A Generic Framework for Verified Compilers

Martin Desharnais

September 4, 2026

Abstract

This is a generic framework for formalizing compiler transformations. It leverages Isabelle/HOLs locales to abstract over concrete languages and transformations. It states common definitions for language semantics, program behaviours, forward and backward simulations, and compilers. We provide generic operations, such as simulation and compiler composition, and prove general (partial) correctness theorems, resulting in reusable proof components. For more details, please see our paper [1].

Contents

1 Infinitely Transitive Closure **2**

2 The Dynamic Representation of a Language **3**
2.1 Behaviour of a dynamic execution 4
2.2 Safe states 5

3 The Static Representation of a Language **5**
3.1 Program behaviour 6

4 Well-foundedness of Relations Defined as Predicate Functions **6**
4.1 Lexicographic product 6
4.2 Lexicographic list 7

5 Simulations Between Dynamic Executions **11**
5.1 Backward simulation 11
5.1.1 Preservation of behaviour 12
5.2 Forward simulation 12
5.2.1 Preservation of behaviour 13
5.2.2 Forward to backward 13
5.3 Bisimulation 13
5.4 Composition of simulations 16

5.4.1 Composition of backward simulations 16
5.4.2 Composition of forward simulations 16
5.4.3 Composition of bisimulations 17
5.5 Miscellaneous 17

6 Compiler Between Static Representations **18**
6.1 Preservation of behaviour 19
6.2 Composition of compilers 20

7 Fixpoint of Converging Program Transformations **20**

theory *Behaviour*
imports *Main*
begin

datatype *'state behaviour* =
Terminates 'state | *Diverges* | *is-wrong*: *Goes-wrong 'state*

Terminating behaviours are annotated with the last state of the execution in order to compare the result of two executions with the *rel-behaviour* relation.

The exact meaning of the three behaviours is defined in the semantics locale

end

1 Infinitely Transitive Closure

theory *Inf*
imports *Main*
begin

coinductive *inf* :: $('a \Rightarrow 'a \Rightarrow bool) \Rightarrow 'a \Rightarrow bool$ **for** *r* **where**
inf-step: $r\ x\ y \Longrightarrow inf\ r\ y \Longrightarrow inf\ r\ x$

coinductive *inf-wf* :: $('a \Rightarrow 'a \Rightarrow bool) \Rightarrow ('b \Rightarrow 'b \Rightarrow bool) \Rightarrow 'b \Rightarrow 'a \Rightarrow bool$
for *r order* **where**
inf-wf: $order\ n\ m \Longrightarrow inf\text{-}wf\ r\ order\ n\ x \Longrightarrow inf\text{-}wf\ r\ order\ m\ x$ |
inf-wf-step: $r^{++}\ x\ y \Longrightarrow inf\text{-}wf\ r\ order\ n\ y \Longrightarrow inf\text{-}wf\ r\ order\ m\ x$

lemma *inf-wf-to-step-inf-wf*:
assumes *wfp order*
shows $inf\text{-}wf\ r\ order\ n\ x \Longrightarrow \exists\, y\ m.\ r\ x\ y \wedge inf\text{-}wf\ r\ order\ m\ y$
⟨*proof*⟩

lemma *inf-wf-to-inf*:
assumes *wfp order*
shows $inf\text{-}wf\ r\ order\ n\ x \Longrightarrow inf\ r\ x$
⟨*proof*⟩

lemma *step-inf*:

assumes *right-unique r*
shows $r\ x\ y \Longrightarrow inf\ r\ x \Longrightarrow inf\ r\ y$
⟨*proof*⟩

lemma *star-inf*:
assumes *right-unique r*
shows $r^{**}\ x\ y \Longrightarrow inf\ r\ x \Longrightarrow inf\ r\ y$
⟨*proof*⟩

end
theory *Transfer-Extras*
imports *Main*
begin

lemma *rtranclp-complete-run-right-unique*:
fixes $R :: 'a \Rightarrow 'a \Rightarrow bool$ **and** $x\ y\ z :: 'a$
assumes *right-unique R*
shows $R^{**}\ x\ y \Longrightarrow (\nexists w.\ R\ y\ w) \Longrightarrow R^{**}\ x\ z \Longrightarrow (\nexists w.\ R\ z\ w) \Longrightarrow y = z$
⟨*proof*⟩

lemma *tranclp-complete-run-right-unique*:
fixes $R :: 'a \Rightarrow 'a \Rightarrow bool$ **and** $x\ y\ z :: 'a$
assumes *right-unique R*
shows $R^{++}\ x\ y \Longrightarrow (\nexists w.\ R\ y\ w) \Longrightarrow R^{++}\ x\ z \Longrightarrow (\nexists w.\ R\ z\ w) \Longrightarrow y = z$
⟨*proof*⟩

end

2 The Dynamic Representation of a Language

theory *Semantics*
imports *Main Behaviour Inf Transfer-Extras* **begin**

The definition of programming languages is separated into two parts: an abstract semantics and a concrete program representation.

definition *finished* :: $('a \Rightarrow 'a \Rightarrow bool) \Rightarrow 'a \Rightarrow bool$ **where**
finished r x = $(\nexists y.\ r\ x\ y)$

lemma *finished-star*:
assumes *finished r x*
shows $r^{**}\ x\ y \Longrightarrow x = y$
⟨*proof*⟩

locale *semantics* =
fixes
step :: $'state \Rightarrow 'state \Rightarrow bool$ (**infix** ‹→› *50*) **and**
final :: $'state \Rightarrow bool$
assumes
final-finished: $final\ s \Longrightarrow finished\ step\ s$

begin

The semantics locale represents the semantics as an abstract machine. It is expressed by a transition system with a transition relation $(\rightarrow)$—usually written as an infix $\rightarrow$ arrow—and final states *final*.

lemma *finished-step*:
step s s' $\Longrightarrow \neg$*finished step s*
⟨*proof*⟩

abbreviation *eval* :: *'state* $\Rightarrow$ *'state* $\Rightarrow$ *bool* (**infix** $\langle\rightarrow^*\rangle$ *50*) **where**
eval $\equiv$ *step***

abbreviation *inf-step* :: *'state* $\Rightarrow$ *bool* **where**
inf-step $\equiv$ *inf step*

notation
inf-step ($\langle$*'*$(\rightarrow^\infty$*')*$\rangle$ [] *50*) **and**
inf-step ($\langle$*-* $\rightarrow^\infty\rangle$ [*55*] *50*)

lemma *inf-not-finished*: *s* $\rightarrow^\infty \Longrightarrow \neg$ *finished step s*
⟨*proof*⟩

lemma *eval-deterministic*:
assumes
deterministic: $\bigwedge$*x y z. step x y* $\Longrightarrow$ *step x z* $\Longrightarrow$ *y* = *z* **and**
s1 $\rightarrow^*$ *s2* **and** *s1* $\rightarrow^*$ *s3* **and** *finished step s2* **and** *finished step s3*
shows *s2* = *s3*
⟨*proof*⟩

lemma *step-converges-or-diverges*: $(\exists$ *s'. s* $\rightarrow^*$ *s'* $\wedge$ *finished step s'*$) \vee$ *s* $\rightarrow^\infty$
⟨*proof*⟩

