

A Generic Framework for Verified Compilers

Martin Desharnais

May 14, 2024

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	Well-foundedness of Relations Defined as Predicate Functions	2
1.1	Unit	2
1.2	Lexicographic product	2
1.3	Lexicographic list	3
2	Infinitely Transitive Closure	4
3	The Dynamic Representation of a Language	5
3.1	Behaviour of a dynamic execution	6
3.2	Safe states	6
4	The Static Representation of a Language	7
4.1	Program behaviour	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 backward simulations	16

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

7 Fixpoint of Converging Program Transformations **19**

```
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

1 Well-foundedness of Relations Defined as Predicate Functions

```
theory Well-founded
  imports Main
begin
```

```
locale well-founded =
  fixes R :: 'a ⇒ 'a ⇒ bool (infix □ 70)
  assumes
    wf: wfP (□)
begin
```

```
lemmas induct = wfP-induct-rule[OF wf]
```

end

1.1 Unit

```
lemma wfP-unit: wfP (λ() (). False)
  ⟨proof⟩
```

```
interpretation well-founded λ() (). False
  ⟨proof⟩
```

1.2 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

lemma *lex-prodp-well-founded*:

assumes

well-founded r1 **and**

well-founded r2

shows *well-founded* (*lex-prodp* r1 r2)

⟨proof⟩

1.3 Lexicographic list

context

fixes *order* :: 'a ⇒ 'a ⇒ bool

begin

inductive *lexp* :: 'a list ⇒ 'a list ⇒ bool **where**

lexp-head: *order* x y ⇒ length xs = length ys ⇒ *lexp* (x # xs) (y # ys) |

lexp-tail: *lexp* xs ys ⇒ *lexp* (x # xs) (x # ys)

end

lemma *lexp-prepend*: *lexp* order ys zs ⇒ *lexp* order (xs @ ys) (xs @ zs)

⟨proof⟩

lemma *lexp-lex*: *lexp* order xs ys ⟷ (xs, ys) ∈ *lex* {(x, y). *order* x y}

⟨proof⟩

lemma *lex-list-wfP*: *wfP* order ⇒ *wfP* (*lexp* order)

⟨proof⟩

lemma *lex-list-well-founded*:

assumes *well-founded* order

shows *well-founded* (*lexp* order)

<proof>

end

2 Infinitely Transitive Closure

theory *Inf*

imports *Well-founded*

begin

coinductive *inf* :: ('a ⇒ 'a ⇒ bool) ⇒ 'a ⇒ bool **for** *r* **where**

inf-step: $r\ x\ y \Longrightarrow \text{inf}\ r\ y \Longrightarrow \text{inf}\ r\ x$

coinductive *inf-wf* :: ('a ⇒ 'a ⇒ bool) ⇒ ('b ⇒ 'b ⇒ bool) ⇒ 'b ⇒ 'a ⇒ bool **for** *r* **order** **where**

inf-wf: $\text{order}\ n\ m \Longrightarrow \text{inf-wf}\ r\ \text{order}\ n\ x \Longrightarrow \text{inf-wf}\ r\ \text{order}\ m\ x \mid$

inf-wf-step: $r^{++}\ x\ y \Longrightarrow \text{inf-wf}\ r\ \text{order}\ n\ y \Longrightarrow \text{inf-wf}\ r\ \text{order}\ m\ x$

lemma *inf-wf-to-step-inf-wf*:

assumes *well-founded order*

shows $\text{inf-wf}\ r\ \text{order}\ n\ x \Longrightarrow \exists y\ m. r\ x\ y \wedge \text{inf-wf}\ r\ \text{order}\ m\ y$

<proof>

lemma *inf-wf-to-inf*:

assumes *well-founded order*

shows $\text{inf-wf}\ r\ \text{order}\ n\ x \Longrightarrow \text{inf}\ r\ x$

<proof>

lemma *step-inf*:

assumes *right-unique r*

shows $r\ x\ y \Longrightarrow \text{inf}\ r\ x \Longrightarrow \text{inf}\ r\ y$

<proof>

lemma *star-inf*:

assumes *right-unique r*

shows $r^{**}\ x\ y \Longrightarrow \text{inf}\ r\ x \Longrightarrow \text{inf}\ r\ y$

<proof>

end

theory *Transfer-Extras*

imports *Main*

begin

lemma *rtranclp-complete-run-right-unique*:

