Trace Based Semantics for Rely-Guarantee

Marialena Hadjikosti, Andrei Popescu & Jamie Wright

August 26, 2026

Abstract

We formalize a trace-based Rely-Guarantee semantics based on the work of Xu et al. The foundation includes an abstract Sequential/While/Parallel programming language equipped with operational semantics designed for concurrent reasoning. Upon this, we build a reasoning infrastructure that includes Rely-Guarantee inversion rules, alongside abstract definitions and standard soundness proofs for both unary and binary settings.

Contents

1 Introduction

This development provides a formalization of a trace-based Rely-Guarantee semantics. The core approach is built upon the foundational work of Xu et

al. [2], who established a framework for reasoning about concurrent programs using interactive traces.

To support this semantics, we first define an abstract programming language featuring sequential composition, while loops, and parallel execution. This language is equipped with an operational semantics explicitly designed to facilitate concurrent reasoning by interleaving command executions with environment steps.

Building upon these trace-based foundations, we develop a robust reasoning infrastructure. This includes comprehensive Rely-Guarantee inversion rules that characterize the structural decomposition of traces, abstract definitions of trace satisfiability, and standard soundness proofs adapted for both unary and binary postcondition settings.

The formalization presented here serves as the foundational basis for the broader investigations into coinductive Rely-Guarantee semantics and equivalence proofs described by Derrick et al. [1].

2 Preliminaries

This theory sets up notation and preliminary definitions used throughout the mechanization

theory *Prelim* **imports** *Complex-Main*
begin

thm *le-fun-def*
thm *relcompp-apply*
thm *relcompp.simps*

hide-const *stable*

lemma *map-Pair-zip-replicate-length*:
$map\ (Pair\ x)\ ys = zip\ (replicate\ (length\ ys)\ x)\ ys$
⟨*proof*⟩

lemma *OO-rtranclp-le*:
assumes $U\ OO\ A \leq V\ (U\ OO\ Z\widehat{}**)\ OO\ (Z\ OO\ A) \leq V$
shows $(U\ OO\ Z\widehat{}**)\ OO\ A \leq V$
⟨*proof*⟩

lemma *rtranclp-chain*:
assumes $R^{**}\ a\ b$
shows $\exists n\ as.\ as\ 0 = a \wedge as\ n = b \wedge (\forall i{<}n.\ R\ (as\ i)\ (as\ (Suc\ i)))$
⟨*proof*⟩

lemma *chain-rtranclp*:
assumes $as\ 0 = a\ as\ n = b\ \forall i{<}n.\ R\ (as\ i)\ (as\ (Suc\ i))$
shows $R^{**}\ a\ b$
⟨*proof*⟩

definition $notP :: ('a \Rightarrow bool) \Rightarrow ('a \Rightarrow bool)$
$(\neg 1\ -\ [40]\ 70)$
where
$\neg 1\ P \equiv \lambda a.\ \neg\ P\ a$

definition $conjP :: ('a \Rightarrow bool) \Rightarrow ('a \Rightarrow bool) \Rightarrow ('a \Rightarrow bool)$
(**infixr** $\wedge 1\ 65$)
where
$P1\ \wedge 1\ P2 \equiv \lambda a.\ P1\ a \wedge P2\ a$

definition $disjP :: ('a \Rightarrow bool) \Rightarrow ('a \Rightarrow bool) \Rightarrow ('a \Rightarrow bool)$
(**infixr** $\vee 1\ 60$)
where
$P1\ \vee 1\ P2 \equiv \lambda a.\ P1\ a \vee P2\ a$

definition $notR :: ('a \Rightarrow 'b \Rightarrow bool) \Rightarrow ('a \Rightarrow 'b \Rightarrow bool)$
$(\neg 2\ -\ [40]\ 40)$
where
$\neg 2\ R \equiv \lambda a\ b.\ \neg\ R\ a\ b$

definition $conjR ::$
$('a \Rightarrow 'b \Rightarrow bool) \Rightarrow ('a \Rightarrow 'b \Rightarrow bool) \Rightarrow ('a \Rightarrow 'b \Rightarrow bool)$
(**infixr** $\wedge 2\ 65$)
where
$R1\ \wedge 2\ R2 \equiv \lambda a\ b.\ R1\ a\ b \wedge R2\ a\ b$

definition $disjR ::$
$('a \Rightarrow 'b \Rightarrow bool) \Rightarrow ('a \Rightarrow 'b \Rightarrow bool) \Rightarrow ('a \Rightarrow 'b \Rightarrow bool)$
(**infixr** $\vee 2\ 60$)
where
$R1\ \vee 2\ R2 \equiv \lambda a\ b.\ R1\ a\ b \vee R2\ a\ b$

definition $interR ::$

$('a \Rightarrow 'b \Rightarrow bool) \Rightarrow ('a \Rightarrow 'b \Rightarrow bool) \Rightarrow ('a \Rightarrow 'b \Rightarrow bool)$
(**infixr** $\cap 2$ *70*)
where
$R1 \cap 2\ R2 \equiv \lambda a\ b.\ \forall c\ d.\ R1\ c\ d \wedge R2\ c\ d \longrightarrow a = c \wedge b = d$

definition *satR* ::
$('a \Rightarrow 'b \Rightarrow bool)$
where
$satR \equiv \lambda a\ b.\ True$

definition *imR* :: $('a \Rightarrow 'b \Rightarrow bool) \Rightarrow ('a \Rightarrow bool) \Rightarrow ('b \Rightarrow bool)$ **where**
$imR\ R\ P \equiv \lambda b.\ \exists a.\ P\ a \wedge R\ a\ b$

lemmas *imR-defs* = *imR-def le-bool-def le-fun-def*

definition *rimR* :: $('a \Rightarrow 'b \Rightarrow bool) \Rightarrow ('b \Rightarrow bool) \Rightarrow ('a \Rightarrow bool)$ **where**
$rimR\ R\ P \equiv \lambda a.\ \exists b.\ P\ b \wedge R\ a\ b$

definition *grR* :: $('a \Rightarrow 'b) \Rightarrow 'a \Rightarrow 'b \Rightarrow bool$ **where**
$grR\ f \equiv (\lambda a\ b.\ f\ a = b)$

definition *diagR* :: $('a \Rightarrow bool) \Rightarrow 'a \Rightarrow 'a \Rightarrow bool$ **where**
$diagR\ P \equiv \lambda a\ b.\ a = b \wedge P\ a$

definition *lift1* :: $('a \Rightarrow bool) \Rightarrow 'a \Rightarrow 'b \Rightarrow bool$
where $lift1\ P \equiv \lambda a\ b.\ P\ a$

definition *lift2* :: $('b \Rightarrow bool) \Rightarrow 'a \Rightarrow 'b \Rightarrow bool$
where $lift2\ P \equiv \lambda a\ b.\ P\ b$

lemma *lift1-mono*: $P1 \leq P2 \Longrightarrow lift1\ P1 \leq lift1\ P2$
⟨*proof*⟩

lemma *lift2-mono*: $P1 \leq P2 \Longrightarrow lift2\ P1 \leq lift2\ P2$
⟨*proof*⟩

lemma *conjR-lift2-rel-1-OO*: $(R \wedge 2\ lift2\ P)\ OO\ Q = R\ OO\ (lift1\ P \wedge 2\ Q)$
⟨*proof*⟩

lemma *conjP-idem*[*simp*]: $P \wedge 1\ P = P$
⟨*proof*⟩

lemma *disjP-idem*[*simp*]: $P \vee 1\ P = P$
⟨*proof*⟩

lemma *conjR-idem*[*simp*]: $R \wedge 2\ R = R$

⟨*proof*⟩

lemma *disjR-idem*[*simp*]: $R \vee 2\ R = R$
⟨*proof*⟩

lemma *conjP-idem2*[*simp*]: $P \wedge 1\ P \wedge 1\ P' = P \wedge 1\ P'$
⟨*proof*⟩

lemma *notP-item*: $\neg\ P\ s \longleftrightarrow (\neg 1\ P)\ s$
⟨*proof*⟩

lemma *disjR-I1*[*intro*]:$P\ s\ s' \Longrightarrow (P \vee 2\ Q)\ s\ s'$ ⟨*proof*⟩
lemma *disjR-I2*[*intro*]:$Q\ s\ s' \Longrightarrow (P \vee 2\ Q)\ s\ s'$ ⟨*proof*⟩

lemma *conjR-I*:$Q1\ s\ s' \wedge\ Q2\ s\ s' \Longrightarrow Q1 \wedge 2\ Q2 \leq Q \Longrightarrow Q\ s\ s'$ ⟨*proof*⟩

lemma *lift1-inject*:$lift1\ P \leq Q \Longrightarrow P\ s \Longrightarrow \forall\, s'.\ Q\ s\ s'$ ⟨*proof*⟩

definition *stable* :: $('s \Rightarrow bool) \Rightarrow ('s \Rightarrow 's \Rightarrow bool) \Rightarrow bool$ **where**
$stable\ P\ R \equiv \forall\ s\ s'.\ P\ s \wedge R\ s\ s' \longrightarrow P\ s'$

definition *stableLeft* :: $('s \Rightarrow 's \Rightarrow bool) \Rightarrow ('s \Rightarrow 's \Rightarrow bool) \Rightarrow bool$ **where**
$stableLeft\ Q\ R \equiv R\ OO\ Q \leq Q$

definition *stableRight* :: $('s \Rightarrow 's \Rightarrow bool) \Rightarrow ('s \Rightarrow 's \Rightarrow bool) \Rightarrow bool$ **where**
$stableRight\ Q\ R \equiv Q\ OO\ R \leq Q$

definition *stable2* :: $('a \Rightarrow 'a \Rightarrow bool) \Rightarrow ('a \Rightarrow 'a \Rightarrow bool) \Rightarrow bool$ **where**
$stable2\ Q\ R \equiv stableLeft\ Q\ R \wedge stableRight\ Q\ R$

lemma *stable2-def2*: $stable2\ Q\ R \longleftrightarrow R\ OO\ Q \leq Q \wedge Q\ OO\ R \leq Q$
⟨*proof*⟩

lemma *stableLeft-alt*:
$stableLeft\ Q\ R \longleftrightarrow$
$(\forall\ s\ s'\ s''.\ R\ s\ s' \wedge\ Q\ s'\ s'' \longrightarrow Q\ s\ s'')$
⟨*proof*⟩

lemma *stableRight-alt*:
$stableRight\ Q\ R \longleftrightarrow$
$(\forall\ s\ s'\ s''.\ Q\ s\ s' \wedge\ R\ s'\ s'' \longrightarrow Q\ s\ s'')$
⟨*proof*⟩

lemmas *stable2-defs* = *stable2-def stableRight-alt stableLeft-alt*
lemma *stable-stable2*:
stable P R $\longleftrightarrow$ *stable2* (*lift2 P*) *R*
⟨*proof*⟩

lemma *stable-rtraclp*:
assumes *stable P R*
shows *stable P R***
⟨*proof*⟩

lemma *stableLeft-rtraclp*:
assumes *stableLeft Q R*
shows *stableLeft Q R***
⟨*proof*⟩

lemma *stableRight-rtraclp*:
assumes *stableRight Q R*
shows *stableRight Q R***
⟨*proof*⟩

lemma *stable2-rtraclp*:
assumes *stable2 Q R*
shows *stable2 Q R***
⟨*proof*⟩

lemma *stable-Eq*[*simp*,*intro*!]:
stable P (=)
⟨*proof*⟩

lemma *stable-OO*:
assumes *stable P R1 stable P R2*
shows *stable P* (*R1 OO R2*)
⟨*proof*⟩

lemma *stableLeft-Eq*[*simp*,*intro*!]:
stableLeft Q (=)
⟨*proof*⟩

lemma *stableLeft-OO*:
assumes *stableLeft Q R1 stableLeft Q R2*
shows *stableLeft Q* (*R1 OO R2*)
⟨*proof*⟩

lemma *stableRight-Eq*[*simp*,*intro*!]:
stableRight Q (=)
⟨*proof*⟩

lemma *stableRight-OO*:
assumes *stableRight Q R1 stableRight Q R2*
shows *stableRight Q* (*R1 OO R2*)
⟨*proof*⟩

lemma *stable2-Eq*[*simp*,*intro*!]:
stable2 Q (=)
⟨*proof*⟩

lemma *stable2-OO*:
assumes *stable2 Q R1 stable2 Q R2*
shows *stable2 Q* (*R1 OO R2*)
⟨*proof*⟩

lemma *stable-iff-imR*: *stable P R* ⟷ *imR R P* ≤ *P*
⟨*proof*⟩

lemma *stable-imR-rtraclp*: *stable P R* ⟹ *imR* (*R*^**) *P* ≤ *P*
⟨*proof*⟩

lemma *stable-lift1-lift2*: *stable P R* ⟹ *lift1 P* ∧2 *R* ≤ *R* ∧2 *lift2 P*
⟨*proof*⟩

lemma *stable-lift1-lift2-rtraclp*: *stable P R* ⟹ *lift1 P* ∧2 *R*^** ≤ *R*^** ∧2 *lift2 P*
⟨*proof*⟩

lemma *stableLeft-OO-leq*: *stableLeft Q R* ⟹ *Q1* ≤ *Q* ⟹ *R OO Q1* ≤ *Q*
⟨*proof*⟩

lemma *stable2-OO-leqL*: *stable2 Q R* ⟹ *Q1* ≤ *Q* ⟹ *R OO Q1* ≤ *Q*
⟨*proof*⟩

lemma *stableRight-OO-leq*: *stableRight Q R* ⟹ *Q1* ≤ *Q* ⟹ *Q1 OO R* ≤ *Q*
⟨*proof*⟩

lemma *stable2-OO-leqR*: *stable2 Q R* ⟹ *Q1* ≤ *Q* ⟹ *Q1 OO R* ≤ *Q*
⟨*proof*⟩

lemma *stable2-imp-OO*:
assumes *stable2 Q R1 stable2 Q R2*
shows *R1 OO Q OO R2* ≤ *Q*
⟨*proof*⟩

lemma *stable2-OO-leq*: *stable2 Q R1* ⟹ *stable2 Q R2* ⟹
Q' ≤ *Q* ⟹ *R1 OO Q' OO R2* ≤ *Q*
⟨*proof*⟩

lemma *stableLeft-lift2*[*simp*,*intro*!]: *stableLeft* (*lift2 P*) *R*
⟨*proof*⟩

lemma *stableRight-lift1*[*simp*,*intro*!]: *stableRight* (*lift1 P*) *R*
⟨*proof*⟩

lemma *rtranclp-least*: $(=) \leq Z \Longrightarrow Z\ OO\ A \leq Z \Longrightarrow A^{**} \leq Z$
⟨*proof*⟩

inductive
R-star **where**
refl[*simp*]: *R-star 0 R s s* |
stepRel: $R\ s\ s' \Longrightarrow$ *R-star n R s' s''* $\Longrightarrow$ *R-star* (*Suc n*) *R s s''*

lemma *R-star0*[*simp*]:*Ex* (*R-star 0 R sa*) ⟨*proof*⟩
lemma *R-star-refl*:*R-star 0 R s s'* $\Longrightarrow s = s'$ ⟨*proof*⟩
lemma *R-starRR*:*R-star n R s s'* $\Longrightarrow$ *R s' s''* $\Longrightarrow$ *R-star* (*Suc n*) *R s s''*
⟨*proof*⟩

lemma *R-star-imp*: *R-star n R s s'* $\Longrightarrow R^{**}\ s\ s'$
⟨*proof*⟩

lemma *R-step-star*:$R^{**}\ s\ s' = (\exists\, n.$ *R-star n R s s'*$)$
⟨*proof*⟩

lemma *Suc-assoc*:$(Suc\ (x + y)) = (Suc\ x + y)$ ⟨*proof*⟩

lemma *R-star-trans*:
assumes *R-star n R x y*
and *R-star m R y z*
shows *R-star* (*n*+*m*) *R x z*
⟨*proof*⟩