2.1 Behaviour of a dynamic execution

inductive *state-behaves* :: *'state* $\Rightarrow$ *'state behaviour* $\Rightarrow$ *bool* (**infix** $\langle\downarrow\rangle$ *50*) **where**
state-terminates:
s1 $\rightarrow^*$ *s2* $\Longrightarrow$ *finished step s2* $\Longrightarrow$ *final s2* $\Longrightarrow$ *s1* $\downarrow$ (*Terminates s2*) |
state-diverges:
s1 $\rightarrow^\infty \Longrightarrow$ *s1* $\downarrow$ *Diverges* |
state-goes-wrong:
s1 $\rightarrow^*$ *s2* $\Longrightarrow$ *finished step s2* $\Longrightarrow \neg$ *final s2* $\Longrightarrow$ *s1* $\downarrow$ (*Goes-wrong s2*)

Even though the $(\rightarrow)$ transition relation in the *semantics* locale need not be deterministic, if it happens to be, then the behaviour of a program becomes deterministic too.

lemma *right-unique-state-behaves*:
assumes
right-unique $(\rightarrow)$
shows *right-unique* $(\downarrow)$

⟨*proof*⟩

lemma *left-total-state-behaves*: *left-total* (↓)
⟨*proof*⟩

2.2 Safe states

definition *safe* **where**
safe s ⟷ (∀ *s'*. *step*** *s s'* ⟶ *final s'* ∨ (∃ *s''*. *step s' s''*))

lemma *final-safeI*: *final s* ⟹ *safe s*
⟨*proof*⟩

lemma *step-safe*: *step s s'* ⟹ *safe s* ⟹ *safe s'*
⟨*proof*⟩

lemma *steps-safe*: *step*** *s s'* ⟹ *safe s* ⟹ *safe s'*
⟨*proof*⟩

lemma *safe-state-behaves-not-wrong*:
assumes *safe s* **and** *s* ↓ *b*
shows ¬ *is-wrong b*
⟨*proof*⟩

end

end

3 The Static Representation of a Language

theory *Language*
imports *Semantics*
begin

locale *language* =
semantics step final
for
step :: *'state* ⇒ *'state* ⇒ *bool* **and**
final :: *'state* ⇒ *bool* +
fixes
load :: *'prog* ⇒ *'state* ⇒ *bool*

context *language* **begin**

The language locale represents the concrete program representation (type variable *'prog*), which can be transformed into a program state (type variable *'state*) by the *load* function. The set of initial states of the transition system is implicitly defined by the codomain of *load*.

3.1 Program behaviour

definition *prog-behaves* :: *'prog* ⇒ *'state behaviour* ⇒ *bool* (**infix** ‹⇓› *50*) **where**
prog-behaves = *load OO state-behaves*

If both the *load* and *step* relations are deterministic, then so is the behaviour of a program.

lemma *right-unique-prog-behaves*:
assumes
right-unique-load: *right-unique load* **and**
right-unique-step: *right-unique step*
shows *right-unique prog-behaves*
⟨*proof*⟩

end

end

4 Well-foundedness of Relations Defined as Predicate Functions

theory *Well-founded*
imports *Main*
begin

4.1 Lexicographic product

context
fixes
r1 :: *'a* ⇒ *'a* ⇒ *bool* **and**
r2 :: *'b* ⇒ *'b* ⇒ *bool*
begin

definition *lex-prodp* :: *'a* × *'b* ⇒ *'a* × *'b* ⇒ *bool* **where**
lex-prodp x y ≡ *r1* (*fst x*) (*fst y*) ∨ *fst x* = *fst y* ∧ *r2* (*snd x*) (*snd y*)

lemma *lex-prodp-lex-prod*:
shows *lex-prodp x y* ⟷ (*x*, *y*) ∈ *lex-prod* { (*x*, *y*). *r1 x y* } { (*x*, *y*). *r2 x y* }
⟨*proof*⟩

lemma *lex-prodp-wfP*:
assumes
wfp r1 **and**
wfp r2
shows *wfp lex-prodp*
⟨*proof*⟩

end

4.2 Lexicographic list

context
  fixes *order* :: $'a \Rightarrow 'a \Rightarrow bool$
begin

inductive *lexp* :: $'a\ list \Rightarrow 'a\ list \Rightarrow bool$ **where**
  lexp-head: *order x y* $\Longrightarrow$ *length xs* = *length ys* $\Longrightarrow$ *lexp* (*x* # *xs*) (*y* # *ys*) |
  lexp-tail: *lexp xs ys* $\Longrightarrow$ *lexp* (*x* # *xs*) (*x* # *ys*)

end

lemma *lexp-prepend*: *lexp order ys zs* $\Longrightarrow$ *lexp order* (*xs* @ *ys*) (*xs* @ *zs*)
  ⟨*proof*⟩

lemma *lexp-lex*: *lexp order xs ys* $\longleftrightarrow$ (*xs*, *ys*) $\in$ *lex* {(*x*, *y*). *order x y*}
⟨*proof*⟩

lemma *lex-list-wfP*: *wfP order* $\Longrightarrow$ *wfP* (*lexp order*)
  ⟨*proof*⟩

end
theory *Lifting-Simulation-To-Bisimulation*
  imports *Well-founded*
begin

definition *stuck-state* :: $('a \Rightarrow 'b \Rightarrow bool) \Rightarrow 'a \Rightarrow bool$ **where**
  stuck-state $\mathcal{R}$ *s* $\longleftrightarrow$ ($\nexists s'.\ \mathcal{R}\ s\ s'$)

definition *simulation* ::
  $('a \Rightarrow 'a \Rightarrow bool) \Rightarrow ('b \Rightarrow 'b \Rightarrow bool) \Rightarrow ('c \Rightarrow 'a \Rightarrow 'b \Rightarrow bool) \Rightarrow ('c \Rightarrow 'c \Rightarrow bool) \Rightarrow bool$
where
  simulation $\mathcal{R}_1$ $\mathcal{R}_2$ *match order* $\longleftrightarrow$
    ($\forall$ *i s1 s2 s1'*. *match i s1 s2* $\longrightarrow$ $\mathcal{R}_1$ *s1 s1'* $\longrightarrow$
      ($\exists$ *s2' i'*. ${\mathcal{R}_2}^{++}$ *s2 s2'* $\wedge$ *match i' s1' s2'*) $\vee$ ($\exists$ *i'*. *match i' s1' s2* $\wedge$ *order i' i*))

lemma *finite-progress*:
  fixes
    step1 :: $'s1 \Rightarrow 's1 \Rightarrow bool$ **and**
    step2 :: $'s2 \Rightarrow 's2 \Rightarrow bool$ **and**
    match :: $'i \Rightarrow 's1 \Rightarrow 's2 \Rightarrow bool$ **and**
    order :: $'i \Rightarrow 'i \Rightarrow bool$
  assumes
    matching-states-agree-on-stuck:
     $\forall$ *i s1 s2*. *match i s1 s2* $\longrightarrow$ *stuck-state step1 s1* $\longleftrightarrow$ *stuck-state step2 s2* **and**
    well-founded-order: *wfp order* **and**
    sim: *simulation step1 step2 match order*
  shows *match i s1 s2* $\Longrightarrow$ *step1 s1 s1'* $\Longrightarrow$

