}$
$|\ \mathit{decompose\text{-}com}\ (c1 \oplus c2) = \{\ (c, \lambda t.\ \mathit{ictxt}\ t \oplus c2, \mathit{fctxt})\ |c\ \mathit{fctxt\ ictxt}.\ (c, \mathit{ictxt}, \mathit{fctxt}) \in \mathit{decompose\text{-}com}\ c1\ \}$

$\cup$ { (c, $\lambda t.$ $c1 \oplus ictxt\ t$, $fctxt$) |c $fctxt$ $ictxt$. (c, $ictxt$, $fctxt$) $\in$ $decompose\text{-}com\ c2$ }
| $decompose\text{-}com\ c$ = {(c, id, $\langle[]\rangle$)}

definition
  $decomposeLS$ :: ($'answer$, $'location$, $'question$, $'state$) $local\text{-}state$
    $\Rightarrow$ (($'answer$, $'location$, $'question$, $'state$) com
    $\times$ (($'answer$, $'location$, $'question$, $'state$) com $\Rightarrow$ ($'answer$, $'location$, $'question$, $'state$) com)
    $\times$ (($'answer$, $'location$, $'question$, $'state$) com $\Rightarrow$ ($'answer$, $'location$, $'question$, $'state$) $com\ list$)) set
where
  $decomposeLS\ s$ = ($case$ $cPGM\ s$ of c # - $\Rightarrow$ $decompose\text{-}com\ c$ | - $\Rightarrow$ {})

lemma $ctxt\text{-}inj$:
  assumes (E, $fctxt$) $\in$ $ctxt$
  assumes $E\ x = E\ y$
  shows $x = y$
$\langle proof \rangle$

lemma $decompose\text{-}com\text{-}non\text{-}empty$: $decompose\text{-}com\ c \neq$ {}
$\langle proof \rangle$

lemma $decompose\text{-}com\text{-}basic\text{-}com$:
  assumes (c', $ctxts$) $\in$ $decompose\text{-}com\ c$
  shows $basic\text{-}com\ c'$
$\langle proof \rangle$

lemma $decomposeLS\text{-}basic\text{-}com$:
  assumes (c', $ctxts$) $\in$ $decomposeLS\ s$
  shows $basic\text{-}com\ c'$
$\langle proof \rangle$

lemma $decompose\text{-}com\text{-}ctxt$:
  assumes (c', $ctxts$) $\in$ $decompose\text{-}com\ c$
  shows $ctxts \in ctxt$
$\langle proof \rangle$

lemma $decompose\text{-}com\text{-}ictxt$:
  assumes (c', $ictxt$, $fctxt$) $\in$ $decompose\text{-}com\ c$
  shows $ictxt\ c' = c$
$\langle proof \rangle$

lemma $decompose\text{-}com\text{-}small\text{-}step$:
  assumes as: (c' # $fctxt\ c'$ @ cs, s) $\rightarrow_\alpha$ s'
  assumes ds: (c', $ictxt$, $fctxt$) $\in$ $decompose\text{-}com\ c$
  shows (c # cs, s) $\rightarrow_\alpha$ s'
$\langle proof \rangle$

theorem $context\text{-}decompose$:
  $s \rightarrow_\alpha s' \longleftrightarrow$ ($\exists$(c, $ictxt$, $fctxt$) $\in$ $decomposeLS\ s$.
    $cPGM\ s = ictxt\ c$ # tl ($cPGM\ s$)
    $\wedge$ (c # $fctxt\ c$ @ tl ($cPGM\ s$), $cTKN\ s$, $cLST\ s$) $\rightarrow_\alpha$ s'
    $\wedge$ ($\forall\ l{\in}atC\ c$. $cTKN\ s' = Some\ l$)) (**is** $?lhs = ?rhs$)
$\langle proof \rangle$

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
  $loc\text{-}compC$ :: ($'answer$, $'location$, $'question$, $'state$) com