inductive
star :: $('a \Rightarrow 'a \Rightarrow bool) \Rightarrow 'a \Rightarrow 'a \Rightarrow bool$
for *r* **where**
refl: *star r x x* |
step: *r x y* $\Longrightarrow$ *star r y z* $\Longrightarrow$ *star r x z*

hide-fact (**open**) *refl step* — names too generic

lemma *star-trans*:
star r x y $\Longrightarrow$ *star r y z* $\Longrightarrow$ *star r x z*
⟨*proof*⟩

lemmas *star-induct* =
star.induct[*of r*:: *'a*∗*'b* ⇒ *'a*∗*'b* ⇒ *bool*, *split-format*(*complete*)]
lemmas *star-cases* =
star.cases[*of r*:: *'a*∗*'b* ⇒ *'a*∗*'b* ⇒ *bool*, *split-format*(*complete*)]

declare *star.refl*[*simp*,*intro*]

lemma *star-step1*[*simp*, *intro*]: *r x y* ⟹ *star r x y*
⟨*proof*⟩

definition *final r x* ≡ ∀ *y*. ¬ *r x y*

lemma *final-star-eq*: *final r x* ⟹ *star r x y* ⟹ *x* = *y*
⟨*proof*⟩
code-pred *star* ⟨*proof*⟩

lemma *add-less*:∀ *i'*. *i* ≠ (*x*::*nat*) + *i'* ⟹ *x* ≠ *0* ⟹ *i* < *x* ⟨*proof*⟩

lemma *Suc-red*:*Suc x* = *n* + *Suc m'* ⟷ *x* = *n* + *m' Suc x* = *Suc n* + *m'* ⟷
x = *n* + *m'* ⟨*proof*⟩

lemma *le-Suc-iff0*: ‹*m* ≤ *Suc n* ⟷ *m* = *0* ∨ (∃ *m'*. *m* = *Suc m'* ∧ *m'* ≤ *n*)›
⟨*proof*⟩

locale *Step* =
fixes
small-step :: *'com* × *'state* ⇒ *'com* × *'state* ⇒ *bool* (**infix** → *55*) **and**
final :: *'com* × *'state* ⇒ *bool*
begin

abbreviation
small-steps :: *'com* × *'state* ⇒ *'com* × *'state* ⇒ *bool* (**infix** →∗ *55*)
where *x* →∗ *y* == *star small-step x y*

inductive *step-rel* **where**
R-Step:*R s s'* ⟹ *step-rel R* (*c*, *s*) (*c*, *s'*)
|
S-Step:*small-step* (*c*, *s*) (*c'*, *s'*) ⟹ *step-rel R* (*c*, *s*) (*c'*, *s'*)

lemma *step-rel-cases-cfg* [*consumes 1*, *case-names R-Step S-Step*, *elim*]:
assumes *step-rel R* (*c*, *s*) *cfg'*
obtains (*R-Step*) *s'* **where** *cfg'* = (*c*, *s'*) **and** *R s s'*
| (*S-Step*) *c' s'* **where** *cfg'* = (*c'*, *s'*) **and** (*c*, *s*) → (*c'*, *s'*)
⟨*proof*⟩

inductive-cases *step-rel-inv-cases* [*elim*!]: *step-rel R* (*c*, *s*) (*c′*, *s′*)

lemma *step-rel-strengthenR*:*step-rel R* (*c*, *s*) (*c′*, *s′*) $\Longrightarrow$ *R* $\leq$ *R′* $\Longrightarrow$ *step-rel R′* (*c*, *s*) (*c′*, *s′*)
⟨*proof*⟩

lemma *step-rel-cases*: *step-rel R* (*c*, *s*) (*c′*, *s′*) $\Longrightarrow$ (*c* = *c′* $\Longrightarrow$ *R s s′* $\Longrightarrow$ *P*) $\Longrightarrow$ ((*c*, *s*) $\rightarrow$ (*c′*, *s′*) $\Longrightarrow$ *P*) $\Longrightarrow$ *P*
⟨*proof*⟩

definition *lift R* $\equiv$ (λ(*c*,*s*) (*c′*,*s′*). *c′* = *c* $\wedge$ *R s s′*)

lemma *rtranclp-lift-extract*:
assumes (*lift R*)** (*c*, *s*) (*c′*, *s′*)
shows *c* = *c′* $\wedge$ *R*** *s s′*
⟨*proof*⟩

definition *fstep-rel R* $\equiv$ *rtranclp* (*lift R*) *OO small-step* — OO rtranclp (lift R)

lemma *rel-incl-lift-rtranclp*: *R*** *s s″* $\Longrightarrow$ (*lift R*)** (*c*, *s*) (*c*, *s″*)
⟨*proof*⟩

lemma *rtranclp-small-step-fstep-rel*:
assumes *R*: *R*** *s s″* **and** *st*: (*c*, *s″*) $\rightarrow$ (*c1*, *s1*)
shows *fstep-rel R* (*c*, *s*) (*c1*, *s1*)
⟨*proof*⟩

lemma *lift-fst*: *lift R cfg cfg′* $\Longrightarrow$ *fst cfg′* = *fst cfg*
⟨*proof*⟩

lemma *rtranclp-lift-fst*: *rtranclp* (*lift R*) *cfg cfg′* $\Longrightarrow$ *fst cfg′* = *fst cfg*
⟨*proof*⟩
end

lemma *whileAssms-inject*:
assumes *st*: *stable P R stable2 Q R stable2 Pt1 R*
and *Pr1*: (*evalT t*) $\wedge$*1 P* $\leq$ *Pr1*
and *P*: *imR Pt1* ((*evalT t*) $\wedge$*1 Pr1*) $\leq$ *P*
and *Q*: *lift1 P* $\wedge$*2* (*lift1* (*evalT t* $\wedge$*1 P*) $\wedge$*2 Pt1*)^** $\wedge$*2 lift2* ($\neg$*1 evalT t*) $\leq$ *Q*
shows
imR (*R*** $\wedge$*2 lift2* (*evalT t*)) *P* $\leq$ *Pr1*

imR Pt1 (*imR* R^{**} (*imR* R^{**} *P* $\wedge 1$ *evalT t*) $\wedge 1$ *Pr1*) $\leq$ *P*
(*lift1 P* $\wedge 2$ ((((*lift1 P* $\wedge 2$ R^{**}) $\wedge 2$ *lift2* (*evalT t*)) *OO* R^{**}) *OO Pt1*)**) *OO* (R^{**} $\wedge 2$ *lift2* ($\neg 1$ *evalT t*)) *OO* R^{**} $\leq$ *Q*
⟨*proof*⟩

inductive *terminFrom* **for** *R* **where**
($\bigwedge$*cfg'*. *R cfg cfg'* $\Longrightarrow$ *terminFrom R cfg'*) $\Longrightarrow$ *terminFrom R cfg*

lemma *terminFrom-induct-split* [*consumes 1*, *case-names Step*]:
assumes *terminFrom R* (*c*, *s*)
assumes $\bigwedge$*c s*. ($\bigwedge$*c' s'*. *R* (*c*, *s*) (*c'*, *s'*) $\Longrightarrow$ *terminFrom R* (*c'*, *s'*) $\wedge$ *P c' s'*) $\Longrightarrow$ *P c s*
shows *P c s*
⟨*proof*⟩

context *Step*
begin

lemma *terminFrom-env-step*:
assumes *terminFrom* (*fstep-rel R*) (*c*, *s*)
assumes $R\widehat{}**$ *s s'*
shows *terminFrom* (*fstep-rel R*) (*c*, *s'*)
⟨*proof*⟩

lemma *step-rel-refl*:
assumes *R s s*
shows *step-rel R* (*c*, *s*) (*c*, *s*)
⟨*proof*⟩

lemma *lift-step-rel*:
assumes *lift R x y*
shows *step-rel R x y*
⟨*proof*⟩

lemma *step-rel-not-small-lift*:
assumes *step-rel R x y* **and** ¬ *small-step x y*
shows *lift R x y*
⟨*proof*⟩

lemma *fstep-has-fseq*:

assumes *fstep-rel R x y*
shows $\exists N\, f.\ f\ 0 = x \wedge f\ N = y \wedge N > 0 \wedge$
$(\forall i{<}N.\ \textit{step-rel}\ R\ (f\ i)\ (f\ (\textit{Suc}\ i))) \wedge$
$(\textit{small-step}\ (f\ (N - 1))\ (f\ N))$
⟨*proof*⟩

fun *ch-step* :: $(nat \Rightarrow nat) \Rightarrow nat \Rightarrow nat \times nat$ **where**
ch-step N 0 = (*0*,*0*)
| *ch-step N* (*Suc k*) =
(*let* (*i*,*j*) = *ch-step N k in*
if Suc j < *N i then* (*i*, *Suc j*)
else (*Suc i*, *0*))

fun *start-idx* :: $(nat \Rightarrow nat) \Rightarrow nat \Rightarrow nat$ **where**
start-idx N 0 = *0* |
start-idx N (*Suc i*) = *start-idx N i* + *N i*

fun *forwardSS* :: $(nat \Rightarrow nat) \Rightarrow nat \Rightarrow nat$ **where**
forwardSS next-ss 0 = *0* |
forwardSS next-ss (*Suc n*) = *Suc* (*next-ss* (*forwardSS next-ss n*))

lemma *fstep-rel-iff-fair-step-rel*:
$(\exists\, ch.\ ch\ 0 = cfg \wedge (\forall i.\ \textit{fstep-rel}\ R\ (ch\ i)\ (ch\ (\textit{Suc}\ i))))$
$\longleftrightarrow$
$(\exists\, ch'.\ ch'\ 0 = cfg \wedge (\forall i.\ \textit{step-rel}\ R\ (ch'\ i)\ (ch'\ (\textit{Suc}\ i))) \wedge (\forall i.\ \exists j \geq i.\ \textit{small-step}\ (ch'\ j)\ (ch'\ (\textit{Suc}\ j))))$
⟨*proof*⟩

end

definition *stableWithDescent R P ord* $\equiv \forall n\ s\ s'.\ P\ n\ s \wedge R\ s\ s' \longrightarrow (\exists\, m.\ (m,n) \in ord \wedge P\ m\ s')$

lemma *rtranclp-Id*: $(=)^{**} = (=)$
⟨*proof*⟩
end

3 Basic Language Definition

This theory formalises a small-step semantics for a basic while language with parallel composition

theory *Sequential-Par-While-Language*
imports *Prelim*

begin

datatype (*'atom*,*'test*)*com* =
  Done
 |*Atom 'atom*
 |*Seq* (*leftSeq*:(*'atom*,*'test*)*com*) (*'atom*,*'test*)*com* (- $$ - [*61*, *60*] *60*)
 |*If 'test* (*'atom*,*'test*)*com* (*'atom*,*'test*)*com* ((*if* (-)/ {-}/ *else*/ {-}) [*0*, *0*, *61*] *61*)
 |*While 'test* (*'atom*,*'test*)*com* ((*while* (-)/ {-}) [*0*, *61*] *61*)
 |*Par* (*'atom*,*'test*)*com* (*'atom*,*'test*)*com* (- || - [*61*, *60*] *60*)

locale *SeqParWhileLang* =
fixes *evalA* :: *'atom* ⇒ *'state* ⇒ *'state*
and *evalT* :: *'test* ⇒ *'state* ⇒ *bool*
and *Skip* :: *'atom* **and** *Not* :: *'test* ⇒ *'test*
assumes *evalA-Skip-id*[*simp*,*intro*!]: *evalA Skip* = *id*
and *evalT-Not*[*simp*]: ⋀*s*. *evalT* (*Not t*) *s* = (¬ *evalT t s*)
begin

lemma *evalA-Skip*[*simp*,*intro*!]: *evalA Skip s* = *s*
⟨*proof*⟩

definition *Await* :: *'test* ⇒ (*'atom*,*'test*)*com* **where**
Await t = *While* (*Not t*) (*Atom Skip*)

3.1 Small-step semantics

inductive *small-step* :: (*'atom*,*'test*)*com* × *'state* ⇒ (*'atom*,*'test*)*com* × *'state* ⇒ *bool* (**infix** → *55*) **where**

Atom: *small-step* (*Atom a*, *s*) (*Done*, *evalA a s*)
|
Seq-Done: *small-step* (*Seq Done c2*, *s*) (*c2*,*s*)
|
Seq: *c1* ≠ *Done* ⟹ *small-step* (*c1*,*s*) (*c1'*,*s'*) ⟹ *small-step* (*Seq c1 c2*, *s*) (*Seq c1' c2*, *s'*)
|
If: *evalT t s* ⟹ *small-step* (*If t c1 c2*, *s*) (*c1*,*s*)
|
If-not: ¬ *evalT t s* ⟹ *small-step* (*If t c1 c2*, *s*) (*c2*,*s*)
|
While: *small-step* (*While t c*, *s*) (*If t* (*Seq c* (*While t c*)) *Done*, *s*)
|
Par-Done: *small-step* (*Par Done Done*, *s*) (*Done*, *s*)
|
Par1: ¬ (*c1* = *Done* ∧ *c2* = *Done*) ⟹ *small-step* (*c1*, *s*) (*c1'*, *s'*) ⟹ *small-step* (*Par c1 c2*, *s*) (*Par c1' c2*, *s'*)

|
Par2: $\neg$ (*c1* = *Done* $\wedge$ *c2* = *Done*) $\Longrightarrow$ *small-step* (*c2*, *s*) (*c2'*, *s'*) $\Longrightarrow$ *small-step* (*Par c1 c2*, *s*) (*Par c1 c2'*, *s'*)

abbreviation
small-step-star :: ('*atom*,'*test*)*com* $\times$ '*state* $\Rightarrow$ ('*atom*,'*test*)*com* $\times$ '*state* $\Rightarrow$ *bool* (**infix** $\rightarrow*$ *55*)
where *x* $\rightarrow*$ *y* == *star small-step x y*

lemma *step-Atom*[*simp*]: *small-step* (*Atom a*, *s*) *cf'* $\longleftrightarrow$ *cf'* = (*Done*,*evalA a s*)
⟨*proof*⟩

lemma *sstep-Atom*[*simp*]: (*Atom a*, *s*) $\rightarrow*$ (*Done*, *evalA a s*) ⟨*proof*⟩

lemma *step-Seq-Done*[*simp*]: *small-step* (*Seq Done c2*, *s*) *cf'* $\longleftrightarrow$ *cf'* = (*c2*,*s*)
⟨*proof*⟩

lemma *step-Seq-Done1*[*simp*]: *small-step* (*Seq Done c2*, *s*) (*c'*,*s'*) $\longleftrightarrow$ (*c'*,*s'*) = (*c2*,*s*)
⟨*proof*⟩

lemma *step-Seq-notDone*[*simp*]:
c1 $\neq$ *Done* $\Longrightarrow$ *small-step* (*Seq c1 c2*, *s*) *cf'* $\longleftrightarrow$ ($\exists$ *c1' s'*. *small-step* (*c1*,*s*) (*c1'*,*s'*) $\wedge$ *cf'* = (*Seq c1' c2*,*s'*))
⟨*proof*⟩

lemma *step-Seq-notDonePair*:
c1 $\neq$ *Done* $\Longrightarrow$ *small-step* (*Seq c1 c2*, *s*) (*c'*, *s'*) $\longleftrightarrow$ ($\exists$ *c1' s''*. *small-step* (*c1*,*s*) (*c1'*,*s''*) $\wedge$ (*c'*, *s'*) = (*Seq c1' c2*,*s''*))
⟨*proof*⟩

lemma *step-Seq-If*[*simp*]:
evalT t s $\Longrightarrow$ *small-step* (*If t c1 c2*, *s*) *cf'* $\longleftrightarrow$ *cf'* = (*c1*,*s*)
⟨*proof*⟩

lemma *step-Seq-If-not*[*simp*]:
$\neg$ *evalT t s* $\Longrightarrow$ *small-step* (*If t c1 c2*, *s*) *cf'* $\longleftrightarrow$ *cf'* = (*c2*,*s*)
⟨*proof*⟩

lemma *step-Seq-If-s*:
small-step (*If t c1 c2*, *s*) (*c*,*s'*) $\Longrightarrow$ *s* = *s'*
⟨*proof*⟩

lemma *step-While*[*simp*]:
small-step (*While t c*, *s*) *cf'* $\longleftrightarrow$ *cf'* = (*If t* (*Seq c* (*While t c*)) *Done*, *s*)