fixes $R :: 'a \Rightarrow 'a \Rightarrow \text{bool}$ **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 $R :: 'a \Rightarrow 'a \Rightarrow \text{bool}$ **and** $x\ y\ z :: 'a$
assumes *right-unique* R
shows $R^{++}\ x\ y \Longrightarrow (\nexists w. R\ y\ w) \Longrightarrow R^{++}\ x\ z \Longrightarrow (\nexists w. R\ z\ w) \Longrightarrow y = z$
 $\langle \text{proof} \rangle$

end

3 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* $:: ('a \Rightarrow 'a \Rightarrow \text{bool}) \Rightarrow 'a \Rightarrow \text{bool}$ **where**
finished $r\ x = (\nexists y. r\ x\ y)$

lemma *finished-star*:
assumes *finished* $r\ x$
shows $r^{**}\ x\ y \Longrightarrow x = y$
 $\langle \text{proof} \rangle$

locale *semantics* =
fixes
 $\text{step} :: 'state \Rightarrow 'state \Rightarrow \text{bool}$ (**infix** \rightarrow 50) **and**
 $\text{final} :: 'state \Rightarrow \text{bool}$
assumes
 $\text{final-finished}: \text{final}\ s \Longrightarrow \text{finished}\ \text{step}\ s$
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*:
 $\text{step}\ s\ s' \Longrightarrow \neg \text{finished}\ \text{step}\ s$
 $\langle \text{proof} \rangle$

abbreviation *eval* $:: 'state \Rightarrow 'state \Rightarrow \text{bool}$ (**infix** \rightarrow^* 50) **where**
 $\text{eval} \equiv \text{step}^{**}$

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

notation
 $\text{inf-step}\ ('(\rightarrow^\infty) []\ 50)$ **and**
 $\text{inf-step}\ (- \rightarrow^\infty [55]\ 50)$

lemma *inf-not-finished*: $s \rightarrow^\infty \implies \neg \text{finished step } s$
 ⟨proof⟩

lemma *eval-deterministic*:

assumes

deterministic: $\bigwedge x y z. \text{step } x y \implies \text{step } x z \implies y = z$ **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*: $(\exists s'. s \rightarrow^* s' \wedge \text{finished step } s') \vee s \rightarrow^\infty$
 ⟨proof⟩

3.1 Behaviour of a dynamic execution

inductive *state-behaves* :: *'state* \Rightarrow *'state behaviour* \Rightarrow *bool* (**infix** \downarrow 50) **where**

state-terminates:

$s1 \rightarrow^* s2 \implies \text{finished step } s2 \implies \text{final } s2 \implies s1 \downarrow (\text{Terminates } s2) \mid$

state-diverges:

$s1 \rightarrow^\infty \implies s1 \downarrow \text{Diverges} \mid$

state-goes-wrong:

$s1 \rightarrow^* s2 \implies \text{finished step } s2 \implies \neg \text{final } s2 \implies s1 \downarrow (\text{Goes-wrong } s2)$

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* (\downarrow)

⟨proof⟩

3.2 Safe states

definition *safe* **where**

$\text{safe } s \iff (\forall s'. \text{step}^{**} s s' \longrightarrow \text{final } s' \vee (\exists s''. \text{step } s' s''))$

lemma *final-safeI*: $\text{final } s \implies \text{safe } s$

⟨proof⟩

lemma *step-safe*: $\text{step } s s' \implies \text{safe } s \implies \text{safe } s'$

⟨proof⟩

lemma *steps-safe*: $\text{step}^{**} s s' \implies \text{safe } s \implies \text{safe } s'$

⟨proof⟩

```

lemma safe-state-behaves-not-wrong:
  assumes safe s and s ↓ b
  shows  $\neg$  is-wrong b
  <proof>

end

end

```

4 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*.

4.1 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
theory Lifting-Simulation-To-Bisimulation