$\Rightarrow$ ('*answer*, '*location*, '*question*, '*state*) *com list*
$\Rightarrow$ ('*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*) $\in$ *decompose-com c0*
shows (*c*, *loc-compC c* (*fctxt c* @ *cs*)) $\in$ *fragments c0* (*atCs cs*)
$\langle$*proof*$\rangle$

lemma *at-decompose*:
assumes (*c*, *ictxt*, *fctxt*) $\in$ *decompose-com c0*
shows *atC c* $\subseteq$ *atC c0*
$\langle$*proof*$\rangle$

lemma *at-decomposeLS*:
assumes (*c*, *ictxt*, *fctxt*) $\in$ *decomposeLS s*
shows *atC c* $\subseteq$ *atCs* (*cPGM s*)
$\langle$*proof*$\rangle$

lemma *decomposeLS-fragmentsLS*:
assumes (*c*, *ictxt*, *fctxt*) $\in$ *decomposeLS s*
shows (*c*, *loc-compC c* (*fctxt c* @ *tl* (*cPGM s*))) $\in$ *fragmentsLS s*
$\langle$*proof*$\rangle$

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'*)
$\langle$*proof*$\rangle$

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*
$\langle$*proof*$\rangle$

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*
    ⇒ ′*proc*
    ⇒ (′*answer*, ′*location*, ′*question*, ′*state*) *loc-comp*
    ⇒ (′*answer*, ′*location*, ′*proc*, ′*question*, ′*state*) *state-pred*
    ⇒ (′*answer*, ′*location*, ′*question*, ′*state*) *com*
    ⇒ (′*answer*, ′*location*, ′*proc*, ′*question*, ′*state*) *state-pred*
    ⇒ *bool* (‹-, -, - ⊢/ ⦃-⦄/ -/ ⦃-⦄› [*11*,*0*,*0*,*0*,*0*,*0*] *11*)
where
  ⟦ ⋀*aft′ action′ s ps′ p′s′ l′ β s′ p′*.
    ⟦ *pre s*; (⦃*l′*⦄ *Response action′*, *aft′*) ∈ *fragments* (*coms p′*) {}; *p* ≠ *p′*;
      ps′ ∈ *val β* (*s*↓ *p*); (*p′s′*, *β*) ∈ *action′* (*action* (*s*↓ *p*)) (*s*↓ *p′*);
      at p l s; *at p′ l′ s*;
      AT s′ = (*AT s*)(*p* := *aft* (*s*↓ *p*), *p′* := *aft′* (*s*↓ *p′*));
      s′↓ = *s*↓(*p* := *ps′*, *p′* := *p′s′*);
      taken p l s′;
      HST s′ = *HST s* @ [(*action* (*s*↓ *p*), *β*)];
      ∀ *p″*∈−{*p*,*p′*}. *GST s′ p″* = *GST s p″*
    ⟧ ⟹ *post s′*
  ⟧ ⟹ *coms*, *p*, *aft* ⊢ ⦃*pre*⦄ ⦃*l*⦄ *Request action val* ⦃*post*⦄
| ⟦ ⋀*s ps′ s′*.
    ⟦ *pre s*; *ps′* ∈ *f* (*s*↓ *p*);
      at p l s;
      AT s′ = (*AT s*)(*p* := *aft* (*s*↓ *p*));
      s′↓ = *s*↓(*p* := *ps′*);
      taken p l s′;
      HST s′ = *HST s*;
      ∀ *p″*∈−{*p*}. *GST s′ p″* = *GST s p″*
    ⟧ ⟹ *post s′*
  ⟧ ⟹ *coms*, *p*, *aft* ⊢ ⦃*pre*⦄ ⦃*l*⦄ *LocalOp f* ⦃*post*⦄
| ⟦ ⋀*s s′*.
    ⟦ *pre s*;
      at p l s;
      AT s′ = (*AT s*)(*p* := *aft* (*s*↓ *p*));
      s′↓ = *s*↓;
      taken p l s′;
      HST s′ = *HST s*;
      ∀ *p″*∈−{*p*}. *GST s′ p″* = *GST s p″*
    ⟧ ⟹ *post s′*
  ⟧ ⟹ *coms*, *p*, *aft* ⊢ ⦃*pre*⦄ ⦃*l*⦄ *IF b THEN t FI* ⦃*post*⦄
| ⟦ ⋀*s s′*.
    ⟦ *pre s*;
      at p l s;
      AT s′ = (*AT s*)(*p* := *aft* (*s*↓ *p*));
      s′↓ = *s*↓;
      taken p l s′;
      HST s′ = *HST s*;
      ∀ *p″*∈−{*p*}. *GST s′ p″* = *GST s p″*
    ⟧ ⟹ *post s′*
  ⟧ ⟹ *coms*, *p*, *aft* ⊢ ⦃*pre*⦄ ⦃*l*⦄ *IF b THEN t ELSE e FI* ⦃*post*⦄
| ⟦ ⋀*s s′*.
    ⟦ *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
⇒ 'location set
⇒ (('answer, 'location, 'question, 'state) com
× ('answer, 'location, 'question, 'state) loc-comp) set
where
$vcg\text{-}fragments'\ (\{\!|l|\!\}\ Response\ action)\ aft = \{\}$
$|\ vcg\text{-}fragments'\ (\{\!|l|\!\}\ IF\ b\ THEN\ c\ FI)\ aft$
$= vcg\text{-}fragments'\ c\ aft$
$\cup \{\ (\{\!|l|\!\}\ IF\ b\ THEN\ c'\ FI,\ lcond\ (atC\ c)\ aft\ b)\ |c'.\ True\ \}$
$|\ vcg\text{-}fragments'\ (\{\!|l|\!\}\ IF\ b\ THEN\ c1\ ELSE\ c2\ FI)\ aft$
$= vcg\text{-}fragments'\ c2\ aft \cup vcg\text{-}fragments'\ c1\ aft$
$\cup \{\ (\{\!|l|\!\}\ IF\ b\ THEN\ c1'\ ELSE\ c2'\ FI,\ lcond\ (atC\ c1)\ (atC\ c2)\ b)\ |c1'\ c2'.\ True\ \}$
$|\ vcg\text{-}fragments'\ (LOOP\ DO\ c\ OD)\ aft = vcg\text{-}fragments'\ c\ (atC\ c)$
$|\ vcg\text{-}fragments'\ (\{\!|l|\!\}\ WHILE\ b\ DO\ c\ OD)\ aft$
$= vcg\text{-}fragments'\ c\ \{l\} \cup \{\ (\{\!|l|\!\}\ WHILE\ b\ DO\ c'\ OD,\ lcond\ (atC\ c)\ aft\ b)\ |c'.\ True\ \}$
$|\ vcg\text{-}fragments'\ (c1\ ;;\ c2)\ aft = vcg\text{-}fragments'\ c2\ aft \cup vcg\text{-}fragments'\ c1\ (atC\ c2)$
$|\ vcg\text{-}fragments'\ (c1 \oplus c2)\ aft = vcg\text{-}fragments'\ c1\ aft \cup vcg\text{-}fragments'\ c2\ aft$
$|\ vcg\text{-}fragments'\ c\ aft = \{(c,\ lconst\ aft)\}$