⟨*proof*⟩

lemma *step-Par-Done*[*simp*]:
small-step (*Done* || *Done*, *s*) *cf'* ⟷ *cf'* = (*Done*, *s*)
⟨*proof*⟩

lemma *step-Par*[*simp*]:
¬ (*c1* = *Done* ∧ *c2* = *Done*) ⟹ *small-step* (*c1* || *c2*, *s*) *cf'* ⟷
(∃ *c1' s'* . *small-step* (*c1*, *s*) (*c1'*, *s'*) ∧ *cf'* = (*c1'* || *c2*, *s'*)) ∨
(∃ *c2' s'* . *small-step* (*c2*, *s*) (*c2'*, *s'*) ∧ *cf'* = (*c1* || *c2'*, *s'*))
⟨*proof*⟩

definition *final* **where**
final cf ≡ ∀ *cf'*. ¬ *small-step cf cf'*

lemma *step-iff-not-Done*: (∃ *cf'*. *small-step cf cf'*) ⟷ *fst cf* ≠ *Done*
⟨*proof*⟩

lemma *final-iff-Done*: *final cf* ⟷ *fst cf* = *Done*
⟨*proof*⟩

lemma *step-Done-simp* [*simp*]: ((*Done*, *s*) → (*c'*, *s'*)) ⟷ *False* ⟨*proof*⟩

lemma *step-Done-elim* [*elim*!]:
assumes (*Done*, *s*) → (*c'*, *s'*)
shows *P*
⟨*proof*⟩

lemma *final-iff-Done2*: *final* (*c*,*s*) ⟷ *c* = *Done* ⟨*proof*⟩

lemma *final-Done*[*simp*]:*final* (*Done*, *s*) ⟨*proof*⟩

lemma *par-cases*: (*c1* || *c2*, *s*) → *cf* ⟹
(*c1* = *Done* ∧ *c2* = *Done* ∧ *cf* = (*Done*, *s*))∨ (∃ *c' s'*. (*c1*, *s*) → (*c'*, *s'*) ∧ *cf*
= (*c'* || *c2*, *s'*))
∨ (∃ *c' s'*. (*c2*, *s*) → (*c'*, *s'*) ∧ *cf* = (*c1* || *c'*, *s'*))
⟨*proof*⟩

lemma *par-cases'*[*case-names pdone step1 step2*]:
assumes (*Par c1 c2*, *s*) → *cf*
obtains
(*pdone*) *c1* = *Done c2* = *Done cf* = (*Done*,*s*)
| (*step1*) *c' s'* **where** (*c1*,*s*) → (*c'*,*s'*) *cf* = (*c'* || *c2*,*s'*)
| (*step2*) *c' s'* **where** (*c2*,*s*) → (*c'*,*s'*) *cf* = (*c1* || *c'*,*s'*)
⟨*proof*⟩

inductive *is-seq-com* :: (*'atom*,*'test*)*com* ⇒ *bool* **where**
Done-seq: *is-seq-com Done*
|
Atom-seq: *is-seq-com* (*Atom a*)
|
Seq-seq: *is-seq-com c1* ⟹ *is-seq-com c2* ⟹ *is-seq-com* (*Seq c1 c2*)
|
If-seq: *is-seq-com c1* ⟹ *is-seq-com c2* ⟹ *is-seq-com* (*If t c1 c2*)
|
While-seq: *is-seq-com c* ⟹ *is-seq-com* (*While t c*)
|
Par-done-seq: *c1* = *Done* ⟹ *c2* = *Done* ⟹ *is-seq-com* (*c1* || *c2*)

lemma *is-seq-com-Seq*[*simp*]:
is-seq-com (*Seq c1 c2*) ⟷ *is-seq-com c1* ∧ *is-seq-com c2*
⟨*proof*⟩

lemma *is-seq-com-if*[*simp*]:
is-seq-com (*If t c1 c2*) ⟷ *is-seq-com c1* ∧ *is-seq-com c2*
⟨*proof*⟩

lemma *is-seq-com-while*[*simp*]:
is-seq-com (*While t c*) ⟷ *is-seq-com c*
⟨*proof*⟩

lemma *par-not-seq*: *c1* ≠ *Done* ∨ *c2* ≠ *Done* ⟹ ¬ *is-seq-com* (*c1* || *c2*)
⟨*proof*⟩

definition *is-seq-cfg* **where**
is-seq-cfg cf ⟷ (∃ *c s* . *cf* = (*c*, *s*) ∧ *is-seq-com c*)

lemma *is-seq-cfg-split*[*simp*]:
is-seq-cfg (*c*, *s*) ⟷ *is-seq-com c*
⟨*proof*⟩

lemma *is-seq-com-step*:
is-seq-com c ⟹ (*c*, *s*) → (*c'*, *s'*) ⟹ *is-seq-com c'*
⟨*proof*⟩

lemma *step-determ-seq*:
assumes *is-seq-cfg cf small-step cf cf'* *small-step cf cf''*
shows *cf'* = *cf''*
⟨*proof*⟩

end

sublocale *SeqParWhileLang* < *Step* **where**
small-step = *small-step* **and** *final* = *final* ⟨*proof*⟩

end

4 Trace Semantics and Annotated Configurations

This theory defines annotated configurations and the formal definition of valid execution traces, interleaving small-step command executions with environment steps.

theory *RG-Semantics*
imports *../Sequential-Par-While-Language*
begin

datatype (*'com*,*'state*) *acfg* =
  isS: *S 'com* × *'state* |
  isE: *E 'com* × *'state*

fun *cfgOf* **where**
 cfgOf (*S cf*) = *cf*
|*cfgOf* (*E cf*) = *cf*

fun *updCfg* **where**
 updCfg (*S cf*) *cf'* = *S cf'*
|*updCfg* (*E cf*) *cf'* = *E cf'*

lemma *cfgOf-updCfg*[*simp*]: *cfgOf* (*updCfg acfg cf*) = *cf*
⟨*proof*⟩

lemma *updCfg2*[*simp*]: *updCfg* (*updCfg acfg cf*) *cf'* = *updCfg acfg cf'*
⟨*proof*⟩

lemma *updCfg-eq-cfgOf*[*simp*]: *updCfg acfg cf* = *acfg* ⟷ *cfgOf acfg* = *cf*
⟨*proof*⟩

lemma *updCfg-eq-cfgOf2*[*simp*]: *acfg* = *updCfg acfg cf* ⟷ *cfgOf acfg* = *cf*
 ⟨*proof*⟩

lemma *cfgOf-isS*:*cfgOf x* = (*c*, *s*) ⟹ *isS x* ⟹ *x* = *S* (*c*, *s*) ⟨*proof*⟩
lemma *cfgOf-isE*:*cfgOf x* = (*c*, *s*) ⟹ *isE x* ⟹ *x* = *E* (*c*, *s*) ⟨*proof*⟩

context *Step*
begin

definition *trace* :: (*'com*, *'state*) *acfg list* ⇒ *bool* **where**
trace tr ≡

$(\forall\, i\ c\ s\ c'\ s'.\ Suc\ i < length\ tr \wedge (c,s) = cfgOf\ (tr!i) \wedge (c',s') = cfgOf\ (tr!(Suc\ i))$
$\longrightarrow (small\text{-}step\ (c,s)\ (c',s') \wedge tr!(Suc\ i) = S\ (c',s')) \vee$
$(c' = c \wedge tr!(Suc\ i) = E\ (c',s')))$

lemma *trace-take*: $trace\ tr \Longrightarrow 0 < i \Longrightarrow trace\ (take\ i\ tr)$
⟨*proof*⟩

lemma *trace-Nil*[*simp*,*intro*!]: $trace\ []$
⟨*proof*⟩

lemma *trace-singl*[*simp*,*intro*!]: $trace\ [acfg]$
⟨*proof*⟩

lemma *trace-cases*:$trace\ (a\ \#\ tr) \Longrightarrow$
$Suc\ i < length\ (a\ \#\ tr) \wedge (c,\ s) = cfgOf\ ((a\ \#\ tr)\ !\ i) \wedge (c',\ s') = cfgOf\ ((a\ \#\ tr)\ !\ Suc\ i) \Longrightarrow$
$(c,\ s) \rightarrow (c',\ s') \wedge (a\ \#\ tr)\ !\ Suc\ i = S\ (c',\ s') \vee c' = c \wedge (a\ \#\ tr)\ !\ Suc\ i = E\ (c',\ s')$
⟨*proof*⟩

lemma *trace-append*:
assumes *tr*: $trace\ tr$ **and** *str*: $trace\ tr'$ **and**
juncture: $\bigwedge c\ s\ c'\ s'.$
$(c,s) = cfgOf\ (last\ tr) \Longrightarrow (c',s') = cfgOf\ (hd\ tr')$
$\Longrightarrow (small\text{-}step\ (c,s)\ (c',s') \wedge hd\ tr' = S\ (c',s')) \vee$
$(c' = c \wedge hd\ tr' = E\ (c',s'))$
shows $trace\ (tr\ @\ tr')$
⟨*proof*⟩

lemma *trace-Cons*:
assumes *str*: $trace\ tr$ **and**
juncture: $\bigwedge c\ s\ c'\ s'.$
$(c,s) = cfgOf\ acfg \Longrightarrow (c',s') = cfgOf\ (hd\ tr)$
$\Longrightarrow (small\text{-}step\ (c,s)\ (c',s') \wedge hd\ tr = S\ (c',s')) \vee$
$(c' = c \wedge hd\ tr = E\ (c',s'))$
shows $trace\ (acfg\ \#\ tr)$
⟨*proof*⟩

lemma *tl-trace*:
$trace\ tr \Longrightarrow trace\ (tl\ tr)$
⟨*proof*⟩

lemma *trace-ConsL*: $trace\ (tr\ @\ tr') \Longrightarrow trace\ tr$
⟨*proof*⟩

lemma *trace-ConsR*: $trace\ (tr\ @\ tr') \Longrightarrow trace\ tr'$

⟨*proof*⟩

lemma *trace-Cons-isS*:
assumes *ptr*: *trace* (*acfg* # *tr*) **and** *tr*: *tr* ≠ [] *isS* (*hd tr*)
shows *small-step* (*cfgOf acfg*) (*cfgOf* (*hd tr*))
⟨*proof*⟩

lemma *trace-Cons-isE*:
assumes *ptr*: *trace* (*acfg* # *tr*) **and** *tr*: *tr* ≠ [] *isE* (*hd tr*)
shows *fst* (*cfgOf acfg*) = *fst* (*cfgOf* (*hd tr*))
⟨*proof*⟩

lemma *step-traceE*[*simp*]:*trace* [*E* (*c*, *s*), *E* (*c*, *s*′)] ⟨*proof*⟩

lemma *step-traceS*[*simp*]:(*c*, *s*) → (*c*′, *s*′) ⟹ *trace* [*S* (*c*, *s*), *S* (*c*′, *s*′)] ⟨*proof*⟩

lemma *hd-tr-isE*:*tr*≠[] ⟹ *isE* (*tr* ! *0*) ⟹ *cfgOf* (*hd tr*) = (*c*,*s*) ⟹ *hd tr* = *E* (*c*, *s*)
⟨*proof*⟩

lemma *hd-tr-isS*:*tr*≠[] ⟹ *isS* (*tr* ! *0*) ⟹ *cfgOf* (*hd tr*) = (*c*,*s*) ⟹ *hd tr* = *S* (*c*, *s*)
⟨*proof*⟩

lemma *hd-trE*:*hd tr* = *E a* ⟹ *hd tr* = *E* (*cfgOf* (*hd tr*)) ⟨*proof*⟩
lemma *hd-trS*:*hd tr* = *E a* ⟹ *hd tr* ≠ *S* (*cfgOf* (*hd tr*)) ⟨*proof*⟩

lemma *cfgOf-hd*:*tr*≠[] ⟹ *cfgOf* (*hd tr*) = *cfgOf* (*tr*!*0*) ⟨*proof*⟩
lemma *last-cfgOf*:*tr*≠[] ⟹ *tl tr* = [] ⟹ *cfgOf* (*tr* ! *0*) = *cfgOf* (*last tr*) ⟨*proof*⟩
lemma *last-cfgOf-hd*:*tr*≠[] ⟹ *tl tr* = [] ⟹ *cfgOf* (*hd tr*) = *cfgOf* (*last tr*) ⟨*proof*⟩

lemma *cfgOf-lastE2*:*cfgOf* (*last* [*E* (*c*, *s*), *E* (*c*′, *s*′)]) = (*c*′, *s*′) ⟨*proof*⟩
lemma *cfgOf-lastS2*:*cfgOf* (*last* [*S* (*c*, *s*), *S* (*c*′, *s*′)]) = (*c*′, *s*′) ⟨*proof*⟩

lemma *hd-tl-eq*: *tl tr* = *x* # *xs* ⟹ *tr* ! *Suc 0* = *hd* (*tl tr*)
⟨*proof*⟩

lemma *tl-eq-nth*:*tl tr* = *x* # *xs* ⟹ (*tl tr* ! *i*) = (*tr* ! *Suc i*)
⟨*proof*⟩

lemma *tl-ne-imp-length-tr*:*tl tr* = *x* # *xs* ⟹ *Suc 0* < *length tr*
⟨*proof*⟩

lemma *trace-append-hdS*:
assumes *step*: $(c,s) \to (c',s')$ **and** *tr*: *trace tr* **and**
tr-hd: *cfgOf* (*hd tr*) = (c',s')
shows *trace* ([*S* (*c*, *s*), *S* (c', s')] @ *tl tr*)
⟨*proof*⟩

lemma *trace-append-hdS-cons*:
assumes *step*: $(c,s) \to (c',s')$ **and** *tr*: *trace tr* **and**
tr-hd: *cfgOf* (*hd tr*) = (c',s')
shows *trace* (*S* (*c*, *s*) # *S* (c', s') # *tl tr*)
⟨*proof*⟩

lemma *trace-append-hdE*:
assumes *tr*: *trace tr* **and**
tr-hd: (c,s') = *cfgOf* (*hd tr*)
shows *trace* ([*E* (*c*, *s*), *E* (*c*, s')] @ *tl tr*)
⟨*proof*⟩

lemma *trace-append-hdE-cons*:
assumes *tr*: *trace tr* **and**
tr-hd: *cfgOf* (*hd tr*) = (c,s')
shows *trace* (*E* (*c*, *s*) # *E* (*c*, s') # *tl tr*)
⟨*proof*⟩

lemma *trace-hdE*[*simp*]:*trace tr* ⟹ *cfgOf* (*hd tr*) = (*c*, σ) ⟹ *trace* (*E* (*c*, σ) # *tl tr*) ⟨*proof*⟩
lemma *trace-hdS*[*simp*]:*trace tr* ⟹ *cfgOf* (*hd tr*) = (*c*, σ) ⟹ *trace* (*S* (*c*, σ) # *tl tr*) ⟨*proof*⟩

lemma *trace-step-or-env*:
assumes *tr*: *trace tr* **and** *i*: *Suc i* < *length tr*
shows *small-step* (*cfgOf* (*tr*!*i*)) (*cfgOf* (*tr*!(*Suc i*))) ∧ *isS* (*tr*!(*Suc i*)) ∨
(*fst* (*cfgOf* (*tr*!(*Suc i*))) = *fst* (*cfgOf* (*tr*!*i*)) ∧ *isE* (*tr* ! *Suc i*))
⟨*proof*⟩

lemma *trace-hd-step-or-env*:
assumes *trace* (*t* # *tr*) **and** *tr* ≠ []
shows (*cfgOf t* → *cfgOf* (*hd* (*tr*)) ∧ *isS* (*hd tr*)) ∨ (*fst* (*cfgOf t*) = *fst* (*cfgOf* (*hd tr*)) ∧ *isE* (*hd tr*))
⟨*proof*⟩

lemma *isE-all*:*tr* = *map* (λ*s*. *E* (*Done*, *s*)) *sl* ⟹ *i* < *length tr* ⟹ *isE* (*tr* ! *i*)
⟨*proof*⟩

lemma *isE-allSuc*:*tr* = *hd tr* # *map* (λ*s*. *E* (*Done*, *s*)) *sl* ⟹ *Suc i* < *length tr* ⟹ *isE* (*tr* ! *Suc i*)
⟨*proof*⟩

end

end

5 Trace Inversion Rules

This theory mechanises the inversion lemmas for our semantics, characterizing the structural decomposition of traces for each command in the language.