```

```

imports
  Main
  VeriComp.Well-founded
begin

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

definition simulation ::
  ('a ⇒ 'a ⇒ bool) ⇒ ('b ⇒ 'b ⇒ bool) ⇒ ('c ⇒ 'a ⇒ 'b ⇒ bool) ⇒ ('c ⇒ 'c ⇒
  bool) ⇒ 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 ⇒ 's1 ⇒ bool and
  step2 :: 's2 ⇒ 's2 ⇒ bool and
  match :: 'i ⇒ 's1 ⇒ 's2 ⇒ bool and
  order :: 'i ⇒ 'i ⇒ 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  $\implies$  step1 s1 s1'  $\implies$ 
   $\exists m$  s1'' n s2'' i'. (step1  $\rightsquigarrow$  m) s1' s1''  $\wedge$  (step2  $\rightsquigarrow$  Suc n) s2 s2''  $\wedge$  match i'
  s1'' s2''
  <proof>

context begin

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

  bisim-steps: match i s10 s20  $\implies$   $\mathcal{R}_1^{**}$  s10 s1  $\implies$  ( $\mathcal{R}_1$   $\rightsquigarrow$  Suc m) s1 s1'  $\implies$ 
   $\mathcal{R}_2^{**}$  s20 s2  $\implies$  ( $\mathcal{R}_2$   $\rightsquigarrow$  Suc n) s2 s2'  $\implies$  match i' s1' s2'  $\implies$ 
  match-bisim  $\mathcal{R}_1$   $\mathcal{R}_2$  match order (m, n) s1 s2

theorem lift-strong-simulation-to-bisimulation:
fixes
  step1 :: 's1 ⇒ 's1 ⇒ 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 $step2 \Longrightarrow simulation\ step2\ step1\ (\lambda i\ s2\ s1. MATCH\ i\ s1\ s2)$
 $ORDER$
 $\langle proof \rangle$

end

definition *safe-state* **where**

$safe\text{-}state\ \mathcal{R}\ \mathcal{F}\ s \longleftrightarrow (\forall s'. \mathcal{R}^{**}\ s\ s' \longrightarrow \mathcal{F}\ s' \vee (\exists s''. \mathcal{R}\ s'\ s''))$

lemma *step-preserves-safe-state:*

$\mathcal{R}\ s\ s' \Longrightarrow safe\text{-}state\ \mathcal{R}\ \mathcal{F}\ s \Longrightarrow safe\text{-}state\ \mathcal{R}\ \mathcal{F}\ s'$
 $\langle proof \rangle$

lemma *rtranclp-step-preserves-safe-state:*

$\mathcal{R}^{**}\ s\ s' \Longrightarrow safe\text{-}state\ \mathcal{R}\ \mathcal{F}\ s \Longrightarrow safe\text{-}state\ \mathcal{R}\ \mathcal{F}\ s'$
 $\langle proof \rangle$

lemma *tranclp-step-preserves-safe-state:*

$\mathcal{R}^{++}\ s\ s' \Longrightarrow safe\text{-}state\ \mathcal{R}\ \mathcal{F}\ s \Longrightarrow safe\text{-}state\ \mathcal{R}\ \mathcal{F}\ s'$
 $\langle proof \rangle$

lemma *safe-state-before-step-if-safe-state-after:*

assumes *right-unique* \mathcal{R}
shows $\mathcal{R}\ s\ s' \Longrightarrow safe\text{-}state\ \mathcal{R}\ \mathcal{F}\ s' \Longrightarrow safe\text{-}state\ \mathcal{R}\ \mathcal{F}\ s$
 $\langle proof \rangle$

lemma *safe-state-before-rtranclp-step-if-safe-state-after:*

assumes *right-unique* \mathcal{R}
shows $\mathcal{R}^{**}\ s\ s' \Longrightarrow safe\text{-}state\ \mathcal{R}\ \mathcal{F}\ s' \Longrightarrow safe\text{-}state\ \mathcal{R}\ \mathcal{F}\ s$
 $\langle proof \rangle$

lemma *safe-state-before-tranclp-step-if-safe-state-after*:
assumes *right-unique* \mathcal{R}
shows $\mathcal{R}^{++} s s' \implies \text{safe-state } \mathcal{R} \mathcal{F} s' \implies \text{safe-state } \mathcal{R} \mathcal{F} s$
<proof>

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

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

corollary *lift-strong-simulation-to-bisimulation'*:

fixes
step1 :: $'s1 \Rightarrow 's1 \Rightarrow \text{bool}$ **and**
step2 :: $'s2 \Rightarrow 's2 \Rightarrow \text{bool}$ **and**
match :: $'i \Rightarrow 's1 \Rightarrow 's2 \Rightarrow \text{bool}$ **and**
order :: $'i \Rightarrow 'i \Rightarrow \text{bool}$
assumes
right-unique step1 **and**
right-unique step2 **and**
final1-stuck: $\forall s1. \text{final1 } s1 \longrightarrow (\nexists s1'. \text{step1 } s1 s1')$ **and**
final2-stuck: $\forall s2. \text{final2 } s2 \longrightarrow (\nexists s2'. \text{step2 } s2 s2')$ **and**
matching-states-agree-on-final:
 $\forall i s1 s2. \text{match } i s1 s2 \longrightarrow \text{final1 } s1 \longleftrightarrow \text{final2 } s2$ **and**
matching-states-are-safe:
 $\forall i s1 s2. \text{match } i s1 s2 \longrightarrow \text{safe-state step1 final1 } s1 \wedge \text{safe-state step2 final2 } s2$
and
order-well-founded: *wfP* *order* **and**
sim: *simulation* *step1* *step2* *match* *order*
obtains
MATCH :: $\text{nat} \times \text{nat} \Rightarrow 's1 \Rightarrow 's2 \Rightarrow \text{bool}$ **and**
ORDER :: $\text{nat} \times \text{nat} \Rightarrow \text{nat} \times \text{nat} \Rightarrow \text{bool}$
where
 $\bigwedge i s1 s2. \text{match } i s1 s2 \implies (\exists j. \text{MATCH } j s1 s2)$
 $\bigwedge j s1 s2. \text{MATCH } j s1 s2 \implies \text{final1 } s1 \longleftrightarrow \text{final2 } s2$ **and**
 $\bigwedge j s1 s2. \text{MATCH } j s1 s2 \implies \text{stuck-state step1 } s1 \longleftrightarrow \text{stuck-state step2 } s2$