abbreviation
vcg-fragments :: ('answer, 'location, 'question, 'state) com
⇒ (('answer, 'location, 'question, 'state) com
× ('answer, 'location, 'question, 'state) loc-comp) set
where
$vcg\text{-}fragments\ c \equiv vcg\text{-}fragments'\ c\ \{\}$

fun *isResponse* :: ('answer, 'location, 'question, 'state) com ⇒ bool **where**
$isResponse\ (\{\!|l|\!\}\ Response\ action) \longleftrightarrow True$
$|\ isResponse\ \text{-} \longleftrightarrow False$

lemma *fragments-vcg-fragments'*:
$[\![$ $(c, aft) \in fragments\ c'\ aft'$; $\neg isResponse\ c$ $]\!] \Longrightarrow (c, aft) \in vcg\text{-}fragments'\ c'\ aft'$
⟨*proof*⟩

lemma *vcg-fragments'-fragments*:
$vcg\text{-}fragments'\ c'\ aft' \subseteq fragments\ c'\ aft'$
⟨*proof*⟩

lemma *VCG-step*:
assumes V: $\bigwedge p.\ \forall (c, aft) \in vcg\text{-}fragments\ (PGMs\ sys\ p).\ PGMs\ sys, p, aft \vdash \{\!|pre|\!\}\ c\ \{\!|post|\!\}$
assumes S: *system-step p sh' sh*
assumes R: *reachable-state sys sh*
assumes P: *pre sh*
shows *post sh'*
⟨*proof*⟩