theory *RG-Inversion-Rules*
imports *RG-Semantics*
begin

context *SeqParWhileLang*
begin

lemma *isE-all*:$tr = map\ (\lambda s.\ E\ (Done,\ s))\ sl \Longrightarrow i < length\ tr \Longrightarrow isE\ (tr\ !\ i)$
⟨*proof*⟩

lemma *isE-allSuc*:$tr = hd\ tr\ \#\ map\ (\lambda s.\ E\ (Done,\ s))\ sl \Longrightarrow Suc\ i < length\ tr$
$\Longrightarrow isE\ (tr\ !\ Suc\ i)$
⟨*proof*⟩

lemma *trace-Done-Suc*:
assumes *tr*: *trace tr* **and** *i*: *Suc i* < *length tr* **and** *f*: *fst* (*cfgOf* (*tr*!*i*)) = *Done*
shows *fst* (*cfgOf* (*tr*!(*Suc i*))) = *Done* ∧ *isE* (*tr* ! *Suc i*)
⟨*proof*⟩

lemma *trace-Done′*:
assumes *tr*: *trace tr* **and** *f*: *fst* (*cfgOf* (*tr*!*i*)) = *Done*
and *j*: *i* < *j j* < *length tr*
shows *fst* (*cfgOf* (*tr*!*j*)) = *Done* ∧ *isE* (*tr* ! *j*)
⟨*proof*⟩

lemma *trace-Done-tl*:
assumes *tr*: *trace tr* **and** *ntr*: *tr* ≠ [] **and** *f*: *fst* (*cfgOf* (*hd tr*)) = *Done*
shows ∃ *sl*. *tl tr* = *map* (λ*s*. *E* (*Done*,*s*)) *sl*
⟨*proof*⟩

lemma *trace-Done*:
assumes *tr*: *trace tr tr* ≠ [] **and** *f*: *fst* (*cfgOf* (*hd tr*)) = *Done*
shows ∃ *sl*. *tr* = (*hd tr*) # *map* (λ*s*. *E* (*Done*,*s*)) *sl*
⟨*proof*⟩

lemma *trace-Done-last*: **assumes** *trace tr* **and** *tr* ≠ [] **and** *f*: *fst* (*cfgOf* (*hd* (*tr*)))
= *Done*

shows *fst* (*cfgOf* (*last tr*)) = *Done*
⟨*proof*⟩

lemma *trace-not-Done-index*:
assumes *tr*: *trace tr* **and** *i*: *i* < *length tr* **and** *j*: *j* < *length tr* **and** *iltj*: *i* < *j* **and**
f: ¬ *final* (*cfgOf* (*tr*!*j*))
shows ¬ *final* (*cfgOf* (*tr* ! *i*))
⟨*proof*⟩

lemma *trace-not-Done-last*:
assumes *tr*: *trace tr* **and** *i*: *i* < *length tr* **and** *tr* ≠ [] **and**
f: ¬ *final* (*cfgOf* (*last tr*))
shows ¬ *final* (*cfgOf* (*tr* ! *i*))
⟨*proof*⟩

lemma *trace-Atom-Suc*:
assumes *tr*: *trace tr* **and** *i*: *Suc i* < *length tr* **and** *f*: *fst* (*cfgOf* (*tr*!*i*)) = *Atom a*
shows (*fst* (*cfgOf* (*tr*!(*Suc i*))) = *Done* ∧ *isS* (*tr* ! *Suc i*)) ∨
(*fst* (*cfgOf* (*tr*!(*Suc i*))) = *Atom a* ∧ *isE* (*tr* ! *Suc i*))
⟨*proof*⟩

lemma *trace-Atom-tl*:
assumes *tr*: *trace tr* **and** *ntr*: *tr* ≠ [] **and** *f*: *fst* (*cfgOf* (*hd tr*)) = *Atom a*
shows ∃ *sl tr*′. *tl tr* = *map* (λ*s*. *E* (*Atom a*,*s*)) *sl* @ *tr*′ ∧ *trace tr*′ ∧
(*tr*′ = [] ∨ *fst* (*cfgOf* (*hd tr*′)) = *Done* ∧ *isS* (*hd tr*′))
⟨*proof*⟩

lemma *trace-Atom*:
assumes *tr*: *trace tr tr* ≠ [] **and** *f*: *fst* (*cfgOf* (*hd tr*)) = *Atom a*
shows (∃ *sl*. *tr* = *hd tr* # *map* (λ*s*. *E* (*Atom a*,*s*)) *sl*) ∨
(∃ *sl s*′ *sl*′. *tr* = *hd tr* # *map* (λ*s*. *E* (*Atom a*,*s*)) *sl* @ [*S* (*Done*,*s*′)] @ *map* (λ*s*. *E* (*Done*,*s*)) *sl*′)
⟨*proof*⟩

lemma *trace-Seq-Suc*:
assumes *tr*: *trace tr* **and** *i*: *Suc i* < *length tr* **and** *f*: *fst* (*cfgOf* (*tr*!*i*)) = *Seq c1 c2*
shows (∃ *c1*′. *c1* ≠ *Done* ∧ *small-step* (*c1*,*snd* (*cfgOf* (*tr*!*i*))) (*c1*′,*snd* (*cfgOf* (*tr*!(*Suc i*)))) ∧
fst (*cfgOf* (*tr*!(*Suc i*))) = *Seq c1*′ *c2* ∧ *isS* (*tr* ! *Suc i*))
∨
(*c1* = *Done* ∧ *fst* (*cfgOf* (*tr*!(*Suc i*))) = *c2* ∧ *isS* (*tr* ! *Suc i*))
∨

$(fst\ (cfgOf\ (tr!(Suc\ i))) = Seq\ c1\ c2 \wedge isE\ (tr\ !\ Suc\ i))$
⟨*proof*⟩

definition *makeSeq* :: $('atom,'test)com \Rightarrow (('atom,'test)com,'state)acfg \Rightarrow (('atom,'test)com,'state)acfg$
where
makeSeq c2 acfg ≡ *updCfg acfg* (*Seq* (*fst* (*cfgOf acfg*)) *c2*,*snd* (*cfgOf acfg*))

lemma *snd-cfgOf-makeSeq*[*simp*]: *snd* (*cfgOf* (*makeSeq c2 acfg*)) = *snd* (*cfgOf acfg*)
⟨*proof*⟩

lemma *isE-makeSeq*[*simp*]: *isE* (*makeSeq c2 acfg*) = *isE acfg*
⟨*proof*⟩

lemma *isS-makeSeq*[*simp*]: *isS* (*makeSeq c2 acfg*) = *isS acfg*
⟨*proof*⟩

lemma *map-eq-length*:
assumes *map* (*snd* ∘ *cfgOf*) *tr* = *map* (*snd* ∘ *cfgOf*) (*map* (*makeSeq c2*) *tr1* @ *tr2*)
shows *length tr* = *length tr1* + *length tr2*
⟨*proof*⟩

lemma *map-cfgOf-makeSeq1*:
assumes *map* (*snd* ∘ *cfgOf*) *tr* = *map* (*snd* ∘ *cfgOf*) (*map* (*makeSeq c2*) *tr1* @ *tr2*)
and *i* < *length tr1*
shows *snd* (*cfgOf* (*tr*!*i*)) = *snd* (*cfgOf* (*tr1*!*i*))
⟨*proof*⟩

lemma *map-cfgOf-makeSeq2*:
assumes *map* (*snd* ∘ *cfgOf*) *tr* = *map* (*snd* ∘ *cfgOf*) (*map* (*makeSeq c2*) *tr1* @ *tr2*)
and *i* < *length tr2*
shows *snd* (*cfgOf* (*tr*!(*length tr1*+*i*))) = *snd* (*cfgOf* (*tr2*!*i*))
⟨*proof*⟩

lemma *map-cfgOf-makeSeq2'*:
assumes *map* (*snd* ∘ *cfgOf*) *tr* = *map* (*snd* ∘ *cfgOf*) (*map* (*makeSeq c2*) *tr1* @ *tr2*)
and *i* ≥ *length tr1 i* < *length tr*
shows *snd* (*cfgOf* (*tr*!*i*)) = *snd* (*cfgOf* (*tr2*!(*i*−*length tr1*)))
⟨*proof*⟩

lemma *list-all2-isE-makeSeq1*:
assumes *list-all2* (λ*acfg acfg'*. *isE acfg* = *isE acfg'*) *tr* (*map* (*makeSeq c2*) *tr1* @ *tr2*)
and *i* < *length tr1*

shows *isE* (*tr*!*i*) = *isE* (*tr1*!*i*)
⟨*proof*⟩

lemma *list-all2-isE-makeSeq2*:
assumes *list-all2* (λ*acfg acfg'*. *isE acfg* = *isE acfg'*) *tr* (*map* (*makeSeq c2*) *tr1* @ *tr2*)
and *i* < *length tr2*
shows *isE* (*tr*!(*length tr1* + *i*)) = *isE* (*tr2*!*i*)
⟨*proof*⟩

lemma *list-all2-isE-makeSeq2'*:
assumes *list-all2* (λ*acfg acfg'*. *isE acfg* = *isE acfg'*) *tr* (*map* (*makeSeq c2*) *tr1* @ *tr2*)
and *i* ≥ *length tr1 i* < *length tr*
shows *isE* (*tr*!*i*) = *isE* (*tr2*!(*i*−*length tr1*))
⟨*proof*⟩

lemma *trace-Seq*:
assumes *tr*: *trace tr* **and** *ntr*: *tr* ≠ [] **and** *c*: *fst* (*cfgOf* (*hd tr*)) = *Seq c1 c2*
shows (∃ *tr1*. *trace tr1* ∧ *tr1* ≠ [] ∧ *fst* (*cfgOf* (*hd tr1*)) = *c1* ∧ *tr* = *map* (*makeSeq c2*) *tr1*) ∨
(∃ *tr1 tr2*. *trace tr1* ∧ *tr1* ≠ [] ∧ *fst* (*cfgOf* (*hd tr1*)) = *c1* ∧ *fst* (*cfgOf* (*last tr1*)) = *Done* ∧
trace tr2 ∧ *tr2* ≠ [] ∧ *fst* (*cfgOf* (*hd tr2*)) = *c2* ∧
snd (*cfgOf* (*last tr1*)) = *snd* (*cfgOf* (*hd tr2*)) ∧ *isS* (*hd tr2*) ∧
tr = *map* (*makeSeq c2*) *tr1* @ *tr2*)
⟨*proof*⟩

lemma *trace-If*:
assumes *tr*: *trace tr* **and** *ntr*: *tr* ≠ [] **and** *f*: *fst* (*cfgOf* (*hd tr*)) = *If t c1 c2*
shows ∃ *sl tr'*. *tl tr* = *map* (λ*s*. *E* (*If t c1 c2*,*s*)) *sl* @ *tr'* ∧
(*tr'* ≠ [] ⟶
trace tr' ∧ *isS* (*hd tr'*) ∧
fst (*cfgOf* (*hd tr'*)) = (*if evalT t* (*snd* (*cfgOf* (*hd tr'*))) *then c1 else c2*) ∧
snd (*cfgOf* (*tr*!(*length sl*))) = *snd* (*cfgOf* (*hd tr'*)))
⟨*proof*⟩

lemma *trace-While*:
assumes *tr*: *trace tr* **and** *ntr*: *tr* ≠ [] **and** *f*: *fst* (*cfgOf* (*hd tr*)) = *While t c1*
shows ∃ *sl tr'*. *tl tr* = *map* (λ*s*. *E* (*While t c1*,*s*)) *sl* @ *tr'* ∧

$(tr' \neq [] \longrightarrow$
$trace\ tr' \wedge isS\ (hd\ tr') \wedge$
$fst\ (cfgOf\ (hd\ tr')) = If\ t\ (Seq\ c1\ (While\ t\ c1))\ Done \wedge$
$snd\ (cfgOf\ (tr!(length\ sl))) = snd\ (cfgOf\ (hd\ tr')))$
⟨*proof*⟩

definition *envOrIdle* **where**
$envOrIdle\ tr \equiv \forall i.\ Suc\ i < length\ tr \longrightarrow isE\ (tr!Suc\ i) \vee snd\ (cfgOf\ (tr!i)) = snd\ (cfgOf\ (tr!Suc\ i))$

lemma *envOrIdle-append*:
assumes *envOrIdle tr1 envOrIdle tr2*
$tr1 \neq [] \Longrightarrow tr2 \neq [] \Longrightarrow isE\ (hd\ tr2) \vee snd\ (cfgOf\ (last\ tr1)) = snd\ (cfgOf\ (hd\ tr2))$
shows $envOrIdle\ (tr1\ @\ tr2)$
⟨*proof*⟩

lemma *envOrIdle-Nil*[*simp*,*intro*!]: *envOrIdle* []
⟨*proof*⟩

lemma *envOrIdle-singl*[*simp*,*intro*!]: $envOrIdle\ [acfg]$
⟨*proof*⟩

lemma *envOrIdle-map-E*[*simp*,*intro*!]: $envOrIdle\ (map\ (\lambda s.\ E\ (c,s))\ sl)$
⟨*proof*⟩

lemma *envOrIdle-tl*:
assumes $envOrIdle\ (tl\ tr)$
$tl\ tr \neq [] \Longrightarrow isE\ (hd\ (tl\ tr)) \vee snd\ (cfgOf\ (hd\ tr)) = snd\ (cfgOf\ (hd\ (tl\ tr)))$
shows *envOrIdle tr*
⟨*proof*⟩

lemma *trace-While2*:
assumes *tr*: *trace tr* **and** *ntr*: $tr \neq []$ **and** *f*: $fst\ (cfgOf\ (hd\ tr)) = While\ t\ c1$
shows
— The loop-terminating case:
$(\exists tr1\ tr2.\ tr = tr1\ @\ tr2 \wedge$
$tr1 \neq [] \wedge trace\ tr1 \wedge envOrIdle\ tr1 \wedge Done \notin (fst\ o\ cfgOf)\ `\ set\ tr1$
$\wedge$
$(tr2 \neq [] \longrightarrow \neg\ evalT\ t\ (snd\ (cfgOf\ (last\ tr1))) \wedge fst\ (cfgOf\ (last\ tr1)) = If\ t\ (Seq\ c1\ (While\ t\ c1))\ Done \wedge$
$trace\ tr2 \wedge fst\ (cfgOf\ (hd\ tr2)) = Done \wedge isS\ (hd\ tr2) \wedge snd\ (cfgOf\ (hd\ tr2)) = snd\ (cfgOf\ (last\ tr1))))$
$\vee$
— The loop-continuing case:
$(\exists tr1\ tr2\ tr3.\ tr = tr1\ @\ map\ (makeSeq\ (While\ t\ c1))\ tr2\ @\ tr3 \wedge$
$tr1 \neq [] \wedge trace\ tr1 \wedge envOrIdle\ tr1 \wedge Done \notin (fst\ o\ cfgOf)\ `\ set\ tr1$