$\exists\, m\ s1''\ n\ s2''\ i'.\ (step1\ ^{\frown\frown}\ m)\ s1'\ s1'' \wedge (step2\ ^{\frown\frown}\ Suc\ n)\ s2\ s2'' \wedge match\ i'\ s1''\ s2''$
⟨*proof*⟩

context begin

private inductive *match-bisim*
for $\mathcal{R}_1 :: 'a \Rightarrow 'a \Rightarrow bool$ **and** $\mathcal{R}_2 :: 'b \Rightarrow 'b \Rightarrow bool$ **and**
match $:: 'c \Rightarrow 'a \Rightarrow 'b \Rightarrow bool$ **and** *order* $:: 'c \Rightarrow 'c \Rightarrow bool$
where
bisim-stuck: *stuck-state* $\mathcal{R}_1$ *s1* $\Longrightarrow$ *stuck-state* $\mathcal{R}_2$ *s2* $\Longrightarrow$ *match i s1 s2* $\Longrightarrow$
match-bisim $\mathcal{R}_1$ $\mathcal{R}_2$ *match order* $(0, 0)$ *s1 s2* |

bisim-steps: *match i* $s1_0$ $s2_0$ $\Longrightarrow$ ${\mathcal{R}_1}^{**}$ $s1_0$ *s1* $\Longrightarrow$ $(\mathcal{R}_1\ ^{\frown\frown}\ Suc\ m)$ *s1 s1'* $\Longrightarrow$
${\mathcal{R}_2}^{**}$ $s2_0$ *s2* $\Longrightarrow$ $(\mathcal{R}_2\ ^{\frown\frown}\ Suc\ n)$ *s2 s2'* $\Longrightarrow$ *match i' s1' s2'* $\Longrightarrow$
match-bisim $\mathcal{R}_1$ $\mathcal{R}_2$ *match order* (m, n) *s1 s2*

theorem *lift-strong-simulation-to-bisimulation*:
fixes
step1 $:: 's1 \Rightarrow 's1 \Rightarrow bool$ **and**
step2 $:: 's2 \Rightarrow 's2 \Rightarrow bool$ **and**
match $:: 'i \Rightarrow 's1 \Rightarrow 's2 \Rightarrow bool$ **and**
order $:: 'i \Rightarrow 'i \Rightarrow bool$
assumes
matching-states-agree-on-stuck:
$\forall\, i\ s1\ s2.\ match\ i\ s1\ s2 \longrightarrow stuck\text{-}state\ step1\ s1 \longleftrightarrow stuck\text{-}state\ step2\ s2$ **and**
well-founded-order: *wfp order* **and**
sim: *simulation step1 step2 match order*
obtains
MATCH $:: nat \times nat \Rightarrow 's1 \Rightarrow 's2 \Rightarrow bool$ **and**
ORDER $:: nat \times nat \Rightarrow nat \times nat \Rightarrow bool$
where
$\bigwedge i\ s1\ s2.\ match\ i\ s1\ s2 \Longrightarrow (\exists\, j.\ MATCH\ j\ s1\ s2)$
$\bigwedge j\ s1\ s2.\ MATCH\ j\ s1\ s2 \Longrightarrow$
$(\exists\, i.\ stuck\text{-}state\ step1\ s1 \wedge stuck\text{-}state\ step2\ s2 \wedge match\ i\ s1\ s2) \vee$
$(\exists\, i\ s1'\ s2'.\ step1^{++}\ s1\ s1' \wedge step2^{++}\ s2\ s2' \wedge match\ i\ s1'\ s2')$ **and**
wfp ORDER **and**
right-unique step1 $\Longrightarrow$ *simulation step1 step2* $(\lambda i\ s1\ s2.\ MATCH\ i\ s1\ s2)$ *ORDER* **and**
right-unique step2 $\Longrightarrow$ *simulation step2 step1* $(\lambda i\ s2\ s1.\ MATCH\ i\ s1\ s2)$ *ORDER*
⟨*proof*⟩

end

definition *safe-state* **where**
safe-state $\mathcal{R}\ \mathcal{F}\ s \longleftrightarrow (\forall\, s'.\ \mathcal{R}^{**}\ s\ s' \longrightarrow stuck\text{-}state\ \mathcal{R}\ s' \longrightarrow \mathcal{F}\ s')$

lemma *step-preserves-safe-state*:

$\mathcal{R}\ s\ s' \Longrightarrow$ *safe-state* $\mathcal{R}\ \mathcal{F}\ s \Longrightarrow$ *safe-state* $\mathcal{R}\ \mathcal{F}\ s'$
⟨*proof*⟩

lemma *rtranclp-step-preserves-safe-state*:
$\mathcal{R}^{**}\ s\ s' \Longrightarrow$ *safe-state* $\mathcal{R}\ \mathcal{F}\ s \Longrightarrow$ *safe-state* $\mathcal{R}\ \mathcal{F}\ s'$
⟨*proof*⟩

lemma *tranclp-step-preserves-safe-state*:
$\mathcal{R}^{++}\ s\ s' \Longrightarrow$ *safe-state* $\mathcal{R}\ \mathcal{F}\ s \Longrightarrow$ *safe-state* $\mathcal{R}\ \mathcal{F}\ s'$
⟨*proof*⟩

lemma *safe-state-before-step-if-safe-state-after*:
assumes *right-unique* $\mathcal{R}$
shows $\mathcal{R}\ s\ s' \Longrightarrow$ *safe-state* $\mathcal{R}\ \mathcal{F}\ s' \Longrightarrow$ *safe-state* $\mathcal{R}\ \mathcal{F}\ s$
⟨*proof*⟩

lemma *safe-state-before-rtranclp-step-if-safe-state-after*:
assumes *right-unique* $\mathcal{R}$
shows $\mathcal{R}^{**}\ s\ s' \Longrightarrow$ *safe-state* $\mathcal{R}\ \mathcal{F}\ s' \Longrightarrow$ *safe-state* $\mathcal{R}\ \mathcal{F}\ s$
⟨*proof*⟩