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

lemma *VCG-step-inv-stable*:
assumes V: $\bigwedge p.\ \forall (c, aft) \in vcg\text{-}fragments\ (PGMs\ sys\ p).\ PGMs\ sys, p, aft \vdash \{\!|I|\!\}\ c$
assumes *prerun sys* σ
shows $(\lceil I \rceil \hookrightarrow \bigcirc\lceil I \rceil)\ \sigma$
⟨*proof*⟩

lemma *VCG*:
assumes I: $\forall s.\ initial\text{-}state\ sys\ s \longrightarrow I\ (\!|GST = s, HST = []|\!)$
assumes V: $\bigwedge p.\ \forall (c, aft) \in vcg\text{-}fragments\ (PGMs\ sys\ p).\ PGMs\ sys, p, aft \vdash \{\!|I|\!\}\ c$
shows $sys \models_{pre} I$
⟨*proof*⟩

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**
$valid\text{-}syn\ P\ c\ Q \equiv coms, p, aft \vdash \{\!|P|\!\}\ c\ \{\!|Q|\!\}$
notation *valid-syn* (‹{|-|}/ -/ {|-|}›)

abbreviation
valid-inv-syn :: (*'answer*, *'location*, *'proc*, *'question*, *'state*) *state-pred*
⇒ (*'answer*, *'location*, *'question*, *'state*) *com* ⇒ *bool* **where**
$valid\text{-}inv\text{-}syn\ P\ c \equiv \{\!|P|\!\}\ c\ \{\!|P|\!\}$
notation *valid-inv-syn* (‹{|-|}/ -›)

lemma *vcg-True*:
$\{\!|P|\!\}\ c\ \{\!|\langle True\rangle|\!\}$
⟨*proof*⟩

lemma *vcg-conj*:
$\llbracket$ $\{\!|I|\!\}$ c $\{\!|Q|\!\}$; $\{\!|I|\!\}$ c $\{\!|R|\!\}$ $\rrbracket \Longrightarrow \{\!|I|\!\}$ c $\{\!|Q \wedge R|\!\}$
$\langle proof \rangle$

lemma *vcg-pre-imp*:
$\llbracket \bigwedge s.\ P\ s \Longrightarrow Q\ s$; $\{\!|Q|\!\}$ c $\{\!|R|\!\}$ $\rrbracket \Longrightarrow \{\!|P|\!\}$ c $\{\!|R|\!\}$
$\langle proof \rangle$

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

lemma *vcg-post-imp*:
$\llbracket \bigwedge s.\ Q\ s \Longrightarrow R\ s$; $\{\!|P|\!\}$ c $\{\!|Q|\!\}$ $\rrbracket \Longrightarrow \{\!|P|\!\}$ c $\{\!|R|\!\}$
$\langle proof \rangle$

lemma *vcg-prop*[*intro*]:
$\{\!|\langle P \rangle|\!\}$ c
$\langle proof \rangle$

lemma *vcg-drop-imp*:
assumes $\{\!|P|\!\}$ c $\{\!|Q|\!\}$
shows $\{\!|P|\!\}$ c $\{\!|R \longrightarrow Q|\!\}$
$\langle proof \rangle$

lemma *vcg-conj-lift*:
assumes x: $\{\!|P|\!\}$ c $\{\!|Q|\!\}$
assumes y: $\{\!|P'|\!\}$ c $\{\!|Q'|\!\}$
shows $\{\!|P \wedge P'|\!\}$ c $\{\!|Q \wedge Q'|\!\}$
$\langle proof \rangle$

lemma *vcg-disj-lift*:
assumes x: $\{\!|P|\!\}$ c $\{\!|Q|\!\}$
assumes y: $\{\!|P'|\!\}$ c $\{\!|Q'|\!\}$
shows $\{\!|P \vee P'|\!\}$ c $\{\!|Q \vee Q'|\!\}$
$\langle proof \rangle$