$\wedge$
$tr2 \neq [] \wedge evalT\ t\ (snd\ (cfgOf\ (last\ tr1))) \wedge fst\ (cfgOf\ (last\ tr1)) = If\ t\ (Seq\ c1\ (While\ t\ c1))\ Done \wedge$
$trace\ tr2 \wedge fst\ (cfgOf\ (hd\ tr2)) = c1 \wedge isS\ (hd\ tr2) \wedge snd\ (cfgOf\ (hd\ tr2)) = snd\ (cfgOf\ (last\ tr1)) \wedge$
$Done \notin (fst\ o\ cfgOf)\ `\ set\ (map\ (makeSeq\ (While\ t\ c1))\ tr2)$
$\wedge$
$(tr3 \neq [] \longrightarrow fst\ (cfgOf\ (last\ tr2)) = Done \wedge snd\ (cfgOf\ (hd\ tr3)) = snd\ (cfgOf\ (last\ tr2)) \wedge$
$trace\ tr3 \wedge fst\ (cfgOf\ (hd\ tr3)) = While\ t\ c1))$
⟨*proof*⟩

lemma *small-step-not-same-c*: $(c, s) \rightarrow (c', s') \Longrightarrow (c \neq c')$
⟨*proof*⟩

lemma *isS-iff*:
assumes $Suc\ i < length\ tr$
assumes *trace tr*
shows $isS\ (tr\ !\ (Suc\ i)) \longleftrightarrow cfgOf\ (tr\ !\ i) \rightarrow cfgOf\ (tr\ !\ (Suc\ i))$
⟨*proof*⟩

lemma *isE-iff*:
assumes $Suc\ i < length\ tr$
assumes *trace tr*
shows $isE\ (tr\ !\ (Suc\ i)) \longleftrightarrow fst\ (cfgOf\ (tr\ !\ i)) = fst\ (cfgOf\ (tr\ !\ (Suc\ i)))$
⟨*proof*⟩

lemma *trace-Par-Done-Split*:
assumes *tr*: *trace tr* **and** *ntr*: $tr \neq []$ **and** *c*: $fst\ (cfgOf\ (hd\ tr)) = c1\ ||\ c2$
shows $\exists\ tr1\ tr2\ .trace\ tr1 \wedge trace\ tr2 \wedge tr = tr1\ @\ tr2 \wedge ((final\ (cfgOf\ (last\ tr)) \wedge isS\ (hd\ tr2) \wedge final\ (cfgOf(hd\ (tr2))) \wedge$
$fst\ (cfgOf\ (last\ tr1)) = Done\ ||\ Done \wedge snd(cfgOf\ (last\ tr1)) = snd\ (cfgOf\ (hd\ (tr2)))) \vee (\neg final\ (cfgOf\ (last\ tr)) \wedge tr1 = tr \wedge tr2 = []))$
⟨*proof*⟩

lemma *trace-snoc-S*:
assumes $trace\ (tr\ @\ [x])$ **assumes** $isS\ x$ **assumes** $tr \neq []$
shows $(cfgOf\ (last\ tr) \rightarrow cfgOf\ x)$
⟨*proof*⟩

lemma *trace-snoc-E*:
assumes $trace\ (tr\ @\ [x])$ **assumes** $isE\ x$ **assumes** $tr \neq []$
shows $fst\ (cfgOf\ (last\ tr)) = fst(\ cfgOf\ x)$
⟨*proof*⟩

definition
isStepLeft tr trl i ≡ *isS* (*tr* ! (*Suc i*)) ⟶ (∀ *c1 c1′ c2* . *c1′* ≠ *c1* ∧ *fst* (*cfgOf* (*tr* ! *i*)) = *c1* || *c2*
∧ *fst* (*cfgOf* (*tr* ! (*Suc i*))) = *c1′* || *c2* ⟶ *isS* (*trl* ! (*Suc i*)) ∧ *fst* (*cfgOf* (*trl* ! (*Suc i*))) = *c1′*)

definition
isStepRight tr trr i ≡ *isS* (*tr* ! (*Suc i*)) ⟶ (∀ *c1 c2 c2′* . *c2′* ≠ *c2* ∧ *fst* (*cfgOf* (*tr* ! *i*)) = *c1* || *c2*
∧ *fst* (*cfgOf* (*tr* ! (*Suc i*))) = *c1* || *c2′* ⟶ *isS* (*trr* ! (*Suc i*)) ∧ *fst* (*cfgOf* (*trr* ! (*Suc i*))) = *c2′*)

lemma *isStepLeftSingleton*: *Suc i* < *length* [*x*] ⟹ *isStepLeft* [*x*] *tr i* = *True*
⟨*proof*⟩

lemma *isStepRightSingleton*: *Suc i* < *length* [*x*] ⟹ *isStepRight* [*x*] *tr i* = *True*
⟨*proof*⟩

lemma *isStepLeftTail*: *length trl* = *length* (*a* # *tr*) ⟹ *Suc* (*Suc i*) < *length* (*a* # *tr*) ⟹ *isStepLeft* (*a* # *tr*) *trl* ((*Suc i*)) ⟹
isStepLeft tr (*tl trl*) *i*
⟨*proof*⟩

lemma *nth-snoc*: *i* < *length tr* ⟹ (*tr* @ [*x*]) ! *i* = *tr* ! *i*
⟨*proof*⟩

lemma *isStepLeftsnoc*:
assumes *length trl* = *length tr* (*Suc i*) < *length* (*tr* @[*x*])
assumes ⋀ *i* . (*Suc i*) < *length tr* ⟹ *isStepLeft tr trl i*
assumes *fst* (*cfgOf* (*last tr*)) = *c1* || *c2*
assumes (*isS x* ⟹ (*fst* (*cfgOf* (*x*)) = *c1′* || *c2* ∧ *x1* = *S* (*c1′*, *snd* (*cfgOf x*)))
∨ (*fst* (*cfgOf* (*x*)) = *c1* || *c2′*))
shows *isStepLeft* (*tr* @ [*x*]) (*trl* @ [*x1*]) *i*
⟨*proof*⟩

lemma *isStepRightTail*: *length trr* = *length* (*a* # *tr*) ⟹ *Suc* (*Suc i*) < *length* (*a* # *tr*) ⟹ *isStepRight* (*a* # *tr*) *trr* ((*Suc i*)) ⟹
isStepRight tr (*tl trr*) (*i*)
⟨*proof*⟩

lemma *isStepRightsnoc*:
assumes *length trr* = *length tr* (*Suc i*) < *length* (*tr* @[*x*])
assumes ⋀ *i* . (*Suc i*) < *length tr* ⟹ *isStepRight tr trr i*
assumes *fst* (*cfgOf* (*last tr*)) = *c1* || *c2*
assumes (*isS x* ⟹ (*fst* (*cfgOf* (*x*)) = *c1* || *c2′* ∧ *x2* = *S* (*c2′*, *snd* (*cfgOf x*)))
∨ (*fst* (*cfgOf* (*x*)) = *c1′* || *c2*))

shows *isStepRight* (*tr* @ [*x*]) (*trr* @ [*x2*]) *i*
⟨*proof*⟩

definition
isSiff tr trl trr i ≡ (*isS* (*tr* ! *i*) ⟷ ((*isS* (*trl* ! *i*) ∧ *isE* (*trr* ! *i*)) ∨ (*isE* (*trl* ! *i*)) ∧ *isS* (*trr* ! *i*)))

lemma *isSiffTail*: *length tr1* = *length* (*a* # *tr*) ⟹ *length* (*tr2*) = *length* (*a* # *tr*) ⟹ *tr* ≠ [] ⟹
Suc i < *length* (*a* # *tr*) ⟹ *isSiff* (*a* # *tr*) *tr1 tr2* (*Suc i*) ⟹ *isSiff* (*tr*) (*tl tr1*) (*tl tr2*) *i*
⟨*proof*⟩

lemma *isSiffsnoc1*:
assumes *length tr1* = *length tr length tr2* = *length tr i* < *length tr isSiff tr tr1 tr2 i*
shows *isSiff* (*tr* @ [*x*]) (*tr1* @ [*x1*]) (*tr2* @ [*x2*]) *i*
⟨*proof*⟩

lemma *isSiffsnoc*:
assumes *length tr1* = *length tr length tr2* = *length tr i* < *length* (*tr* @ [*x*])
assumes ⋀ *i* . *i* < *length tr* ⟹ *isSiff tr tr1 tr2 i*
assumes (*isS x* ⟷ ((*isS x1* ∧ *isE x2*) ∨ (*isE x1*) ∧ *isS x2*))
shows *isSiff* (*tr* @ [*x*]) (*tr1* @ [*x1*]) (*tr2* @ [*x2*]) *i*
⟨*proof*⟩

lemma *isEiffTail*: *length tr1* = *length* (*a* # *tr*) ⟹ *length* (*tr2*) = *length* (*a* # *tr*) ⟹ *tr* ≠ [] ⟹
Suc i < *length* (*a* # *tr*) ⟹ *isE* ((*a* # *tr*) ! *Suc i*) ⟷ *isE* (*tr1* ! (*Suc i*)) ∧ *isE* (*tr2* ! (*Suc i*)) ⟹
isE ((*tr*) ! *i*) ⟷ *isE* (*tl tr1* ! *i*) ∧ *isE* (*tl tr2* ! *i*)
⟨*proof*⟩

lemma *isEiffsnoc*:
assumes *length tr1* = *length tr length tr2* = *length tr i* < *length* (*tr* @ [*x*])
assumes ⋀ *i* . *i* < *length tr* ⟹ *isE* (*tr* ! *i*) ⟷ *isE* (*tr1* ! *i*) ∧ *isE* (*tr2* ! *i*)
assumes (*isE x* ⟷ *isE x1* ∧ *isE x2*)
shows *isE* ((*tr* @ [*x*]) ! *i*) = (*isE* ((*tr1* @ [*x1*]) ! *i*) ∧ *isE* ((*tr2* @ [*x2*]) ! *i*))
⟨*proof*⟩

lemma *sndeqsnoc*:
assumes (*i*<*length* (*tr* @ [*x*])) *length tr1* = *length tr length tr2* = *length tr*
assumes ⋀ *i* . *i*< *length tr* ⟹ *snd* (*cfgOf* ((*tr*) ! *i*)) = *snd* (*cfgOf* (*tr1* ! *i*)) ∧ *snd* (*cfgOf* ((*tr*) ! *i*)) = *snd* (*cfgOf* (*tr2* ! *i*))
assumes *snd* (*cfgOf x1*) = *snd* (*cfgOf x*) *snd* (*cfgOf x2*) = *snd* (*cfgOf x*)
shows *snd* (*cfgOf* ((*tr* @ [*x*]) ! *i*)) = *snd* (*cfgOf* ((*tr1* @ [*x1*]) ! *i*)) ∧ *snd* (*cfgOf*

$((tr\ @\ [x])\ !\ i)) = snd\ (cfgOf\ ((tr2\ @\ [x2])\ !\ i))$
⟨*proof*⟩

lemma *small-step-diff*: **assumes** $(c,\ s) \rightarrow (c',\ s')$
shows $c' \neq c$
⟨*proof*⟩

lemma *trace-Par-helper*:
assumes *tr*: *trace tr* **and** *ntr*: $tr \neq []$ **and** *c*: $fst\ (cfgOf\ (hd\ tr)) = c1\ ||\ c2$ **and** $\neg\ final\ (cfgOf\ (last\ tr))$
assumes *trace tr1* **and** *trace tr2* **and** $fst\ (cfgOf\ (hd\ tr1)) = c1$ **and** $fst\ (cfgOf\ (hd\ tr2)) = c2$
and *length tr = length tr1 length tr = length tr2* **and**
$\bigwedge i.\ i < length\ tr \Longrightarrow snd\ (cfgOf\ (tr\ !\ i)) = snd\ (cfgOf\ (tr1\ !\ i)) \wedge snd\ (cfgOf\ (tr\ !\ i)) = snd\ (cfgOf\ (tr2\ !\ i))$
and $\bigwedge i\ .\ i < length\ tr \Longrightarrow\ (isE\ (tr\ !\ i) \longleftrightarrow isE\ (tr1\ !\ i) \wedge isE\ (tr2\ !\ i))$
and $\bigwedge i\ .\ i < length\ tr \Longrightarrow isSiff\ tr\ tr1\ tr2\ i$
and $\bigwedge i.\ Suc\ i < length\ tr \Longrightarrow isStepLeft\ tr\ tr1\ i \wedge isStepRight\ tr\ tr2\ i$
shows $\exists\ c1'\ c2'\ .\ fst\ (cfgOf\ (last\ tr)) = c1'\ ||\ c2' \wedge fst\ (cfgOf\ (last\ tr1)) = c1' \wedge fst\ (cfgOf\ (last\ tr2)) = c2'$
⟨*proof*⟩

lemma *trace-Par-strong*:
assumes *tr*: *trace tr* **and** *ntr*: $tr \neq []$ **and** *c*: $fst\ (cfgOf\ (hd\ tr)) = c1\ ||\ c2$ **and** $\neg\ final\ (cfgOf\ (last\ tr))$
shows $\exists\ tr1\ tr2\ .\ trace\ tr1 \wedge trace\ tr2 \wedge fst\ (cfgOf\ (hd\ tr1)) = c1 \wedge fst\ (cfgOf\ (hd\ tr2)) = c2$
$\wedge\ length\ tr = length\ tr1 \wedge length\ tr = length\ tr2$
$\wedge\ ((fst\ (cfgOf\ (last\ tr)) = Done\ ||\ Done) \longleftrightarrow final\ (cfgOf\ (last\ tr1)) \wedge final\ (cfgOf\ (last\ tr2))) \wedge$
$(\forall\ i < length\ tr\ .\ snd\ (cfgOf\ (tr\ !\ i)) = snd\ (cfgOf\ (tr1\ !\ i)) \wedge snd\ (cfgOf\ (tr\ !\ i)) = snd\ (cfgOf\ (tr2\ !\ i)))$
$\wedge\ (\forall\ i < length\ tr\ .\ isE\ (tr\ !\ i) \longleftrightarrow isE\ (tr1\ !\ i) \wedge isE\ (tr2\ !\ i))$
$\wedge\ (\forall\ i < length\ tr\ .\ isSiff\ tr\ tr1\ tr2\ i)$
$\wedge$
$(\forall\ i.\ Suc\ i < length\ tr \longrightarrow isStepLeft\ tr\ tr1\ i \wedge isStepRight\ tr\ tr2\ i)$
⟨*proof*⟩

lemma *trace-Par*:
assumes *tr*: *trace tr* **and** *ntr*: $tr \neq []$ **and** *c*: $fst\ (cfgOf\ (hd\ tr)) = c1\ ||\ c2$ **and** $\neg\ final\ (cfgOf\ (last\ tr))$
shows $\exists\ tr1\ tr2\ .\ trace\ tr1 \wedge trace\ tr2 \wedge fst\ (cfgOf\ (hd\ tr1)) = c1 \wedge fst\ (cfgOf\ (hd\ tr2)) = c2$
$\wedge\ length\ tr = length\ tr1 \wedge length\ tr = length\ tr2$
$\wedge\ ((fst\ (cfgOf\ (last\ tr)) = Done\ ||\ Done) \longleftrightarrow final\ (cfgOf\ (last\ tr1)) \wedge final\ (cfgOf\ (last\ tr2))) \wedge$
$(\forall\ i < length\ tr\ .\ snd\ (cfgOf\ (tr\ !\ i)) = snd\ (cfgOf\ (tr1\ !\ i)) \wedge snd\ (cfgOf\ (tr\ !\ i)) = snd\ (cfgOf\ (tr2\ !\ i)))$