lemma *safe-state-before-tranclp-step-if-safe-state-after*:
assumes *right-unique* $\mathcal{R}$
shows $\mathcal{R}^{++}\ s\ s' \Longrightarrow$ *safe-state* $\mathcal{R}\ \mathcal{F}\ s' \Longrightarrow$ *safe-state* $\mathcal{R}\ \mathcal{F}\ s$
⟨*proof*⟩

lemma *safe-state-if-all-states-safe*:
fixes $\mathcal{R}\ \mathcal{F}\ s$
assumes $\bigwedge s.\ \mathcal{F}\ s \vee (\exists\, s'.\ \mathcal{R}\ s\ s')$
shows *safe-state* $\mathcal{R}\ \mathcal{F}\ s$
⟨*proof*⟩

lemma
fixes $\mathcal{R}\ \mathcal{F}\ s$
shows *safe-state* $\mathcal{R}\ \mathcal{F}\ s \Longrightarrow \mathcal{F}\ s \vee (\exists\, s'.\ \mathcal{R}\ s\ s')$
⟨*proof*⟩

lemma *matching-states-agree-on-stuck-if-they-agree-on-final*:
assumes
final1-stuck: $\forall\, s1.$ *final1 s1* $\longrightarrow (\nexists\, s1'.$ *step1 s1 s1'*$)$ **and**
final2-stuck: $\forall\, s2.$ *final2 s2* $\longrightarrow (\nexists\, s2'.$ *step2 s2 s2'*$)$ **and**
matching-states-agree-on-final: $\forall\, i\ s1\ s2.$ *match i s1 s2* $\longrightarrow$ *final1 s1* $\longleftrightarrow$ *final2 s2* **and**
matching-states-are-safe:
$\forall\, i\ s1\ s2.$ *match i s1 s2* $\longrightarrow$ *safe-state step1 final1 s1* $\wedge$ *safe-state step2 final2 s2*
shows $\forall\, i\ s1\ s2.$ *match i s1 s2* $\longrightarrow$ *stuck-state step1 s1* $\longleftrightarrow$ *stuck-state step2 s2*
⟨*proof*⟩

locale *wellbehaved-transition-system* =
fixes $\mathcal{R}$ :: *'s* ⇒ *'s* ⇒ *bool* **and** $\mathcal{F}$ :: *'s* ⇒ *bool* **and** $\mathcal{S}$ :: *'s* ⇒ *bool*
assumes
determ: *right-unique* $\mathcal{R}$ **and**
stuck-if-final: $\bigwedge x.\ \mathcal{F}\ x \Longrightarrow$ *stuck-state* $\mathcal{R}\ x$ **and**
safe-if-invar: $\bigwedge x.\ \mathcal{S}\ x \Longrightarrow$ *safe-state* $\mathcal{R}\ \mathcal{F}\ x$

lemma (**in** *wellbehaved-transition-system*) *final-iff-stuck-if-invar*:
fixes *x*
assumes $\mathcal{S}\ x$
shows $\mathcal{F}\ x \longleftrightarrow$ *stuck-state* $\mathcal{R}\ x$
⟨*proof*⟩

lemma *wellbehaved-transition-systems-agree-on-final-iff-agree-on-stuck*:
fixes
$\mathcal{R}_a$ :: *'a* ⇒ *'a* ⇒ *bool* **and** $\mathcal{F}_a$ :: *'a* ⇒ *bool* **and**
$\mathcal{R}_b$ :: *'b* ⇒ *'b* ⇒ *bool* **and** $\mathcal{F}_b$ :: *'b* ⇒ *bool* **and**
$\mathcal{M}$ :: *'i* ⇒ *'a* ⇒ *'b* ⇒ *bool*
assumes
wellbehaved-transition-system $\mathcal{R}_a\ \mathcal{F}_a$ $(\lambda a.\ \exists\, i\ b.\ \mathcal{M}\ i\ a\ b)$ **and**
wellbehaved-transition-system $\mathcal{R}_b\ \mathcal{F}_b$ $(\lambda b.\ \exists\, i\ a.\ \mathcal{M}\ i\ a\ b)$ **and**
$\mathcal{M}\ i\ a\ b$
shows $(\mathcal{F}_a\ a \longleftrightarrow \mathcal{F}_b\ b) \longleftrightarrow$ (*stuck-state* $\mathcal{R}_a\ a \longleftrightarrow$ *stuck-state* $\mathcal{R}_b\ b$)
⟨*proof*⟩

corollary *lift-strong-simulation-to-bisimulation'*:
fixes
step1 :: *'s1* ⇒ *'s1* ⇒ *bool* **and**
step2 :: *'s2* ⇒ *'s2* ⇒ *bool* **and**
match :: *'i* ⇒ *'s1* ⇒ *'s2* ⇒ *bool* **and**
order :: *'i* ⇒ *'i* ⇒ *bool*
assumes
right-unique step1 **and**
right-unique step2 **and**
final1-stuck: ∀ *s1*. *final1 s1* ⟶ (∄ *s1'*. *step1 s1 s1'*) **and**
final2-stuck: ∀ *s2*. *final2 s2* ⟶ (∄ *s2'*. *step2 s2 s2'*) **and**
matching-states-agree-on-final:
∀ *i s1 s2*. *match i s1 s2* ⟶ *final1 s1* ⟷ *final2 s2* **and**
matching-states-are-safe:
∀ *i s1 s2*. *match i s1 s2* ⟶ *safe-state step1 final1 s1* ∧ *safe-state step2 final2 s2* **and**
order-well-founded: *wfp order* **and**
sim: *simulation step1 step2 match order*
obtains
MATCH :: *nat* × *nat* ⇒ *'s1* ⇒ *'s2* ⇒ *bool* **and**
ORDER :: *nat* × *nat* ⇒ *nat* × *nat* ⇒ *bool*
where
⋀*i s1 s2*. *match i s1 s2* ⟹ (∃ *j*. *MATCH j s1 s2*)
⋀*j s1 s2*. *MATCH j s1 s2* ⟹ *final1 s1* ⟷ *final2 s2* **and**

$\bigwedge j\ s1\ s2.\ MATCH\ j\ s1\ s2 \Longrightarrow stuck\text{-}state\ step1\ s1 \longleftrightarrow stuck\text{-}state\ step2\ s2$
and
$\bigwedge j\ s1\ s2.\ MATCH\ j\ s1\ s2 \Longrightarrow safe\text{-}state\ step1\ final1\ s1 \wedge safe\text{-}state\ step2\ final2\ s2$ **and**
wfp ORDER **and**
simulation step1 step2 $(\lambda i\ s1\ s2.\ MATCH\ i\ s1\ s2)$ *ORDER* **and**
simulation step2 step1 $(\lambda i\ s2\ s1.\ MATCH\ i\ s1\ s2)$ *ORDER*
⟨*proof*⟩

end

5 Simulations Between Dynamic Executions

theory *Simulation*
imports
Semantics
Inf
Well-founded
Lifting-Simulation-To-Bisimulation
begin