lemma *vcg-imp-lift*:
assumes $\{\!|P'|\!\}$ c $\{\!|\neg P|\!\}$
assumes $\{\!|Q'|\!\}$ c $\{\!|Q|\!\}$
shows $\{\!|P' \vee Q'|\!\}$ c $\{\!|P \longrightarrow Q|\!\}$
$\langle proof \rangle$

lemma *vcg-ex-lift*:
assumes $\bigwedge x.$ $\{\!|P\ x|\!\}$ c $\{\!|Q\ x|\!\}$
shows $\{\!|\lambda s.\ \exists x.\ P\ x\ s|\!\}$ c $\{\!|\lambda s.\ \exists x.\ Q\ x\ s|\!\}$
$\langle proof \rangle$

lemma *vcg-all-lift*:
assumes $\bigwedge x.$ $\{\!|P\ x|\!\}$ c $\{\!|Q\ x|\!\}$
shows $\{\!|\lambda s.\ \forall x.\ P\ x\ s|\!\}$ c $\{\!|\lambda s.\ \forall x.\ Q\ x\ s|\!\}$
$\langle proof \rangle$

lemma *vcg-name-pre-state*:
assumes $\bigwedge s.\ P\ s \Longrightarrow \{\!|(=)\ s|\!\}$ c $\{\!|Q|\!\}$
shows $\{\!|P|\!\}$ c $\{\!|Q|\!\}$
$\langle proof \rangle$

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 $\{\!|\lambda s.\ Q\ (f\ s)\ s|\!\}\ c\ \{\!|\lambda s.\ P\ (f\ s)\ s|\!\}$
⟨*proof*⟩

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*:
$[\![\ eq\text{-}imp\ f\ g;\ \forall x.\ f\ x\ s = f\ x\ s'\ ]\!] \Longrightarrow g\ s = g\ s'$
⟨*proof*⟩

lemma *eq-imp-vcg*:
assumes g: *eq-imp f g*
assumes f: $\forall x\ P.\ \{\!|P \circ (f\ x)|\!\}\ c$
shows $\{\!|P \circ g|\!\}\ c$
⟨*proof*⟩

lemma *eq-imp-vcg-LST*:
assumes g: *eq-imp f g*
assumes f: $\forall x\ P.\ \{\!|P \circ (f\ x) \circ LST|\!\}\ c$
shows $\{\!|P \circ g \circ LST|\!\}\ c$
⟨*proof*⟩

lemma *eq-imp-fun-upd*:
assumes g: *eq-imp f g*
assumes f: $\forall x.\ f\ x\ (s(fld := val)) = f\ x\ s$
shows $g\ (s(fld := val)) = g\ s$
⟨*proof*⟩

lemma *curry-forall-eq*:
$(\forall f.\ P\ f) = (\forall f.\ P\ (case\text{-}prod\ f))$
⟨*proof*⟩

lemma *pres-tuple-vcg*:
$(\forall P.\ \{\!|P \circ (\lambda s.\ (f\ s,\ g\ s))|\!\}\ c)$
$\longleftrightarrow ((\forall P.\ \{\!|P \circ f|\!\}\ c) \wedge (\forall P.\ \{\!|P \circ g|\!\}\ c))$
⟨*proof*⟩

lemma *pres-tuple-vcg-LST*:
$(\forall P.\ \{\!|P \circ (\lambda s.\ (f\ s,\ g\ s)) \circ LST|\!\}\ c)$
$\longleftrightarrow ((\forall P.\ \{\!|P \circ f \circ LST|\!\}\ c) \wedge (\forall P.\ \{\!|P \circ g \circ LST|\!\}\ c))$
⟨*proof*⟩

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

end

⟨*ML*⟩

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**
  $com = \{\!|''A''|\!\}\ WHILE\ ((<)\ 0)\ DO\ \{\!|''B''|\!\}\ \lfloor \lambda s.\ s - 1 \rfloor\ OD$

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**
  $com = \{\!|''C''|\!\}\ WHILE\ ((<)\ 0)\ DO\ \{\!|''A''|\!\}\ \lfloor Suc \rfloor\ OD$