$\wedge$ ($\forall$ $i < length\ tr$. $isE\ (tr\ !\ i) \longleftrightarrow isE\ (tr1\ !\ i) \wedge isE\ (tr2\ !\ i)$)
$\wedge$ ($\forall$ $i < length\ tr$. $isSiff\ tr\ tr1\ tr2\ i$)
⟨*proof*⟩

end

end

6 Abstract Rely-Guarantee Satisfaction

This theory defines the core concepts of trace satisfiability for rely-guarantee reasoning, including precondition, postcondition, rely, and guarantee satisfaction over interactive traces.

theory *RG-Abstract*
imports *RG-Inversion-Rules*
begin

context *Step*
begin

definition *satPre* :: ('*com*, '*state*) *acfg list* ⇒ ('*state* ⇒ *bool*) ⇒ *bool* **where**
satPre tr P ≡ *P* (*snd* (*cfgOf* (*hd tr*)))

definition *satPost* :: ('*com*, '*state*) *acfg list* ⇒ ('*state* ⇒ *bool*) ⇒ *bool* **where**
satPost tr Q ≡ *final* (*cfgOf* (*last tr*)) ⟶ *Q* (*snd* (*cfgOf* (*last tr*)))

definition *satRely* :: ('*com*, '*state*) *acfg list* ⇒ ('*state* ⇒ '*state* ⇒ *bool*) ⇒ *bool* **where**
satRely tr R ≡ ∀ *i*. *Suc i* < *length tr* ∧ *isE* (*tr*!(*Suc i*)) ⟶ *R* (*snd* (*cfgOf* (*tr*!*i*))) (*snd* (*cfgOf* (*tr*!(*Suc i*))))

definition *satGuar* :: ('*com*, '*state*) *acfg list* ⇒ ('*state* ⇒ '*state* ⇒ *bool*) ⇒ *bool* **where**
satGuar tr G ≡ ∀ *i*. *Suc i* < *length tr* ∧ *isS* (*tr*!(*Suc i*)) ⟶ *G* (*snd* (*cfgOf* (*tr*!*i*))) (*snd* (*cfgOf* (*tr*!(*Suc i*))))

lemmas *satPost-defs* = *satPost-def lift2-def*

definition *sat* :: '*com* ⇒
('*state* ⇒ *bool*) ⇒
('*state* ⇒ *bool*) ⇒
('*state* ⇒ '*state* ⇒ *bool*) ⇒
('*state* ⇒ '*state* ⇒ *bool*) ⇒
bool **where**

$sat\ c\ P\ Q\ R\ G \equiv \forall\ s\ tr.\ trace\ tr \wedge tr \neq [] \wedge cfgOf\ (hd\ tr) = (c,s) \wedge$
$satPre\ tr\ P \wedge satRely\ tr\ R \longrightarrow satPost\ tr\ Q \wedge satGuar\ tr\ G$

lemma *satPre-take*: $satPre\ tr\ P \Longrightarrow i > 0 \Longrightarrow satPre\ (take\ i\ tr)\ P$
⟨*proof*⟩

lemma *satPre-triv*[*simp*]:$tr \neq [] \Longrightarrow cfgOf\ (hd\ tr) = (c,\ s) \Longrightarrow satPre\ tr\ P \Longrightarrow P\ s$ ⟨*proof*⟩
lemma *satPre-True*[*simp*]:$satPre\ tr\ (\lambda a.\ True)$ ⟨*proof*⟩

lemma *satPreI*[*intro*]:$P\ \sigma \Longrightarrow satPre\ [S\ (c,\ \sigma)]\ P$ ⟨*proof*⟩
lemma *satPostE*[*elim*]:$satPost\ [S\ (c,\ \sigma)]\ Q \Longrightarrow final\ (c,\ \sigma) \Longrightarrow Q\ \sigma$ ⟨*proof*⟩

lemma *satRely-not-isE-snoc*: $satRely\ tr\ R \Longrightarrow \neg\ isE\ acfg \Longrightarrow satRely\ (tr\ @\ [acfg])\ R$
⟨*proof*⟩

lemma *satRely-isS-snoc*: $satRely\ tr\ R \Longrightarrow isS\ acfg \Longrightarrow satRely\ (tr\ @\ [acfg])\ R$
⟨*proof*⟩

lemma *satRely-isS-take*: $i < length\ tr \Longrightarrow$
$satRely\ (take\ i\ tr)\ R \Longrightarrow isS\ (tr!i) \Longrightarrow satRely\ (take\ (Suc\ i)\ tr)\ R$
⟨*proof*⟩

lemma *satRely-tl*:$satRely\ tr\ R \Longrightarrow satRely\ (tl\ tr)\ R$ ⟨*proof*⟩

lemma *satRely-tl'*:$satRely\ (a\#tr)\ R \Longrightarrow satRely\ tr\ R$
⟨*proof*⟩

lemma *satRely-isE*[*simp*]:$filter\ isE\ tr = [] \Longrightarrow satRely\ tr\ R$ ⟨*proof*⟩

lemma *satRely-introS*:$satRely\ tr\ R \Longrightarrow cfgOf(hd\ tr) = (c',s') \Longrightarrow satRely\ (S\ (c,\ s)\ \#\ S\ (c',\ s')\ \#\ tl\ tr)\ R$
⟨*proof*⟩

lemma *satRely-introE*:$R\ s\ s' \Longrightarrow cfgOf\ (hd\ tr) = (c,\ s') \Longrightarrow satRely\ tr\ R \Longrightarrow satRely\ (E\ (c,\ s)\ \#\ E\ (c,\ s')\ \#\ tl\ tr)\ R$
⟨*proof*⟩

lemma *satRely-split1*:
assumes $satRely\ tr\ R$
assumes $tr = tr1\ @\ tr2$
shows $satRely\ tr1\ R$
⟨*proof*⟩

lemma *satRely-split2*:
assumes *satRely tr R*
assumes *tr = tr1 @ tr2*
shows *satRely tr2 R*
⟨*proof*⟩

lemmas *satRely-split = satRely-split1 satRely-split2*

lemma *satPost-hd-rmvS*:*tr* ≠ [] ⟹ *cfgOf* (*hd tr*) = (*c*′, *s*′) ⟹ *satPost* ([*S* (*c*,*s*),*S* (*c*′,*s*′)] @ *tl tr*) *Q* ⟹ *satPost tr Q*
⟨*proof*⟩

lemma *satPost-hd-rmvE*:*tr* ≠ [] ⟹ *cfgOf* (*hd tr*) = (*c*′, *s*′) ⟹ *satPost* ([*E* (*c*,*s*),*E* (*c*′,*s*′)] @ *tl tr*) *Q* ⟹ *satPost tr Q*
⟨*proof*⟩

lemma *satGuar-hd-rmvS*:*cfgOf* (*hd tr*) = (*c*′, *s*′) ⟹ *satGuar* ([*S* (*c*,*s*),*S* (*c*′,*s*′)] @ *tl tr*) *G* ⟹ *satGuar tr G*
⟨*proof*⟩

lemma *satGuar-hd-rmvE*:*cfgOf* (*hd tr*) = (*c*′, *s*′) ⟹ *satGuar* ([*E* (*c*,*s*),*E* (*c*′,*s*′)] @ *tl tr*) *G* ⟹ *satGuar tr G*
⟨*proof*⟩

lemma *satPost-hd-add*:*tr* = *x* # *xs* ⟹ *cfgOf x* = (*c*′, *s*′) ⟹ *satPost tr Q* ⟹
trace (*a* # *tr*) ⟹ *cfgOf a* = (*c*, *s*) ⟹ *satPost* (*a* # *tr*) *Q*
⟨*proof*⟩

lemma *satGuar-hd-add*:*tr* = *x* # *xs* ⟹ *cfgOf x* = (*c*′, *s*′) ⟹ *cfgOf a* = (*c*, *s*) ⟹
trace (*a* # *tr*) ⟹ *satGuar tr G* ⟹ (*x* = *S* (*c*′,*s*′) ⟶ *G s s*′) ⟹ *satGuar* (*a* # *tr*) *G*
⟨*proof*⟩

lemma *satGuar-not-isS-snoc*: *satGuar tr G* ⟹ ¬ *isS acfg* ⟹ *satGuar* (*tr* @ [*acfg*]) *G*
⟨*proof*⟩

lemma *satGuar-isE-snoc*: *satGuar tr G* ⟹ *isE acfg* ⟹ *satGuar* (*tr* @ [*acfg*]) *G*
⟨*proof*⟩

lemma *G-intro*:⋀*i c c*′. *cfgOf* (*tr* ! *i*) = (*c*,*s*) ⟹ *tr* ! *Suc i* = *S* (*c*′,*s*′) ⟹ *Suc i* < *length tr* ⟹ *satGuar tr G* ⟹ *G s s*′
⟨*proof*⟩

definition *reduced-sat cfg Q R G* $\equiv \forall$ *tr . trace tr* $\wedge$ *tr* $\neq$ [] $\wedge$ *cfgOf* (*hd tr*) = *cfg* $\wedge$
satRely tr R $\longrightarrow$ *satPost tr Q* $\wedge$ *satGuar tr G*

lemma *reduced-sat-sat*: ($\bigwedge$ *s . P s* $\Longrightarrow$ *reduced-sat* (*c*, *s*) *Q R G*) $\Longrightarrow$ *sat c P Q R G*
⟨*proof*⟩

lemma *sat-reduced-sat*: *sat c P Q R G* $\Longrightarrow$ *P s* $\Longrightarrow$ *reduced-sat* (*c*, *s*) *Q R G*
⟨*proof*⟩

lemma *reduced-sat-iff*: ($\forall$ *s . P s* $\longrightarrow$ *reduced-sat* (*c*, *s*) *Q R G*) $\longleftrightarrow$ *sat c P Q R G*
⟨*proof*⟩

lemma *reduced-sat-relQ*: *reduced-sat cfg Q R G* $\Longrightarrow$ *snd cfg* = *s* $\Longrightarrow$ *reduced-sat cfg Q R G*
⟨*proof*⟩

lemma *satPre-hd*[*simp*]:*cfgOf a* = (*c*,*s*) $\Longrightarrow$ *P s* $\Longrightarrow$ *satPre* (*a* # *tr*) *P* ⟨*proof*⟩

lemma *sat-step-G*:*P s* $\Longrightarrow$ (*c*, *s*) $\rightarrow$ (*c′*, *s′*) $\Longrightarrow$ *sat c P Q R G* $\Longrightarrow$ *G s s′*
⟨*proof*⟩

lemma *reduced-sat-step-G*: (*c*, *s*) $\rightarrow$ (*c′*, *s′*) $\Longrightarrow$ *reduced-sat* (*c*, *s*) *Q R G* $\Longrightarrow$ *G s s′*
⟨*proof*⟩

lemma *satPost-hd-rmvS2*:*tr* $\neq$ [] $\Longrightarrow$ *cfgOf* (*hd tr*) = (*c′*, *s′*) $\Longrightarrow$ *satPost* ([*S* (*c*,*s*),*S* (*c′*,*s′*)] @ *tl tr*) *Q* $\Longrightarrow$ *satPost tr Q*
⟨*proof*⟩

lemma *reduced-sat-step-pres*: *reduced-sat* (*c*, *s*) *Q R G* $\Longrightarrow$ (*c*, *s*) $\rightarrow$ (*c′*, *s′*) $\Longrightarrow$ *reduced-sat* (*c′*, *s′*) *Q R G*
⟨*proof*⟩

lemma *sat-step-pres*:*P s* $\Longrightarrow$
sat c P Q R G $\Longrightarrow$ *sat c′* (λ*a*. (*c*, *s*) $\rightarrow$ (*c′*, *a*)) *Q R G*
⟨*proof*⟩

lemma *reduced-sat-env-pres*:*reduced-sat* (*c*, *s*) *Q R G* $\Longrightarrow$ *R s s′* $\Longrightarrow$ *reduced-sat* (*c*, *s′*) *Q R G*

⟨*proof*⟩

lemma *sat-env-pres*:$P\ s \Longrightarrow sat\ c\ P\ Q\ R\ G \Longrightarrow R\ s\ s' \Longrightarrow sat\ c\ (R\ s)\ Q\ R\ G$
⟨*proof*⟩

lemma *sat-final-U*:$sat\ c\ P\ Q\ R\ G \Longrightarrow P\ s \Longrightarrow final\ (c,\ s) \Longrightarrow Q\ s$
⟨*proof*⟩

lemma *reduced-sat-final*:$reduced\text{-}sat\ (c,\ s)\ Q\ \ R\ G\ \Longrightarrow final\ (c,\ s) \Longrightarrow Q\ s$
⟨*proof*⟩

lemma *sat-final*:$sat\ c\ P\ Q\ R\ G \Longrightarrow P\ s \Longrightarrow final\ (c,\ s) \Longrightarrow Q\ s$
⟨*proof*⟩

lemma *satPre-mono*: $satPre\ tr\ P \Longrightarrow P \leq P' \Longrightarrow satPre\ tr\ P'$
⟨*proof*⟩

lemma *satRely-mono*: $satRely\ tr\ R \Longrightarrow R \leq R' \Longrightarrow satRely\ tr\ R'$
⟨*proof*⟩

lemma *satPost-mono*: $satPost\ tr\ Q' \Longrightarrow Q' \leq Q \Longrightarrow satPost\ tr\ Q$
⟨*proof*⟩

lemma *satGuar-mono*: $satGuar\ tr\ G' \Longrightarrow G' \leq G \Longrightarrow satGuar\ tr\ G$
⟨*proof*⟩

end

end

7 Unary Rely-Guarantee Soundness

This theory mechanises the soundness proofs of the trace-based semantics with respect to a standard unary rely-guarantee proof system.

theory *RG-Soundness*
imports *RG-Abstract*
begin

context *Step*
begin

lemma *satPre-mono*: $satPre\ tr\ P \Longrightarrow P \leq P' \Longrightarrow satPre\ tr\ P'$
⟨*proof*⟩

lemma *satRely-mono*: $satRely\ tr\ R \Longrightarrow R \leq R' \Longrightarrow satRely\ tr\ R'$