5.1 Backward simulation

locale *backward-simulation* =
L1: *semantics step1 final1* +
L2: *semantics step2 final2*
for
step1 :: $'state1 \Rightarrow 'state1 \Rightarrow bool$ **and** *final1* :: $'state1 \Rightarrow bool$ **and**
step2 :: $'state2 \Rightarrow 'state2 \Rightarrow bool$ **and** *final2* :: $'state2 \Rightarrow bool$ +
fixes
match :: $'index \Rightarrow 'state1 \Rightarrow 'state2 \Rightarrow bool$ **and**
order :: $'index \Rightarrow 'index \Rightarrow bool$ (**infix** ‹⊏› *70*)
assumes
wfp-order:
wfp (⊏) **and**
match-final:
match i s1 s2 $\Longrightarrow$ *final2 s2* $\Longrightarrow$ *final1 s1* **and**
simulation:
match i s1 s2 $\Longrightarrow$ *step2 s2 s2'* $\Longrightarrow$
$(\exists i'\ s1'.\ step1^{++}\ s1\ s1' \wedge match\ i'\ s1'\ s2') \vee (\exists i'.\ match\ i'\ s1\ s2' \wedge i' \sqsubset i)$
begin

A simulation is defined between two *semantics* L1 and L2. A *match* predicate expresses that two states from L1 and L2 are equivalent. The *match* predicate is also parameterized with an ordering used to avoid stuttering. The only two assumptions of a backward simulation are that a final state in L2 will also be a final in L1,and that a step in L2 will either represent a

non-empty sequence of steps in L1 or will result in an equivalent state. Stuttering is ruled out by the requirement that the index on the *match* predicate decreases with respect to the well-founded ($\sqsubset$) ordering.

lemma *lift-simulation-plus*:
step2^{++} *s2 s2'* $\Longrightarrow$ *match i1 s1 s2* $\Longrightarrow$
($\exists$ *i2 s1'*. *step1*$^{++}$ *s1 s1'* $\wedge$ *match i2 s1' s2'*) $\vee$
($\exists$ *i2*. *match i2 s1 s2'* $\wedge$ *order*$^{++}$ *i2 i1*)
thm *tranclp-induct*
⟨*proof*⟩

lemma *lift-simulation-eval*:
L2.eval s2 s2' $\Longrightarrow$ *match i1 s1 s2* $\Longrightarrow$ $\exists$ *i2 s1'*. *L1.eval s1 s1'* $\wedge$ *match i2 s1' s2'*
⟨*proof*⟩

lemma *match-inf*:
assumes
match i s1 s2 **and**
inf step2 s2
shows *inf step1 s1*
⟨*proof*⟩

5.1.1 Preservation of behaviour

The main correctness theorem states that, for any two matching programs, any not wrong behaviour of the later is also a behaviour of the former. In other words, if the compiled program does not crash, then its behaviour, whether it terminates or not, is a also a valid behaviour of the source program.

lemma *simulation-behaviour* :
L2.state-behaves s_2 b_2 $\Longrightarrow$ $\neg$*is-wrong* b_2 $\Longrightarrow$ *match i* s_1 s_2 $\Longrightarrow$
$\exists$ b_1 i'. *L1.state-behaves* s_1 b_1 $\wedge$ *rel-behaviour* (*match i'*) b_1 b_2
⟨*proof*⟩

end

5.2 Forward simulation

locale *forward-simulation* =
L1: *semantics step1 final1* +
L2: *semantics step2 final2*
for
step1 :: *'state1* $\Rightarrow$ *'state1* $\Rightarrow$ *bool* **and** *final1* :: *'state1* $\Rightarrow$ *bool* **and**
step2 :: *'state2* $\Rightarrow$ *'state2* $\Rightarrow$ *bool* **and** *final2* :: *'state2* $\Rightarrow$ *bool* +
fixes
match :: *'index* $\Rightarrow$ *'state1* $\Rightarrow$ *'state2* $\Rightarrow$ *bool* **and**
order :: *'index* $\Rightarrow$ *'index* $\Rightarrow$ *bool* (**infix** ‹$\sqsubset$› *70*)
assumes
wfp-order:

wfp ($\sqsubset$) **and**
match-final:
match i s1 s2 $\Longrightarrow$ *final1 s1* $\Longrightarrow$ *final2 s2* **and**
simulation:
match i s1 s2 $\Longrightarrow$ *step1 s1 s1'* $\Longrightarrow$
$(\exists\, i'\ s2'.\ step2^{++}\ s2\ s2' \wedge match\ i'\ s1'\ s2') \vee (\exists\, i'.\ match\ i'\ s1'\ s2 \wedge i' \sqsubset i)$
begin

lemma *lift-simulation-plus*:
$step1^{++}\ s1\ s1' \Longrightarrow match\ i\ s1\ s2 \Longrightarrow$
$(\exists\, i'\ s2'.\ step2^{++}\ s2\ s2' \wedge match\ i'\ s1'\ s2') \vee$
$(\exists\, i'.\ match\ i'\ s1'\ s2 \wedge order^{++}\ i'\ i)$
⟨*proof*⟩

lemma *lift-simulation-eval*:
$L1.eval\ s1\ s1' \Longrightarrow match\ i\ s1\ s2 \Longrightarrow \exists\, i'\ s2'.\ L2.eval\ s2\ s2' \wedge match\ i'\ s1'\ s2'$
⟨*proof*⟩

lemma *match-inf*:
assumes *match i s1 s2* **and** *inf step1 s1*
shows *inf step2 s2*
⟨*proof*⟩

5.2.1 Preservation of behaviour

lemma *simulation-behaviour* :
$L1.state\text{-}behaves\ s1\ b1 \Longrightarrow \neg\ is\text{-}wrong\ b1 \Longrightarrow match\ i\ s1\ s2 \Longrightarrow$
$\exists\, b2\ i'.\ L2.state\text{-}behaves\ s2\ b2 \wedge rel\text{-}behaviour\ (match\ i')\ b1\ b2$
⟨*proof*⟩