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

end

thm *locset-cache*

primrec *coms* :: $bool \Rightarrow$ (*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
  $rcv\text{-}syn :: 'location \Rightarrow 'channel \Rightarrow ('val \Rightarrow 'state \Rightarrow 'state)$
    $\Rightarrow (unit, 'location, 'channel \times 'val, 'state)\ com\ (\langle\{\!|\text{-}|\!\}/\ \text{-}\triangleright\text{-}\rangle\ [0,0,81]\ 81)$
where
  $\{\!|l|\!\}\ ch\triangleright f \equiv \{\!|l|\!\}\ Response\ (\lambda q\ s.\ if\ fst\ q = ch\ then\ \{(f\ (snd\ q)\ s,\ ())\}\ else\ \{\})$

abbreviation
$snd\text{-}syn :: {}'location \Rightarrow {}'channel \Rightarrow ({}'state \Rightarrow {}'val)$
$\quad\quad \Rightarrow (unit, {}'location, {}'channel \times {}'val, {}'state)\ com$ (‹⦃-⦄/ -◁-› [0,0,81] 81)
where
$⦃l⦄\ ch◁f \equiv ⦃l⦄\ Request\ (\lambda s.\ (ch, f\ s))\ (\lambda ans\ s.\ \{s\})$

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

datatype $ex\text{-}chname = \xi 12 \mid \xi 23$
type-synonym $ex\text{-}val = nat$
type-synonym $ex\text{-}ch = ex\text{-}chname \times ex\text{-}val$
datatype $ex\text{-}loc = r12 \mid r23 \mid s23 \mid s12$
datatype $ex\text{-}proc = p1 \mid p2 \mid p3$

type-synonym $ex\text{-}pgm = (unit, ex\text{-}loc, ex\text{-}ch, ex\text{-}val)\ com$
type-synonym $ex\text{-}pred = (unit, ex\text{-}loc, ex\text{-}proc, ex\text{-}ch, ex\text{-}val)\ state\text{-}pred$
type-synonym $ex\text{-}state = (unit, ex\text{-}loc, ex\text{-}proc, ex\text{-}ch, ex\text{-}val)\ system\text{-}state$
type-synonym $ex\text{-}sys = (unit, ex\text{-}loc, ex\text{-}proc, ex\text{-}ch, ex\text{-}val)\ system$
type-synonym $ex\text{-}history = (ex\text{-}ch \times unit)\ list$

We further specialise these for our particular example.

primrec
$ex\text{-}coms :: ex\text{-}proc \Rightarrow ex\text{-}pgm$
where
$ex\text{-}coms\ p1\ = ⦃s12⦄\ \xi 12◁id$
$\mid ex\text{-}coms\ p2\ = LOOP\ DO\ ⦃r12⦄\ \xi 12▷(\lambda v\ \text{-}.\ v)\ ;;\ ⦃s23⦄\ \xi 23◁id\ OD$
$\mid ex\text{-}coms\ p3\ = ⦃r23⦄\ \xi 23▷(\lambda v\ \text{-}.\ v)$

Each process starts with an arbitrary initial local state.

abbreviation $ex\text{-}init :: (ex\text{-}proc \Rightarrow ex\text{-}val) \Rightarrow bool$ **where**
$ex\text{-}init \equiv \langle True\rangle$

abbreviation $sys :: ex\text{-}sys$ **where**
$sys \equiv (\!|PGMs = ex\text{-}coms,\ INIT = ex\text{-}init,\ FAIR = \langle True\rangle|\!)$