```

and
   $\bigwedge j s1 s2. \text{MATCH } j s1 s2 \implies \text{safe-state step1 final1 } s1 \wedge \text{safe-state step2 final2 } s2$ 
and
  wfP ORDER and
  simulation step1 step2 ( $\lambda i s1 s2. \text{MATCH } i s1 s2$ ) ORDER and
  simulation step2 step1 ( $\lambda i s2 s1. \text{MATCH } i s1 s2$ ) 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 +
  well-founded ( $\square$ )
  for
    step1 :: 'state1  $\Rightarrow$  'state1  $\Rightarrow$  bool and
    step2 :: 'state2  $\Rightarrow$  'state2  $\Rightarrow$  bool and
    final1 :: 'state1  $\Rightarrow$  bool and
    final2 :: 'state2  $\Rightarrow$  bool and
    order :: 'index  $\Rightarrow$  'index  $\Rightarrow$  bool (infix  $\square$  70) +
  fixes
    match :: 'index  $\Rightarrow$  'state1  $\Rightarrow$  'state2  $\Rightarrow$  bool
  assumes
    match-final:
      match i s1 s2  $\implies$  final2 s2  $\implies$  final1 s1 and
    simulation:
      match i s1 s2  $\implies$  step2 s2 s2'  $\implies$ 
      ( $\exists i' s1'. \text{step1}^{++} s1 s1' \wedge \text{match } i' s1' s2'$ )  $\vee$  ( $\exists i'. \text{match } i' s1 s2' \wedge i' \square$ 
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*:

$$\begin{aligned} \text{step2}^{++} \ s2 \ s2' \implies \text{match } i1 \ s1 \ s2 \implies \\ (\exists i2 \ s1'. \text{step1}^{++} \ s1 \ s1' \wedge \text{match } i2 \ s1' \ s2') \vee \\ (\exists i2. \text{match } i2 \ s1 \ s2' \wedge \text{order}^{++} \ i2 \ i1) \end{aligned}$$

thm *tranclp-induct*

<proof>

lemma *lift-simulation-eval*:

$$L2.\text{eval } s2 \ s2' \implies \text{match } i1 \ s1 \ s2 \implies \exists i2 \ s1'. L1.\text{eval } s1 \ s1' \wedge \text{match } i2 \ s1' \ s2'$$

<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 also a valid behaviour of the source program.