5.2.2 Forward to backward

lemma *state-behaves-forward-to-backward*:
assumes
match-s1-s2: *match i s1 s2* **and**
safe-s1: *L1.safe s1* **and**
behaves-s2: *L2.state-behaves s2 b2* **and**
right-unique2: *right-unique step2*
shows $\exists\, b1\ i.\ L1.state\text{-}behaves\ s1\ b1 \wedge rel\text{-}behaviour\ (match\ i)\ b1\ b2$
⟨*proof*⟩

end

5.3 Bisimulation

locale *bisimulation* =
forward-simulation step1 final1 step2 final2 match $order_f$ +
backward-simulation step1 final1 step2 final2 match $order_b$
for

step1 :: *'state1* ⇒ *'state1* ⇒ *bool* **and** *final1* :: *'state1* ⇒ *bool* **and**
step2 :: *'state2* ⇒ *'state2* ⇒ *bool* **and** *final2* :: *'state2* ⇒ *bool* **and**
match :: *'index* ⇒ *'state1* ⇒ *'state2* ⇒ *bool* **and**
$order_f$ :: *'index* ⇒ *'index* ⇒ *bool* **and**
$order_b$ :: *'index* ⇒ *'index* ⇒ *bool*

lemma (**in** *bisimulation*) *agree-on-final*:
assumes *match i s1 s2*
shows *final1 s1* ⟷ *final2 s2*
⟨*proof*⟩

lemma *obtains-bisimulation-from-forward-simulation*:
fixes
step1 :: *'state1* ⇒ *'state1* ⇒ *bool* **and** *final1* :: *'state1* ⇒ *bool* **and**
step2 :: *'state2* ⇒ *'state2* ⇒ *bool* **and** *final2* :: *'state2* ⇒ *bool* **and**
match :: *'index* ⇒ *'state1* ⇒ *'state2* ⇒ *bool* **and**
lt :: *'index* ⇒ *'index* ⇒ *bool*
assumes *right-unique step1* **and** *right-unique step2* **and**
final1-stuck: ∀ *s1*. *final1 s1* ⟶ (∄ *s1'*. *step1 s1 s1'*) **and**
final2-stuck: ∀ *s2*. *final2 s2* ⟶ (∄ *s2'*. *step2 s2 s2'*) **and**
matching-states-agree-on-final: ∀ *i s1 s2*. *match i s1 s2* ⟶ *final1 s1* ⟷ *final2 s2* **and**
matching-states-are-safe:
∀ *i s1 s2*. *match i s1 s2* ⟶ *safe-state step1 final1 s1* ∧ *safe-state step2 final2 s2* **and**
wfP lt **and**
fsim: ∀ *i s1 s2 s1'*. *match i s1 s2* ⟶ *step1 s1 s1'* ⟶
(∃ *i' s2'*. $step2^{++}$ *s2 s2'* ∧ *match i' s1' s2'*) ∨ (∃ *i'*. *match i' s1' s2* ∧ *lt i' i*)
obtains
MATCH :: *nat* × *nat* ⇒ *'state1* ⇒ *'state2* ⇒ *bool* **and**
ORDER :: *nat* × *nat* ⇒ *nat* × *nat* ⇒ *bool*
where
bisimulation step1 final1 step2 final2 MATCH ORDER ORDER
⟨*proof*⟩

corollary *ex-bisimulation-from-forward-simulation*:
fixes
step1 :: *'state1* ⇒ *'state1* ⇒ *bool* **and** *final1* :: *'state1* ⇒ *bool* **and**
step2 :: *'state2* ⇒ *'state2* ⇒ *bool* **and** *final2* :: *'state2* ⇒ *bool* **and**
match :: *'index* ⇒ *'state1* ⇒ *'state2* ⇒ *bool* **and**
lt :: *'index* ⇒ *'index* ⇒ *bool*
assumes *right-unique step1* **and** *right-unique step2* **and**
final1-stuck: ∀ *s1*. *final1 s1* ⟶ (∄ *s1'*. *step1 s1 s1'*) **and**
final2-stuck: ∀ *s2*. *final2 s2* ⟶ (∄ *s2'*. *step2 s2 s2'*) **and**
matching-states-agree-on-final: ∀ *i s1 s2*. *match i s1 s2* ⟶ *final1 s1* ⟷ *final2 s2* **and**
matching-states-are-safe:
∀ *i s1 s2*. *match i s1 s2* ⟶ *safe-state step1 final1 s1* ∧ *safe-state step2 final2 s2* **and**

wfP lt **and**
fsim: $\forall\, i\ s1\ s2\ s1'.\ match\ i\ s1\ s2 \longrightarrow step1\ s1\ s1' \longrightarrow$
$(\exists\, i'\ s2'.\ step2^{++}\ s2\ s2' \wedge match\ i'\ s1'\ s2') \vee (\exists\, i'.\ match\ i'\ s1'\ s2 \wedge lt\ i'\ i)$
shows $\exists\, (MATCH :: nat \times nat \Rightarrow 'state1 \Rightarrow 'state2 \Rightarrow bool)\ ORDER_f\ ORDER_b.$
bisimulation step1 final1 step2 final2 MATCH $ORDER_f$ $ORDER_b$
⟨*proof*⟩

lemma *obtains-bisimulation-from-backward-simulation*:
fixes
step1 :: *'state1* ⇒ *'state1* ⇒ *bool* **and** *final1* :: *'state1* ⇒ *bool* **and**
step2 :: *'state2* ⇒ *'state2* ⇒ *bool* **and** *final2* :: *'state2* ⇒ *bool* **and**
match :: *'index* ⇒ *'state1* ⇒ *'state2* ⇒ *bool* **and**
lt :: *'index* ⇒ *'index* ⇒ *bool*
assumes *right-unique step1* **and** *right-unique step2* **and**
final1-stuck: $\forall\, s1.\ final1\ s1 \longrightarrow (\nexists\, s1'.\ step1\ s1\ s1')$ **and**
final2-stuck: $\forall\, s2.\ final2\ s2 \longrightarrow (\nexists\, s2'.\ step2\ s2\ s2')$ **and**
matching-states-agree-on-final: $\forall\, i\ s1\ s2.\ match\ i\ s1\ s2 \longrightarrow final1\ s1 \longleftrightarrow final2\ s2$ **and**
matching-states-are-safe:
$\forall\, i\ s1\ s2.\ match\ i\ s1\ s2 \longrightarrow safe\text{-}state\ step1\ final1\ s1 \wedge safe\text{-}state\ step2\ final2\ s2$ **and**
wfP lt **and**
bsim: $\forall\, i\ s1\ s2\ s2'.\ match\ i\ s1\ s2 \longrightarrow step2\ s2\ s2' \longrightarrow$
$(\exists\, i'\ s1'.\ step1^{++}\ s1\ s1' \wedge match\ i'\ s1'\ s2') \vee (\exists\, i'.\ match\ i'\ s1\ s2' \wedge lt\ i'\ i)$
obtains
MATCH :: *nat* × *nat* ⇒ *'state1* ⇒ *'state2* ⇒ *bool* **and**
ORDER :: *nat* × *nat* ⇒ *nat* × *nat* ⇒ *bool*
where
bisimulation step1 final1 step2 final2 MATCH ORDER ORDER
⟨*proof*⟩