⟨*proof*⟩

lemma *satPost-mono*: *satPost tr* $Q' \Longrightarrow Q' \leq Q \Longrightarrow$ *satPost tr* Q
⟨*proof*⟩

lemma *satGuar-mono*: *satGuar tr* $G' \Longrightarrow G' \leq G \Longrightarrow$ *satGuar tr* G
⟨*proof*⟩

proposition *RG-Mono*:
assumes *sat c* P' Q' R' G'
assumes $P \leq P'$ $R \leq R'$ $G' \leq G$ $Q' \leq Q$
shows *sat c* P Q R G
⟨*proof*⟩

end

context *SeqParWhileLang*
begin

proposition *RG-Done*:
assumes *Ps-Rl*: $\forall\ s\ s'.\ P\ s \wedge R\hat{}{*}{*}\ s\ s' \longrightarrow Q\ s'$
shows *sat Done P Q R G*
⟨*proof*⟩

proposition *RG-Atom*:
assumes *st*: *stable P R stable Q R*
and *Q*: *imR* $(\lambda s\ s'.\ s' =$ *evalA a s*$)$ $P \leq Q$
and *G*: *lift1 P* $\wedge 2$ $(\lambda s\ s'.\ s' =$ *evalA a s*$) \leq G$
shows *sat* $(Atom\ a)$ *P Q R G*
⟨*proof*⟩

proposition *RG-Seq*:
assumes *sat1*: *sat c1 P P' R G*
and *sat2*: *sat c2 P' Q R G*
and *Grid*: $(=) \leq G$
shows *sat* $(Seq\ c1\ c2)$ *P Q R G*
⟨*proof*⟩

proposition *RG-If*:
assumes *st*: *stable P R*
and *sat1*: *sat c1* $(P \wedge 1\ (evalT\ t))$ *Q R G*

and *sat2*: *sat c2* (*P* ∧*1* (¬*1 evalT t*)) *Q R G*
and *G*: (=) ≤ *G*
shows *sat* (*If t c1 c2*) *P Q R G*
⟨*proof*⟩

proposition *RG-While*:
assumes *st*: *stable P R stable Q R*
and *sat1*: *sat c1* ((*evalT t*) ∧*1 P*) *Q1 R G*
and *P*: *Q1* ≤ *P*
and *Q*: *P* ∧*1* (¬*1 evalT t*) ≤ *Q*
and *G*: (=) ≤ *G*
shows *sat* (*While t c1*) *P Q R G*
⟨*proof*⟩

corollary *RG-While-U*:
assumes *st*: *stable P R stable UPt R*
and *sat1*: *sat c1* ((*evalT t*) ∧*1 P*) *P R G*
and *Q*: *P* ∧*1* (¬*1 evalT t*) ≤ *UPt*
and *G*: (=) ≤ *G*
shows *sat* (*While t c1*) *P UPt R G*
⟨*proof*⟩

proposition *RG-Par*:
assumes *c1-RG*: *sat c1 Pr1 Pt1 Rl1 Gr1*
assumes *c2-RG*: *sat c2 Pr2 Pt2 Rl2 Gr2*
assumes *pre*: *P* ≤ *Pr1 P* ≤ *Pr2*
assumes *rely*: ⋀ *s s'* . *R s s'* ∨ *Gr2 s s'* ⟹ *Rl1 s s'* ⋀ *s s'*. *R s s'* ∨ *Gr1 s s'* ⟹ *Rl2 s s'*
assumes *post*: ⋀ *s s'* . *Pt1 s'* ∧ *Pt2 s'* ⟹ *Q s'*
assumes *guar*: ⋀ *s s'*. *Gr1 s s'* ∨ *Gr2 s s'* ⟹ *G s s'*
assumes *greq*: (=) ≤ *G*
shows *sat* (*Par c1 c2*) *P Q R G*
⟨*proof*⟩

thm *RG-Atom*

proposition *RG-Skip-U*:
assumes *st*: *stable P R stable UPt R*
and *Q*: *P* ≤ *UPt*
and *G*: *lift1 P* ∧*2* (=) ≤ *G*

shows *sat* (*Atom Skip*) *P UPt R G*
⟨*proof*⟩

proposition *RG-Await-U*:
assumes *st*: *stable P R stable UPt R*
and *Q*: *P* ∧*1* (*evalT t*) ≤ *UPt*
and *G*: (=) ≤ *G*
shows *sat* (*Await t*) *P UPt R G*
⟨*proof*⟩
end

end

8 Binary Abstract Rely-Guarantee Satisfaction

This theory presents the abstract infrastructure for binary rely-guarantee reasoning, where postconditions are relational (linking initial and final states) over interactive traces.

theory *RG-Abstract-Binary*
imports *RG-Inversion-Rules*
begin

context *Step*
begin

definition *satPre* :: (*'com*, *'state*) *acfg list* ⇒ (*'state* ⇒ *bool*) ⇒ *bool* **where**
satPre tr P ≡ *P* (*snd* (*cfgOf* (*hd tr*)))

definition *satPost* :: (*'com*, *'state*) *acfg list* ⇒ (*'state* ⇒ *'state* ⇒ *bool*) ⇒ *bool* **where**
satPost tr Q ≡ *final* (*cfgOf* (*last tr*)) ⟶ *Q* (*snd* (*cfgOf* (*hd tr*))) (*snd* (*cfgOf* (*last tr*)))

definition *satRely* :: (*'com*, *'state*) *acfg list* ⇒ (*'state* ⇒ *'state* ⇒ *bool*) ⇒ *bool* **where**
satRely tr R ≡ ∀ *i*. *Suc i* < *length tr* ∧ *isE* (*tr*!(*Suc i*)) ⟶ *R* (*snd* (*cfgOf* (*tr*!*i*))) (*snd* (*cfgOf* (*tr*!(*Suc i*))))

definition *satGuar* :: (*'com*, *'state*) *acfg list* ⇒ (*'state* ⇒ *'state* ⇒ *bool*) ⇒ *bool* **where**
satGuar tr G ≡ ∀ *i*. *Suc i* < *length tr* ∧ *isS* (*tr*!(*Suc i*)) ⟶ *G* (*snd* (*cfgOf* (*tr*!*i*))) (*snd* (*cfgOf* (*tr*!(*Suc i*))))

lemmas *satPost-defs* = *satPost-def lift2-def*

definition *sat* :: *'com* ⇒
(*'state* ⇒ *bool*) ⇒
(*'state* ⇒ *'state* ⇒ *bool*) ⇒
(*'state* ⇒ *'state* ⇒ *bool*) ⇒
(*'state* ⇒ *'state* ⇒ *bool*) ⇒
bool **where**
sat c P Q R G ≡ ∀ *s tr*. *trace tr* ∧ *tr* ≠ [] ∧ *cfgOf* (*hd tr*) = (*c*,*s*) ∧
satPre tr P ∧ *satRely tr R* ⟶ *satPost tr Q* ∧ *satGuar tr G*

lemma *satPre-take*: *satPre tr P* ⟹ *i* > *0* ⟹ *satPre* (*take i tr*) *P*
⟨*proof*⟩

lemma *satPre-triv*[*simp*]:*tr*≠[] ⟹ *cfgOf* (*hd tr*) = (*c*, *s*) ⟹ *satPre tr P* ⟹ *P*
s ⟨*proof*⟩
lemma *satPre-True*[*simp*]:*satPre tr* (λ*a*. *True*) ⟨*proof*⟩

lemma *satPreI*[*intro*]:*P σ* ⟹ *satPre* [*S* (*c*, *σ*)] *P* ⟨*proof*⟩
lemma *satPostE*[*elim*]:*satPost* [*S* (*c*, *σ*)] (*lift2 Q*) ⟹ *final* (*c*, *σ*) ⟹ *Q σ* ⟨*proof*⟩

lemma *satRely-not-isE-snoc*: *satRely tr R* ⟹ ¬ *isE acfg* ⟹ *satRely* (*tr* @ [*acfg*])
R
⟨*proof*⟩

lemma *satRely-isS-snoc*: *satRely tr R* ⟹ *isS acfg* ⟹ *satRely* (*tr* @ [*acfg*]) *R*
⟨*proof*⟩

lemma *satRely-isS-take*: *i* < *length tr* ⟹
satRely (*take i tr*) *R* ⟹ *isS* (*tr*!*i*) ⟹ *satRely* (*take* (*Suc i*) *tr*) *R*
⟨*proof*⟩

lemma *satRely-tl*:*satRely tr R* ⟹ *satRely* (*tl tr*) *R* ⟨*proof*⟩

lemma *satRely-tl'*:*satRely* (*a*#*tr*) *R* ⟹ *satRely tr R*
⟨*proof*⟩

lemma *satRely-isE*[*simp*]:*filter isE tr* = [] ⟹ *satRely tr R* ⟨*proof*⟩

lemma *satRely-introS*:*satRely tr R* ⟹ *cfgOf*(*hd tr*) = (*c'*,*s'*) ⟹ *satRely* (*S* (*c*,
s) # *S* (*c'*, *s'*) # *tl tr*) *R*
⟨*proof*⟩

lemma *satRely-introE*:*R s s'* ⟹ *cfgOf* (*hd tr*) = (*c*, *s'*) ⟹ *satRely tr R* ⟹*satRely*
(*E* (*c*, *s*) # *E* (*c*, *s'*) # *tl tr*) *R*
⟨*proof*⟩

lemma *satRely-split1*:
assumes *satRely tr R*
assumes *tr = tr1 @ tr2*
shows *satRely tr1 R*
⟨*proof*⟩

lemma *satRely-split2*:
assumes *satRely tr R*
assumes *tr = tr1 @ tr2*
shows *satRely tr2 R*
⟨*proof*⟩

lemmas *satRely-split = satRely-split1 satRely-split2*

lemma *satPost-hd-rmvS*:*tr* ≠ [] ⟹ *cfgOf* (*hd tr*) = (*c*′, *s*′) ⟹ *satPost* ([*S* (*c*,*s*),*S* (*c*′,*s*′)] @ *tl tr*) (*lift2 Q*) ⟹ *satPost tr* (*lift2 Q*)
⟨*proof*⟩

lemma *satPost-hd-rmvE*:*tr* ≠ [] ⟹ *cfgOf* (*hd tr*) = (*c*′, *s*′) ⟹ *satPost* ([*E* (*c*,*s*),*E* (*c*′,*s*′)] @ *tl tr*) (*lift2 Q*) ⟹ *satPost tr* (*lift2 Q*)
⟨*proof*⟩

lemma *satGuar-hd-rmvS*:*cfgOf* (*hd tr*) = (*c*′, *s*′) ⟹ *satGuar* ([*S* (*c*,*s*),*S* (*c*′,*s*′)] @ *tl tr*) *G* ⟹ *satGuar tr G*
⟨*proof*⟩

lemma *satGuar-hd-rmvE*:*cfgOf* (*hd tr*) = (*c*′, *s*′) ⟹ *satGuar* ([*E* (*c*,*s*),*E* (*c*′,*s*′)] @ *tl tr*) *G* ⟹ *satGuar tr G*
⟨*proof*⟩

lemma *satPost-hd-add*:*tr* = *x* # *xs* ⟹ *cfgOf x* = (*c*′, *s*′) ⟹ *satPost tr* (*lift2 Q*) ⟹
trace (*a* # *tr*) ⟹ *cfgOf a* = (*c*, *s*) ⟹ *satPost* (*a* # *tr*) (*lift2 Q*)
⟨*proof*⟩

lemma *satPost-hd-add2*:*tr* = *x* # *xs* ⟹ *cfgOf x* = (*c*′, *s*′) ⟹ *satRely* (*t* # *tr*) *R* ⟹
satPost tr (λ*ss ss*′. ((*c*, *s*) → (*c*′, *ss*) ∨ *R s ss*) ∧ *Q s ss*′) ⟹
trace (*a* # *tr*) ⟹ *cfgOf a* = (*c*, *s*) ⟹ *satPost* (*a* # *tr*) *Q*
⟨*proof*⟩

lemma *satGuar-hd-add*:*tr* = *x* # *xs* ⟹ *cfgOf x* = (*c*′, *s*′) ⟹ *cfgOf a* = (*c*, *s*) ⟹
trace (*a* # *tr*) ⟹ *satGuar tr G* ⟹ (*x* = *S* (*c*′,*s*′) ⟶ *G s s*′)

$\Longrightarrow$ *satGuar* (*a* # *tr*) *G*
⟨*proof*⟩

lemma *satGuar-not-isS-snoc*: *satGuar tr G* $\Longrightarrow$ ¬ *isS acfg* $\Longrightarrow$ *satGuar* (*tr* @ [*acfg*]) *G*
⟨*proof*⟩

lemma *satGuar-isE-snoc*: *satGuar tr G* $\Longrightarrow$ *isE acfg* $\Longrightarrow$ *satGuar* (*tr* @ [*acfg*]) *G*
⟨*proof*⟩

lemma *G-intro*:$\bigwedge$*i c c'*. *cfgOf* (*tr* ! *i*) = (*c*,*s*) $\Longrightarrow$ *tr* ! *Suc i* = *S* (*c'*,*s'*) $\Longrightarrow$ *Suc i* < *length tr* $\Longrightarrow$ *satGuar tr G* $\Longrightarrow$ *G s s'*
⟨*proof*⟩

definition *reduced-sat cfg Q R G* ≡ ∀ *tr* . *trace tr* ∧ *tr* ≠ [] ∧ *cfgOf* (*hd tr*) = *cfg* ∧
satRely tr R ⟶ *satPost tr Q* ∧ *satGuar tr G*

lemma *reduced-sat-sat*: (⋀ *s* . *P s* $\Longrightarrow$ *reduced-sat* (*c*, *s*) *Q R G*) $\Longrightarrow$ *sat c P Q R G*
⟨*proof*⟩

lemma *sat-reduced-sat*: *sat c P Q R G* $\Longrightarrow$ *P s* $\Longrightarrow$ *reduced-sat* (*c*, *s*) *Q R G*
⟨*proof*⟩

lemma *reduced-sat-iff*: (∀ *s* . *P s* ⟶ *reduced-sat* (*c*, *s*) *Q R G*) ⟷ *sat c P Q R G*
⟨*proof*⟩

lemma *reduced-sat-relQ*: *reduced-sat cfg Q R G* $\Longrightarrow$ *snd cfg* = *s* $\Longrightarrow$ *reduced-sat cfg* (*lift2* (*Q s*)) *R G*
⟨*proof*⟩

lemma *satPre-hd*[*simp*]:*cfgOf a* = (*c*,*s*) $\Longrightarrow$ *P s* $\Longrightarrow$ *satPre* (*a* # *tr*) *P* ⟨*proof*⟩

lemma *sat-step-G*:*P s* $\Longrightarrow$ (*c*, *s*) → (*c'*, *s'*) $\Longrightarrow$ *sat c P Q R G* $\Longrightarrow$ *G s s'*
⟨*proof*⟩

lemma *reduced-sat-step-G*: (*c*, *s*) → (*c'*, *s'*) $\Longrightarrow$ *reduced-sat* (*c*, *s*) *Q R G* $\Longrightarrow$ *G s s'*
⟨*proof*⟩

lemma *satPost-hd-rmvS2*:*tr* $\neq$ [] $\Longrightarrow$ *cfgOf* (*hd tr*) = (*c'*, *s'*) $\Longrightarrow$ *satPost* ([*S* (*c*,*s*),*S* (*c'*,*s'*)] @ *tl tr*) *Q* $\Longrightarrow$ *satPost tr* (*lift2* (*Q s*))
⟨*proof*⟩

lemma *reduced-sat-step-pres*: *reduced-sat* (*c*, *s*) (*lift2 Q*) *R G* $\Longrightarrow$ (*c*, *s*) $\rightarrow$ (*c'*, *s'*) $\Longrightarrow$ *reduced-sat* (*c'*, *s'*) (*lift2 Q*) *R G*
⟨*proof*⟩

lemma *sat-step-pres*:*P s* $\Longrightarrow$
sat c P (*lift2 Q*) *R G* $\Longrightarrow$ *sat c'* (λ*a*. (*c*, *s*) $\rightarrow$ (*c'*, *a*)) (*lift2 Q*) *R G*
⟨*proof*⟩

lemma *reduced-sat-env-pres*:*reduced-sat* (*c*, *s*) (*lift2 Q*) *R G* $\Longrightarrow$ *R s s'* $\Longrightarrow$ *reduced-sat* (*c*, *s'*) (*lift2 Q*) *R G*
⟨*proof*⟩

lemma *sat-env-pres*:*P s* $\Longrightarrow$ *sat c P* (*lift2 Q*) *R G* $\Longrightarrow$ *R s s'* $\Longrightarrow$ *sat c* (*R s*) (*lift2 Q*) *R G*
⟨*proof*⟩

lemma *sat-final-U*:*sat c P* (*lift2 Q*) *R G* $\Longrightarrow$ *P s* $\Longrightarrow$ *final* (*c*, *s*) $\Longrightarrow$ *Q s*
⟨*proof*⟩

lemma *reduced-sat-final*:*reduced-sat* (*c*, *s*) (*lift2* (*Q s1*)) *R G* $\Longrightarrow$ *final* (*c*, *s*) $\Longrightarrow$ *Q s1 s*
⟨*proof*⟩