lemma *simulation-behaviour* :

$$\begin{aligned} L2.\text{state-behaves } s2 \ b2 \implies \neg \text{is-wrong } b2 \implies \text{match } i \ s1 \ s2 \implies \\ \exists b1 \ i'. L1.\text{state-behaves } s1 \ b1 \wedge \text{rel-behaviour } (\text{match } i') \ b1 \ b2 \end{aligned}$$

<proof>

end

5.2 Forward simulation

locale *forward-simulation* =

L1: semantics step1 final1 +

L2: semantics step2 final2 +

well-founded (\sqsubset)

for

step1 :: *'state1* \Rightarrow *'state1* \Rightarrow *bool* **and**

step2 :: *'state2* \Rightarrow *'state2* \Rightarrow *bool* **and**

final1 :: *'state1* \Rightarrow *bool* **and**

final2 :: *'state2* \Rightarrow *bool* **and**

order :: *'index* \Rightarrow *'index* \Rightarrow *bool* (**infix** \sqsubset 70) +

fixes

$match :: 'index \Rightarrow 'state1 \Rightarrow 'state2 \Rightarrow bool$
assumes
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* :
 $L1.state-behaves\ s1\ b1 \Longrightarrow \neg\ is-wrong\ b1 \Longrightarrow match\ i\ s1\ s2 \Longrightarrow$
 $\exists b2\ i'.\ L2.state-behaves\ s2\ b2 \wedge rel-behaviour\ (match\ i')\ b1\ b2$
<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 $\exists b1\ i.\ L1.state-behaves\ s1\ b1 \wedge rel-behaviour\ (match\ i)\ b1\ b2$
<proof>

end

5.3 Bisimulation

locale *bisimulation* =
forward-simulation $step1\ step2\ final1\ final2\ order\ match$ +
backward-simulation $step1\ step2\ final1\ final2\ order\ match$

for
 $step1 :: 'state1 \Rightarrow 'state1 \Rightarrow bool$ **and**
 $step2 :: 'state2 \Rightarrow 'state2 \Rightarrow bool$ **and**
 $final1 :: 'state1 \Rightarrow bool$ **and**
 $final2 :: 'state2 \Rightarrow bool$ **and**
 $order :: 'index \Rightarrow 'index \Rightarrow bool$ **and**
 $match :: 'index \Rightarrow 'state1 \Rightarrow 'state2 \Rightarrow bool$

lemma *obtains-bisimulation-from-forward-simulation:*

fixes

$step1 :: 'state1 \Rightarrow 'state1 \Rightarrow bool$ **and** $final1 :: 'state1 \Rightarrow bool$ **and**
 $step2 :: 'state2 \Rightarrow 'state2 \Rightarrow bool$ **and** $final2 :: 'state2 \Rightarrow bool$ **and**
 $match :: 'index \Rightarrow 'state1 \Rightarrow 'state2 \Rightarrow bool$ **and**
 $lt :: 'index \Rightarrow 'index \Rightarrow bool$

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:

$\forall i s1 s2. match i s1 s2 \longrightarrow safe-state step1 final1 s1 \wedge safe-state step2 final2$

$s2$ **and**

wfP lt and

fsim: $\forall i s1 s2 s1'. match i s1 s2 \longrightarrow step1 s1 s1' \longrightarrow$

$(\exists i' s2'. step2^{++} s2 s2' \wedge match i' s1' s2') \vee (\exists i'. match i' s1' s2 \wedge lt i' i)$

obtains

$MATCH :: nat \times nat \Rightarrow 'state1 \Rightarrow 'state2 \Rightarrow bool$ **and**

$ORDER :: nat \times nat \Rightarrow nat \times nat \Rightarrow bool$

where

$bisimulation step1 step2 final1 final2 ORDER MATCH$

$\langle proof \rangle$

corollary *ex-bisimulation-from-forward-simulation:*

fixes

$step1 :: 'state1 \Rightarrow 'state1 \Rightarrow bool$ **and** $final1 :: 'state1 \Rightarrow bool$ **and**

$step2 :: 'state2 \Rightarrow 'state2 \Rightarrow bool$ **and** $final2 :: 'state2 \Rightarrow bool$ **and**

$match :: 'index \Rightarrow 'state1 \Rightarrow 'state2 \Rightarrow bool$ **and**

$lt :: 'index \Rightarrow 'index \Rightarrow bool$

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:

$\forall i s1 s2. match i s1 s2 \longrightarrow safe-state step1 final1 s1 \wedge safe-state step2 final2$

$s2$ **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 $\exists(MATCH :: nat \times nat \Rightarrow 'state1 \Rightarrow 'state2 \Rightarrow bool)$
 $(ORDER :: nat \times nat \Rightarrow nat \times nat \Rightarrow bool).$
bisimulation step1 step2 final1 final2 ORDER MATCH
 $\langle proof \rangle$

lemma *obtains-bisimulation-from-backward-simulation:*

fixes

step1 :: 'state1 \Rightarrow 'state1 \Rightarrow bool **and** *final1* :: 'state1 \Rightarrow bool **and**
step2 :: 'state2 \Rightarrow 'state2 \Rightarrow bool **and** *final2* :: 'state2 \Rightarrow bool **and**
match :: 'index \Rightarrow 'state1 \Rightarrow 'state2 \Rightarrow bool **and**
lt :: 'index \Rightarrow 'index \Rightarrow 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:

$\forall i\ s1\ s2. match\ i\ s1\ s2 \longrightarrow safe-state\ step1\ final1\ s1 \wedge safe-state\ step2\ final2$

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 :: $nat \times nat \Rightarrow 'state1 \Rightarrow 'state2 \Rightarrow bool$ **and**

ORDER :: $nat \times nat \Rightarrow nat \times nat \Rightarrow bool$

where

bisimulation step1 step2 final1 final2 ORDER MATCH

$\langle proof \rangle$

corollary *ex-bisimulation-from-backward-simulation:*

fixes

step1 :: 'state1 \Rightarrow 'state1 \Rightarrow bool **and** *final1* :: 'state1 \Rightarrow bool **and**

step2 :: 'state2 \Rightarrow 'state2 \Rightarrow bool **and** *final2* :: 'state2 \Rightarrow bool **and**

match :: 'index \Rightarrow 'state1 \Rightarrow 'state2 \Rightarrow bool **and**

lt :: 'index \Rightarrow 'index \Rightarrow 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:

$\forall i\ s1\ s2. match\ i\ s1\ s2 \longrightarrow safe-state\ step1\ final1\ s1 \wedge safe-state\ step2\ final2$

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 $\exists(MATCH :: nat \times nat \Rightarrow 'state1 \Rightarrow 'state2 \Rightarrow bool)$

$(ORDER :: nat \times nat \Rightarrow nat \times nat \Rightarrow bool).$

bisimulation step1 step2 final1 final2 ORDER MATCH

<proof>

5.4 Composition of backward simulations

definition *rel-comp* ::

$('a \Rightarrow 'b \Rightarrow 'c \Rightarrow \text{bool}) \Rightarrow ('d \Rightarrow 'c \Rightarrow 'e \Rightarrow \text{bool}) \Rightarrow ('a \times 'd) \Rightarrow 'b \Rightarrow 'e \Rightarrow \text{bool}$

where

$\text{rel-comp } r1 \ r2 \ i \equiv (r1 \ (\text{fst } i) \ \text{OO } r2 \ (\text{snd } i))$

lemma *backward-simulation-composition*:

assumes

backward-simulation step1 step2 final1 final2 order1 match1

backward-simulation step2 step3 final2 final3 order2 match2

shows

backward-simulation step1 step3 final1 final3

$(\text{lex-prodp } \text{order1}^{++} \ \text{order2}) \ (\text{rel-comp } \text{match1 } \text{match2})$

<proof>

context

fixes $r :: 'i \Rightarrow 'a \Rightarrow 'a \Rightarrow \text{bool}$

begin

fun *rel-comp-pow* **where**

rel-comp-pow [] $x \ y = \text{False}$ |

rel-comp-pow [i] $x \ y = r \ i \ x \ y$ |

rel-comp-pow (i # is) $x \ z = (\exists y. r \ i \ x \ y \wedge \text{rel-comp-pow } \text{is } y \ z)$

end

lemma *backward-simulation-pow*:

assumes

backward-simulation step step final final order match

shows

backward-simulation step step final final (lexp order⁺⁺) (rel-comp-pow match)