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
$filter\text{-}on\text{-}channel :: ex\text{-}chname \Rightarrow ex\text{-}state \Rightarrow ex\text{-}val\ list$ (‹⌊-› [100] 101)
where
$\lfloor ch \equiv map\ (snd \circ fst) \circ filter\ ((=)\ ch \circ fst \circ fst) \circ HST$

definition $IL :: ex\text{-}pred$ **where**
$IL = pred\text{-}conjoin\ [$
$\quad at\ p1\ s12 \longrightarrow LIST\text{-}NULL\ \lfloor\xi 12$
$\quad ,\ terminated\ p1 \longrightarrow \lfloor\xi 12 = (\lambda s.\ [s\downarrow\ p1])$
$\quad ,\ at\ p2\ r12 \longrightarrow \lfloor\xi 12 = \lfloor\xi 23$
$\quad ,\ at\ p2\ s23 \longrightarrow \lfloor\xi 12 = \lfloor\xi 23\ @\ (\lambda s.\ [s\downarrow\ p2]) \wedge (\lambda s.\ s\downarrow\ p1\ = s\downarrow\ p2)$
$\quad ,\ at\ p3\ r23 \longrightarrow LIST\text{-}NULL\ \lfloor\xi 23$
$\quad ,\ terminated\ p3 \longrightarrow \lfloor\xi 23 = (\lambda s.\ [s\downarrow\ p2]) \wedge (\lambda s.\ s\downarrow\ p1\ = s\downarrow\ p3)$
$\quad]$

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 $Etern\text{-}pred :: ex\text{-}pred$ **where**
$Etern\text{-}pred = (terminated\ p3 \longrightarrow (\lambda s.\ s\downarrow\ p1\ = s\downarrow\ p3))$

Proofs from here down.

lemma *correct-system*:
  assumes $IL\ sh$
  shows $Etern\text{-}pred\ sh$
$\langle proof \rangle$

lemma *IL-p1*: $ex\text{-}coms,\ p1,\ lconst\ \{\} \vdash \{\!| IL |\!\}\ \{\!| s12 |\!\}\ \xi 12 \lhd (\lambda s.\ s)$
$\langle proof \rangle$

lemma *IL-p2*: $ex\text{-}coms,\ p2,\ lconst\ \{r12\} \vdash \{\!| IL |\!\}\ \{\!| s23 |\!\}\ \xi 23 \lhd (\lambda s.\ s)$
$\langle proof \rangle$

lemma *IL*: $sys \models_{pre} IL$
$\langle proof \rangle$

lemma *IL-valid*: $sys \models \Box\lceil IL \rceil$
$\langle proof \rangle$

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 $ex\text{-}chname = \xi 12 \mid \xi 23$
type-synonym $ex\text{-}val = nat$
type-synonym $ex\text{-}ls = ex\text{-}val\ list$
type-synonym $ex\text{-}ch = ex\text{-}chname \times ex\text{-}val$
datatype $ex\text{-}loc = c1 \mid r12 \mid r23 \mid s23 \mid s12$
datatype $ex\text{-}proc = p1 \mid p2 \mid p3$

type-synonym $ex\text{-}pgm = (unit,\ ex\text{-}loc,\ ex\text{-}ch,\ ex\text{-}ls)\ com$
type-synonym $ex\text{-}pred = (unit,\ ex\text{-}loc,\ ex\text{-}proc,\ ex\text{-}ch,\ ex\text{-}ls)\ state\text{-}pred$
type-synonym $ex\text{-}state = (unit,\ ex\text{-}loc,\ ex\text{-}proc,\ ex\text{-}ch,\ ex\text{-}ls)\ system\text{-}state$
type-synonym $ex\text{-}sys = (unit,\ ex\text{-}loc,\ ex\text{-}proc,\ ex\text{-}ch,\ ex\text{-}ls)\ system$
type-synonym $ex\text{-}history = (ex\text{-}ch \times unit)\ list$

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

abbreviation $(input)\ snoc :: 'a \Rightarrow 'a\ list \Rightarrow 'a\ list$ **where** $snoc\ x\ xs \equiv xs\ @\ [x]$

primrec $ex\text{-}coms :: ex\text{-}proc \Rightarrow ex\text{-}pgm$ **where**
  $ex\text{-}coms\ p1 = LOOP\ DO\ \{\!| c1 |\!\}\ LocalOp\ (\lambda xs.\ \{snoc\ x\ xs\ |x.\ True\})\ ;;\ \{\!| s12 |\!\}\ \xi 12 \lhd (last,\ id)\ OD$
$|\ ex\text{-}coms\ p2 = LOOP\ DO\ \{\!| r12 |\!\}\ \xi 12 \rhd snoc$
$\qquad \oplus\ \{\!| c1 |\!\}\ IF\ (\lambda s.\ length\ s > 0)\ THEN\ \{\!| s23 |\!\}\ \xi 12 \lhd (hd,\ tl)\ FI$
$\qquad OD$
$|\ ex\text{-}coms\ p3 = LOOP\ DO\ \{\!| r23 |\!\}\ \xi 23 \rhd snoc\ OD$