corollary *ex-bisimulation-from-backward-simulation*:
fixes
step1 :: *'state1* ⇒ *'state1* ⇒ *bool* **and** *final1* :: *'state1* ⇒ *bool* **and**
step2 :: *'state2* ⇒ *'state2* ⇒ *bool* **and** *final2* :: *'state2* ⇒ *bool* **and**
match :: *'index* ⇒ *'state1* ⇒ *'state2* ⇒ *bool* **and**
lt :: *'index* ⇒ *'index* ⇒ *bool*
assumes *right-unique step1* **and** *right-unique step2* **and**
final1-stuck: $\forall\, s1.\ final1\ s1 \longrightarrow (\nexists\, s1'.\ step1\ s1\ s1')$ **and**
final2-stuck: $\forall\, s2.\ final2\ s2 \longrightarrow (\nexists\, s2'.\ step2\ s2\ s2')$ **and**
matching-states-agree-on-final: $\forall\, i\ s1\ s2.\ match\ i\ s1\ s2 \longrightarrow final1\ s1 \longleftrightarrow final2\ s2$ **and**
matching-states-are-safe:
$\forall\, i\ s1\ s2.\ match\ i\ s1\ s2 \longrightarrow safe\text{-}state\ step1\ final1\ s1 \wedge safe\text{-}state\ step2\ final2\ s2$ **and**
wfP lt **and**
bsim: $\forall\, i\ s1\ s2\ s2'.\ match\ i\ s1\ s2 \longrightarrow step2\ s2\ s2' \longrightarrow$
$(\exists\, i'\ s1'.\ step1^{++}\ s1\ s1' \wedge match\ i'\ s1'\ s2') \vee (\exists\, i'.\ match\ i'\ s1\ s2' \wedge lt\ i'\ i)$
shows $\exists\, (MATCH :: nat \times nat \Rightarrow 'state1 \Rightarrow 'state2 \Rightarrow bool)\ ORDER_f\ ORDER_b.$

bisimulation step1 final1 step2 final2 MATCH ORDER$_f$ ORDER$_b$
⟨*proof*⟩

5.4 Composition of simulations

definition *rel-comp* ::
(*'a* ⇒ *'b* ⇒ *'c* ⇒ *bool*) ⇒ (*'d* ⇒ *'c* ⇒ *'e* ⇒ *bool*) ⇒ (*'a* × *'d*) ⇒ *'b* ⇒ *'e* ⇒ *bool*
where
rel-comp r1 r2 i ≡ (*r1* (*fst i*) *OO r2* (*snd i*))

5.4.1 Composition of backward simulations

lemma *backward-simulation-composition*:
assumes
backward-simulation step1 final1 step2 final2 match1 order1
backward-simulation step2 final2 step3 final3 match2 order2
shows
backward-simulation step1 final1 step3 final3
(*rel-comp match1 match2*) (*lex-prodp order1*$^{++}$ *order2*)
⟨*proof*⟩

context
fixes *r* :: *'i* ⇒ *'a* ⇒ *'a* ⇒ *bool*
begin

fun *rel-comp-pow* **where**
rel-comp-pow [] *x y* = *False* |
rel-comp-pow [*i*] *x y* = *r i x y* |
rel-comp-pow (*i* # *is*) *x z* = (∃ *y*. *r i x y* ∧ *rel-comp-pow is y z*)

end

lemma *backward-simulation-pow*:
assumes
backward-simulation step final step final match order
shows
backward-simulation step final step final (*rel-comp-pow match*) (*lexp order*$^{++}$)
⟨*proof*⟩

5.4.2 Composition of forward simulations

lemma *forward-simulation-composition*:
assumes
forward-simulation step1 final1 step2 final2 match1 order1
forward-simulation step2 final2 step3 final3 match2 order2
defines *ORDER* ≡ λ*i i'*. *lex-prodp order2*$^{++}$ *order1* (*prod.swap i*) (*prod.swap i'*)
shows *forward-simulation step1 final1 step3 final3* (*rel-comp match1 match2*) *ORDER*
⟨*proof*⟩

5.4.3 Composition of bisimulations

lemma *bisimulation-composition*:
fixes
step1 :: *'s1* ⇒ *'s1* ⇒ *bool* **and** *final1* :: *'s1* ⇒ *bool* **and**
step2 :: *'s2* ⇒ *'s2* ⇒ *bool* **and** *final2* :: *'s2* ⇒ *bool* **and**
step3 :: *'s3* ⇒ *'s3* ⇒ *bool* **and** *final3* :: *'s3* ⇒ *bool* **and**
match1 :: *'i* ⇒ *'s1* ⇒ *'s2* ⇒ *bool* **and** $order1_f$ $order1_b$ :: *'i* ⇒ *'i* ⇒ *bool* **and**
match2 :: *'j* ⇒ *'s2* ⇒ *'s3* ⇒ *bool* **and** $order2_f$ $order2_b$ :: *'j* ⇒ *'j* ⇒ *bool*
assumes
bisimulation step1 final1 step2 final2 match1 $order1_f$ $order1_b$
bisimulation step2 final2 step3 final3 match2 $order2_f$ $order2_b$
obtains
$ORDER_f$ :: *'i* × *'j* ⇒ *'i* × *'j* ⇒ *bool* **and**
$ORDER_b$ :: *'i* × *'j* ⇒ *'i* × *'j* ⇒ *bool* **and**
MATCH :: *'i* × *'j* ⇒ *'s1* ⇒ *'s3* ⇒ *bool*
where *bisimulation step1 final1 step3 final3 MATCH* $ORDER_f$ $ORDER_b$
⟨*proof*⟩

5.5 Miscellaneous

definition *lockstep-backward-simulation* **where**
lockstep-backward-simulation step1 step2 match ≡
∀ *s1 s2 s2'*. *match s1 s2* ⟶ *step2 s2 s2'* ⟶ (∃ *s1'*. *step1 s1 s1'* ∧ *match s1' s2'*)

definition *plus-backward-simulation* **where**
plus-backward-simulation step1 step2 match ≡
∀ *s1 s2 s2'*. *match s1 s2* ⟶ *step2 s2 s2'* ⟶ (∃ *s1'*. $step1^{++}$ *s1 s1'* ∧ *match s1' s2'*)

lemma
assumes *lockstep-backward-simulation step1 step2 match*
shows *plus-backward-simulation step1 step2 match*
⟨*proof*⟩

lemma *lockstep-to-plus-backward-simulation*:
fixes
match :: *'state1* ⇒ *'state2* ⇒ *bool* **and**
step1 :: *'state1* ⇒ *'state1* ⇒ *bool* **and**
step2 :: *'state2* ⇒ *'state2* ⇒ *bool*
assumes
lockstep-simulation: ⋀*s1 s2 s2'*. *match s1 s2* ⟹ *step2 s2 s2'* ⟹ (∃ *s1'*. *step1 s1 s1'* ∧ *match s1' s2'*) **and**
match: *match s1 s2* **and**
step: *step2 s2 s2'*
shows ∃ *s1'*. $step1^{++}$ *s1 s1'* ∧ *match s1' s2'*
⟨*proof*⟩

lemma *lockstep-to-option-backward-simulation*:

fixes
match :: *'state1* ⇒ *'state2* ⇒ *bool* **and**
step1 :: *'state1* ⇒ *'state1* ⇒ *bool* **and**
step2 :: *'state2* ⇒ *'state2* ⇒ *bool* **and**
measure :: *'state2* ⇒ *nat*
assumes
lockstep-simulation: ⋀*s1 s2 s2'*. *match s1 s2* ⟹ *step2 s2 s2'* ⟹ (∃ *s1'*. *step1 s1 s1'* ∧ *match s1' s2'*) **and**
match: *match s1 s2* **and**
step: *step2 s2 s2'*
shows (∃ *s1'*. *step1 s1 s1'* ∧ *match s1' s2'*) ∨ *match s1 s2'* ∧ *measure s2'* < *measure s2*
⟨*proof*⟩

lemma *plus-to-star-backward-simulation*:
fixes
match :: *'state1* ⇒ *'state2* ⇒ *bool* **and**
step1 :: *'state1* ⇒ *'state1* ⇒ *bool* **and**
step2 :: *'state2* ⇒ *'state2* ⇒ *bool* **and**
measure :: *'state2* ⇒ *nat*
assumes
star-simulation: ⋀*s1 s2 s2'*. *match s1 s2* ⟹ *step2 s2 s2'* ⟹ (∃ *s1'*. *step1*$^{++}$ *s1 s1'* ∧ *match s1' s2'*) **and**
match: *match s1 s2* **and**
step: *step2 s2 s2'*
shows (∃ *s1'*. *step1*$^{++}$ *s1 s1'* ∧ *match s1' s2'*) ∨ *match s1 s2'* ∧ *measure s2'* < *measure s2*
⟨*proof*⟩

lemma *lockstep-to-plus-forward-simulation*:
fixes
match :: *'state1* ⇒ *'state2* ⇒ *bool* **and**
step1 :: *'state1* ⇒ *'state1* ⇒ *bool* **and**
step2 :: *'state2* ⇒ *'state2* ⇒ *bool*
assumes
lockstep-simulation: ⋀*s1 s2 s2'*. *match s1 s2* ⟹ *step1 s1 s1'* ⟹ (∃ *s2'*. *step2 s2 s2'* ∧ *match s1' s2'*) **and**
match: *match s1 s2* **and**
step: *step1 s1 s1'*
shows ∃ *s2'*. *step2*$^{++}$ *s2 s2'* ∧ *match s1' s2'*
⟨*proof*⟩

end

6 Compiler Between Static Representations

theory *Compiler*
imports *Language Simulation*
begin

definition *option-comp* :: (*'a* ⇒ *'b option*) ⇒ (*'c* ⇒ *'a option*) ⇒ *'c* ⇒ *'b option* (**infix** ‹⇐› *50*) **where**
  (*f* ⇐ *g*) *x* ≡ *Option.bind* (*g x*) *f*

context
  fixes *f* :: (*'a* ⇒ *'a option*)
begin

fun *option-comp-pow* :: *nat* ⇒ *'a* ⇒ *'a option* **where**
  option-comp-pow 0 = (λ-. *None*) |
  option-comp-pow (*Suc 0*) = *f* |
  option-comp-pow (*Suc n*) = (*option-comp-pow n* ⇐ *f*)

end

locale *compiler* =
  L1: *language step1 final1 load1* +
  L2: *language step2 final2 load2* +
  backward-simulation step1 final1 step2 final2 match order
  for
    step1 :: *'state1* ⇒ *'state1* ⇒ *bool* **and** *final1* **and** *load1* :: *'prog1* ⇒ *'state1* ⇒ *bool* **and**
    step2 :: *'state2* ⇒ *'state2* ⇒ *bool* **and** *final2* **and** *load2* :: *'prog2* ⇒ *'state2* ⇒ *bool* **and**
    match **and**
    order :: *'index* ⇒ *'index* ⇒ *bool* +
  fixes
    compile :: *'prog1* ⇒ *'prog2 option*
  assumes
    compile-load:
      compile p1 = *Some p2* ⟹ *load2 p2 s2* ⟹ ∃ *s1 i*. *load1 p1 s1* ∧ *match i s1 s2*
begin

The *compiler* locale relates two languages, L1 and L2, by a backward simulation and provides a *compile* partial function from a concrete program in L1 to a concrete program in L2. The only assumption is that a successful compilation results in a program which, when loaded, is equivalent to the loaded initial program.

6.1 Preservation of behaviour

corollary *behaviour-preservation*:
  assumes
    compiles: *compile p1* = *Some p2* **and**
    behaves: *L2.prog-behaves p2 b2* **and**
    not-wrong: ¬ *is-wrong b2*
  shows ∃ *b1 i*. *L1.prog-behaves p1 b1* ∧ *rel-behaviour* (*match i*) *b1 b2*

⟨*proof*⟩

end

6.2 Composition of compilers

lemma *compiler-composition*:
assumes
compiler step1 final1 load1 step2 final2 load2 match1 order1 compile1 **and**
compiler step2 final2 load2 step3 final3 load3 match2 order2 compile2
shows *compiler step1 final1 load1 step3 final3 load3*
(*rel-comp match1 match2*) (*lex-prodp order1*$^{++}$ *order2*) (*compile2* $\Leftarrow$ *compile1*)
⟨*proof*⟩

lemma *compiler-composition-pow*:
assumes
compiler step final load step final load match order compile
shows *compiler step final load step final load*
(*rel-comp-pow match*) (*lexp order*$^{++}$) (*option-comp-pow compile n*)
⟨*proof*⟩

end

7 Fixpoint of Converging Program Transformations

theory *Fixpoint*
imports *Compiler*
begin

context
fixes
$m :: 'a \Rightarrow nat$ **and**
$f :: 'a \Rightarrow 'a\ option$
begin

function *fixpoint* $:: 'a \Rightarrow 'a\ option$ **where**
fixpoint x = (
case f x of
None $\Rightarrow$ *None* |
Some x' $\Rightarrow$ *if m x'* $<$ *m x then fixpoint x' else Some x'*
)
⟨*proof*⟩
termination
⟨*proof*⟩

end

lemma *fixpoint-to-comp-pow*:
fixpoint m f x = y ⟹ ∃ n. option-comp-pow f n x = y
⟨*proof*⟩

lemma *fixpoint-eq-comp-pow*:
∃ n. fixpoint m f x = option-comp-pow f n x
⟨*proof*⟩

lemma *compiler-composition-fixpoint*:
assumes
compiler step final load step final load match order compile
shows *compiler step final load step final load*
(*rel-comp-pow match*) (*lexp order*$^{++}$) (*fixpoint m compile*)
⟨*proof*⟩

end

References

[1] M. Desharnais and S. Brunthaler. A generic framework for verified compilers using isabelle/hols locales. *31 ème Journées Francophones des Langages Applicatifs*, page 198, 2020.