lemma *sat-final*:*sat c P* (*lift2* (*Q s1*)) *R G* $\Longrightarrow$ *P s* $\Longrightarrow$ *final* (*c*, *s*) $\Longrightarrow$ *Q s1 s*
⟨*proof*⟩

end

end

9 Binary Rely-Guarantee Soundness

This theory mechanizes the soundness of our trace-based RG semantics with respect to the relational RG proof system

theory *RG-Soundness-Binary*
imports *RG-Abstract-Binary*
begin

context *Step*
begin

lemma *satPre-mono*: $satPre\ tr\ P \Longrightarrow P \leq P' \Longrightarrow satPre\ tr\ P'$
⟨*proof*⟩

lemma *satRely-mono*: $satRely\ tr\ R \Longrightarrow R \leq R' \Longrightarrow satRely\ tr\ R'$
⟨*proof*⟩

lemma *satPost-mono*: $satPost\ tr\ Q' \Longrightarrow Q' \leq Q \Longrightarrow satPost\ tr\ Q$
⟨*proof*⟩

lemma *satGuar-mono*: $satGuar\ tr\ G' \Longrightarrow G' \leq G \Longrightarrow satGuar\ tr\ G$
⟨*proof*⟩

proposition *RG-Mono*:
assumes $sat\ c\ P'\ Q'\ R'\ G'$
assumes $P \leq P'\ R \leq R'\ G' \leq G\ Q' \leq Q$
shows $sat\ c\ P\ Q\ R\ G$
⟨*proof*⟩

end

context *SeqParWhileLang*
begin

proposition *RG-Done*:
assumes *Ps-Rl*: $lift1\ P \wedge 2\ R\hat{}** \leq Ps$
shows $sat\ Done\ P\ Ps\ R\ G$
⟨*proof*⟩

proposition *RG-Atom*:

assumes *Ps-R*: $(lift1\ P \wedge 2\ R\hat{}**)\ OO\ (\lambda s\ s'.\ s' = evalA\ a\ s)\ OO\ R\hat{}** \leq Q$
and *G*: $lift1\ (imR\ (R\hat{}**)\ P) \wedge 2\ (\lambda s\ s'.\ s' = evalA\ a\ s) \leq G$
shows $sat\ (Atom\ a)\ P\ Q\ R\ G$
⟨*proof*⟩

corollary *RG-Atom'*:
assumes *st*: $stable\ P\ R\ stable2\ Q\ R$
and *Q*: $lift1\ P \wedge 2\ (\lambda s\ s'.\ s' = evalA\ a\ s) \leq Q$
and *G*: $lift1\ P \wedge 2\ (\lambda s\ s'.\ s' = evalA\ a\ s) \leq G$
shows $sat\ (Atom\ a)\ P\ Q\ R\ G$
⟨*proof*⟩

corollary *RG-Atom-U*:

assumes *st*: *stable P R stable UPt R*
and *Q*: *imR* ($\lambda s\ s'.\ s' = evalA\ a\ s$) $P \leq UPt$
and *G*: *lift1 P* $\wedge 2$ ($\lambda s\ s'.\ s' = evalA\ a\ s$) $\leq G$
shows *sat* (*Atom a*) *P* (*lift2 UPt*) *R G*
⟨*proof*⟩

proposition *RG-Seq*:
assumes *sat1*: *sat c1 Pr1 Pt1 Rl1 Gr1*
and *sat2*: *sat c2 Pr2 Pt2 Rl2 Gr2*
and *P*: $P \leq Pr1$
and *Pr12*: *imR Pt1 Pr1* $\leq Pr2$
and *Q*: *Pt1 OO Pt2* $\leq Q$
and *R*: $R \leq Rl1\ R \leq Rl2$
and *G*: $Gr1 \leq G\ Gr2 \leq G$
and *Grid*: $(=) \leq G$
shows *sat* (*Seq c1 c2*) *P Q R G*
⟨*proof*⟩

corollary *RG-Seq′*:
assumes *sat1*: *sat c1 Pr1 Pt1 R G*
and *sat2*: *sat c2 Pr2 Pt2 R G*
and *P*: $P \leq Pr1$
and *Pr12*: *imR Pt1 Pr1* $\leq Pr2$
and *Q*: *Pt1 OO Pt2* $\leq Q$
and *Grid*: $(=) \leq G$
shows *sat* (*Seq c1 c2*) *P Q R G*
⟨*proof*⟩

corollary *RG-Seq-U*:
assumes *sat1*: *sat c1 P* (*lift2 Pr2*) *R G*
and *sat2*: *sat c2 Pr2* (*lift2 UPt*) *R G*
and *Grid*: $(=) \leq G$
shows *sat* (*Seq c1 c2*) *P* (*lift2 UPt*) *R G*
⟨*proof*⟩

proposition *RG-If*:
assumes *sat1*: *sat c1 Pr1 Pt1 Rl1 Gr1*
and *sat2*: *sat c2 Pr2 Pt2 Rl2 Gr2*
and *Pr1*: (*imR R*$\widehat{}$$**$ *P*) $\wedge 1$ (*evalT t*) $\leq Pr1$
and *Pr2*: (*imR R*$\widehat{}$$**$ *P*) $\wedge 1$ ($\neg 1$ *evalT t*) $\leq Pr2$
and *R*: $R \leq Rl1\ R \leq Rl2$
and *Q*: ((*R*$\widehat{}$$**$ $\wedge 2$ *lift2* (*evalT t*)) *OO Pt1*) $\leq Q$ ((*R*$\widehat{}$$**$ $\wedge 2$ *lift2* ($\neg 1$ *evalT t*)) *OO Pt2*) $\leq Q$

and G: $Gr1 \leq G\ Gr2 \leq G\ (=) \leq G$
shows $sat\ (If\ t\ c1\ c2)\ P\ Q\ R\ G$
⟨*proof*⟩

corollary *RG-If′*:
assumes *st*: $stable\ P\ R\ stableLeft\ Q\ R$

and *sat1*: $sat\ c1\ Pr1\ Pt1\ Rl1\ Gr1$
and *sat2*: $sat\ c2\ Pr2\ Pt2\ Rl2\ Gr2$
and *Pr1*: $P \wedge 1\ (evalT\ t) \leq Pr1$
and *Pr2*: $P \wedge 1\ (\neg 1\ evalT\ t) \leq Pr2$
and *R*: $R \leq Rl1\ R \leq Rl2$
and *Q*: $lift1\ (evalT\ t) \wedge 2\ Pt1 \leq Q\ lift1\ (\neg 1\ evalT\ t) \wedge 2\ Pt2 \leq Q$
and *G*: $Gr1 \leq G\ Gr2 \leq G\ (=) \leq G$
shows $sat\ (If\ t\ c1\ c2)\ P\ Q\ R\ G$
⟨*proof*⟩

corollary *RG-If″*:
assumes *st*: $stable\ P\ R\ stableLeft\ Q\ R$
and *sat1*: $sat\ c1\ (P \wedge 1\ (evalT\ t))\ Q\ R\ G$
and *sat2*: $sat\ c2\ (P \wedge 1\ (\neg 1\ evalT\ t))\ Q\ R\ G$
and *G*: $(=) \leq G$
shows $sat\ (If\ t\ c1\ c2)\ P\ Q\ R\ G$
⟨*proof*⟩

corollary *RG-If-U*:
assumes *st*: $stable\ P\ R$
and *sat1*: $sat\ c1\ (P \wedge 1\ (evalT\ t))\ (lift2\ UPt)\ R\ G$
and *sat2*: $sat\ c2\ (P \wedge 1\ (\neg 1\ evalT\ t))\ (lift2\ UPt)\ R\ G$
and *G*: $(=) \leq G$
shows $sat\ (If\ t\ c1\ c2)\ P\ (lift2\ UPt)\ R\ G$
⟨*proof*⟩

proposition *RG-While*:
assumes *st*: $stable\ P\ R\ stable2\ Q\ R\ stable2\ Pt1\ R$
and *sat1*: $sat\ c1\ Pr1\ Pt1\ Rl1\ Gr1$
and *Pr1*: $(evalT\ t) \wedge 1\ P \leq Pr1$
and *P*: $imR\ Pt1\ ((evalT\ t) \wedge 1\ Pr1) \leq P$
and *R*: $R \leq Rl1$
and *Q*: $lift1\ P \wedge 2\ (lift1\ (evalT\ t \wedge 1\ P) \wedge 2\ Pt1)\hat{}** \wedge 2\ lift2\ (\neg 1\ evalT\ t) \leq Q$
and *G*: $Gr1 \leq G\ (=) \leq G$
shows $sat\ (While\ t\ c1)\ P\ Q\ R\ G$
⟨*proof*⟩

corollary *RG-While′*:
assumes *st*: $stable\ P\ R\ stable2\ Q\ R\ stable2\ Pt1\ R$
and *sat1*: $sat\ c1\ ((evalT\ t) \wedge 1\ P)\ Pt1\ R\ G$

and P: $imR\ Pt1\ ((evalT\ t) \wedge 1\ P) \leq P$
and Q: $lift1\ P \wedge 2\ (lift1\ (evalT\ t \wedge 1\ P) \wedge 2\ Pt1)^{**} \wedge 2\ lift2\ (\neg 1\ evalT\ t) \leq Q$
and G: $(=) \leq G$
shows $sat\ (While\ t\ c1)\ P\ Q\ R\ G$
⟨*proof*⟩

corollary *RG-While''-alt*:
assumes *st*: $stable\ P\ R\ stable2\ Q\ R\ stable2\ Pt1\ R$
and *sat1*: $sat\ c1\ P\ Pt1\ R\ G$
and P: $imR\ Pt1\ ((evalT\ t) \wedge 1\ P) \leq P$
and Q: $lift1\ P \wedge 2\ (lift1\ (evalT\ t \wedge 1\ P) \wedge 2\ Pt1)^{**} \wedge 2\ lift2\ (\neg 1\ evalT\ t) \leq Q$
and G: $(=) \leq G$
shows $sat\ (While\ t\ c1)\ P\ Q\ R\ G$
⟨*proof*⟩

corollary *RG-While-U*:
assumes *st*: $stable\ P\ R\ stable\ UPt\ R$
and *sat1*: $sat\ c1\ ((evalT\ t) \wedge 1\ P)\ (lift2\ P)\ R\ G$
and Q: $P \wedge 1\ (\neg 1\ evalT\ t) \leq UPt$
and G: $(=) \leq G$
shows $sat\ (While\ t\ c1)\ P\ (lift2\ UPt)\ R\ G$
⟨*proof*⟩

corollary *RG-While-U-alt*:
assumes *st*: $stable\ P\ R\ stable\ UPt\ R$
and *sat1*: $sat\ c1\ P\ (lift2\ P)\ R\ G$
and Q: $P \wedge 1\ (\neg 1\ evalT\ t) \leq UPt$
and G: $(=) \leq G$
shows $sat\ (While\ t\ c1)\ P\ (lift2\ UPt)\ R\ G$
⟨*proof*⟩

thm *RG-Atom*

proposition *RG-Skip*:
assumes *Ps-R*: $(lift1\ P \wedge 2\ R^{**})\ OO\ R^{**} \leq Q$
and G: $lift1\ (imR\ (R^{**})\ P) \wedge 2\ (=) \leq G$
shows $sat\ (Atom\ Skip)\ P\ Q\ R\ G$
⟨*proof*⟩

corollary *RG-Skip'*:
assumes *st*: $stable\ P\ R\ stable2\ Q\ R$
and Q: $lift1\ P \wedge 2\ (=) \leq Q$
and G: $lift1\ P \wedge 2\ (=) \leq G$
shows $sat\ (Atom\ Skip)\ P\ Q\ R\ G$
⟨*proof*⟩

corollary *RG-Skip-U*:
assumes *st*: *stable P R stable UPt R*
and *Q*: $P \leq UPt$
and *G*: $lift1\ P \wedge 2\ (=) \leq G$
shows *sat* (*Atom Skip*) *P* (*lift2 UPt*) *R G*
⟨*proof*⟩

proposition *RG-Par*:
assumes *c1-RG*: *sat c1 Pr1 Pt1 Rl1 Gr1*
assumes *c2-RG*: *sat c2 Pr2 Pt2 Rl2 Gr2*
assumes *pre*: $P \leq Pr1\ P \leq Pr2$
assumes *rely*: $\bigwedge s\ s'\ .\ R\ s\ s' \vee Gr2\ s\ s' \Longrightarrow Rl1\ s\ s' \bigwedge s\ s'.\ R\ s\ s' \vee Gr1\ s\ s' \Longrightarrow Rl2\ s\ s'$
assumes *post*: $\bigwedge s\ s'\ .\ Pt1\ s\ s' \wedge Pt2\ s\ s' \Longrightarrow Q\ s\ s'$
assumes *guar*: $\bigwedge s\ s'.\ Gr1\ s\ s' \vee Gr2\ s\ s' \Longrightarrow G\ s\ s'$
assumes *greq*: $(=) \leq G$
shows *sat* (*Par c1 c2*) *P Q R G*
⟨*proof*⟩

proposition *RG-Await*:
assumes *st*: *stable P R stable2 Q R stable2 Pt1 R*
and *st1*: *stable Pr1 Rl1 stable2 Pt1 Rl1*
and *sat1*: $lift1\ Pr1 \wedge 2\ (=) \leq Pt1\ lift1\ Pr1 \wedge 2\ (=) \leq Gr1$
and *Pr1*: $(\neg 1\ evalT\ t) \wedge 1\ P \leq Pr1$
and *P*: $imR\ Pt1\ ((\neg 1\ evalT\ t) \wedge 1\ Pr1) \leq P$
and *R*: $R \leq Rl1$
and *Q*: $lift1\ P \wedge 2\ (lift1\ ((\neg 1\ evalT\ t) \wedge 1\ P) \wedge 2\ Pt1)\hat{}** \wedge 2\ lift2\ (evalT\ t) \leq Q$
and *G*: $Gr1 \leq G\ (=) \leq G$
shows *sat* (*Await t*) *P Q R G*
⟨*proof*⟩

corollary *RG-Await'*:
assumes *st*: *stable P R stable2 Q R*

and *sat1*: $lift1\ P \wedge 2\ (=) \leq Q\ lift1\ P \wedge 2\ (=) \leq G$
and *P*: $imR\ Q\ ((\neg 1\ evalT\ t) \wedge 1\ P) \leq P$
and *Q*: $lift1\ P \wedge 2\ (lift1\ ((\neg 1\ evalT\ t) \wedge 1\ P) \wedge 2\ Q)\hat{}** \wedge 2\ lift2\ (evalT\ t) \leq Q$
and *G*: $(=) \leq G$
shows *sat* (*Await t*) *P Q R G*
⟨*proof*⟩

corollary *RG-Await-U*:
assumes *st*: *stable P R stable UPt R*
and *Q*: $P \wedge 1\ (evalT\ t) \leq UPt$

and G: $(=) \leq G$
shows *sat* (*Await t*) *P* (*lift2 UPt*) *R G*
⟨*proof*⟩
end

end

References

[1] J. Derrick, C. Edmonds, A. Popescu, and J. Wright, "Rely-guarantee is coinductive: a proof-centered investigation of inductively approximated coinduction ," in *Programming Languages and Systems: 35th European Symposium on Programming, ESOP 2026, Held as Part of the International Joint Conferences on Theory and Practice of Software, ETAPS 2026, Turin, Italy, April 1116, 2026, Proceedings, Part I.* Berlin, Heidelberg: Springer-Verlag, 2026, p. 220251. [Online]. Available: https://doi.org/10.1007/978-3-032-22720-1_9

[2] Q. Xu, W. P. de Roever, and J. He, "The rely-guarantee method for verifying shared variable concurrent programs," *Form. Asp. Comput.*, vol. 9, no. 2, p. 149174, Mar. 1997. [Online]. Available: https://doi.org/10.1007/BF01211617