abbreviation $ex\text{-}init :: (ex\text{-}proc \Rightarrow ex\text{-}ls) \Rightarrow bool$ **where**
  $ex\text{-}init\ s \equiv \forall p.\ s\ p = []$

abbreviation $sys :: ex\text{-}sys$ **where**
  $sys \equiv (\!| PGMs = ex\text{-}coms,\ INIT = ex\text{-}init,\ FAIR = \langle True \rangle |\!)$

abbreviation
  $filter\text{-}on\text{-}channel :: ex\text{-}chname \Rightarrow ex\text{-}state \Rightarrow ex\text{-}val\ list\ (\langle\lfloor\text{-}\rangle\ [100]\ 101)$
where
  $\lfloor ch \equiv map\ (snd \circ fst) \circ filter\ ((=)\ ch \circ fst \circ fst) \circ HST$

definition $I\text{-}pred :: ex\text{-}pred$ **where**
  $I\text{-}pred = pred\text{-}conjoin\ [$

$at\ p1\ c1 \longrightarrow \lfloor\xi 12 = (\lambda s.\ s\!\downarrow p1)$
$, at\ p1\ s12 \longrightarrow (\lambda s.\ length\ (s\!\downarrow p1) > 0 \wedge butlast\ (s\!\downarrow p1) = (\lfloor\xi 12)\ s)$
$, \lfloor\xi 12 \leq (\lambda s.\ s\!\downarrow p1)$
$, \lfloor\xi 12 = \lfloor\xi 23\ @\ (\lambda s.\ s\!\downarrow p2)$
$, at\ p2\ s23 \longrightarrow (\lambda s.\ length\ (s\!\downarrow p2) > 0)$
$, (\lambda s.\ s\!\downarrow p3) = \lfloor\xi 23$
]

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

definition *Etern-pred* :: *ex-pred* **where**
$Etern\text{-}pred \equiv \lambda s.\ s\!\downarrow p3 \leq s\!\downarrow p1$

lemma *correct-system*:
assumes *I-pred s*
shows *Etern-pred s*
⟨*proof*⟩

lemma *p1-c1*[*simplified*, *intro*]:
$ex\text{-}coms,\ p1,\ lconst\ \{s12\} \vdash \{\!|I\text{-}pred|\!\}\ \{\!|c1|\!\}\ LocalOp\ (\lambda xs.\ \{\ snoc\ x\ xs\ |x.\ True\ \})$
⟨*proof*⟩

lemma *p1-s12*[*simplified*, *intro*]:
$ex\text{-}coms,\ p1,\ lconst\ \{c1\} \vdash \{\!|I\text{-}pred|\!\}\ \{\!|s12|\!\}\ \xi 12 \triangleleft (last,\ id)$
⟨*proof*⟩

lemma *p2-s23*[*simplified*, *intro*]:
$ex\text{-}coms,\ p2,\ lconst\ \{c1,\ r12\} \vdash \{\!|I\text{-}pred|\!\}\ \{\!|s23|\!\}\ \xi 12 \triangleleft (hd,\ tl)$
⟨*proof*⟩

lemma *p2-pi4*[*intro*]:
$ex\text{-}coms,\ p2,\ lcond\ \{s23\}\ \{c1,\ r12\}\ (\lambda s.\ s \neq []) \vdash \{\!|I\text{-}pred|\!\}\ \{\!|c1|\!\}\ IF\ (\lambda s.\ s \neq [])\ THEN\ c'\ FI$
⟨*proof*⟩

lemma *I*: $sys \models_{pre} I\text{-}pred$
⟨*proof*⟩

lemma *I-valid*: $sys \models \Box\lceil I\text{-}pred\rceil$
⟨*proof*⟩

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).

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

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.

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.