<proof>

definition *lockstep-backward-simulation* **where**

lockstep-backward-simulation step1 step2 match \equiv

$\forall s1 \ s2 \ s2'. \text{match } s1 \ s2 \longrightarrow \text{step2 } s2 \ s2' \longrightarrow (\exists s1'. \text{step1 } s1 \ s1' \wedge \text{match } s1' \ s2')$

definition *plus-backward-simulation* **where**

plus-backward-simulation step1 step2 match \equiv

$\forall s1 \ s2 \ s2'. \text{match } s1 \ s2 \longrightarrow \text{step2 } s2 \ s2' \longrightarrow (\exists s1'. \text{step1}^{++} \ s1 \ s1' \wedge \text{match } s1' \ s2')$

lemma

assumes *lockstep-backward-simulation step1 step2 match*

shows *plus-backward-simulation step1 step2 match*

$\langle \text{proof} \rangle$

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

fixes

$\text{match} :: 'state1 \Rightarrow 'state2 \Rightarrow \text{bool}$ **and**

$\text{step1} :: 'state1 \Rightarrow 'state1 \Rightarrow \text{bool}$ **and**

$\text{step2} :: 'state2 \Rightarrow 'state2 \Rightarrow \text{bool}$

assumes

$\text{lockstep-simulation} :: \bigwedge s1\ s2\ s2'. \text{match}\ s1\ s2 \implies \text{step2}\ s2\ s2' \implies (\exists s1'. \text{step1}\ s1\ s1' \wedge \text{match}\ s1'\ s2')$ **and**

$\text{match} :: \text{match}\ s1\ s2$ **and**

$\text{step} :: \text{step2}\ s2\ s2'$

shows $\exists s1'. \text{step1}^{++}\ s1\ s1' \wedge \text{match}\ s1'\ s2'$

$\langle \text{proof} \rangle$

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

fixes

$\text{match} :: 'state1 \Rightarrow 'state2 \Rightarrow \text{bool}$ **and**

$\text{step1} :: 'state1 \Rightarrow 'state1 \Rightarrow \text{bool}$ **and**

$\text{step2} :: 'state2 \Rightarrow 'state2 \Rightarrow \text{bool}$ **and**

$\text{measure} :: 'state2 \Rightarrow \text{nat}$

assumes

$\text{lockstep-simulation} :: \bigwedge s1\ s2\ s2'. \text{match}\ s1\ s2 \implies \text{step2}\ s2\ s2' \implies (\exists s1'. \text{step1}\ s1\ s1' \wedge \text{match}\ s1'\ s2')$ **and**

$\text{match} :: \text{match}\ s1\ s2$ **and**

$\text{step} :: \text{step2}\ s2\ s2'$

shows $(\exists s1'. \text{step1}\ s1\ s1' \wedge \text{match}\ s1'\ s2') \vee \text{match}\ s1\ s2' \wedge \text{measure}\ s2' < \text{measure}\ s2$

$\langle \text{proof} \rangle$

lemma *plus-to-star-backward-simulation*:

fixes

$\text{match} :: 'state1 \Rightarrow 'state2 \Rightarrow \text{bool}$ **and**

$\text{step1} :: 'state1 \Rightarrow 'state1 \Rightarrow \text{bool}$ **and**

$\text{step2} :: 'state2 \Rightarrow 'state2 \Rightarrow \text{bool}$ **and**

$\text{measure} :: 'state2 \Rightarrow \text{nat}$

assumes

$\text{star-simulation} :: \bigwedge s1\ s2\ s2'. \text{match}\ s1\ s2 \implies \text{step2}\ s2\ s2' \implies (\exists s1'. \text{step1}^{++}\ s1\ s1' \wedge \text{match}\ s1'\ s2')$ **and**

$\text{match} :: \text{match}\ s1\ s2$ **and**

$\text{step} :: \text{step2}\ s2\ s2'$

shows $(\exists s1'. \text{step1}^{++}\ s1\ s1' \wedge \text{match}\ s1'\ s2') \vee \text{match}\ s1\ s2' \wedge \text{measure}\ s2' < \text{measure}\ s2$

$\langle \text{proof} \rangle$

lemma *lockstep-to-plus-forward-simulation*:

fixes

$\text{match} :: 'state1 \Rightarrow 'state2 \Rightarrow \text{bool}$ **and**

$\text{step1} :: 'state1 \Rightarrow 'state1 \Rightarrow \text{bool}$ **and**

$\text{step2} :: 'state2 \Rightarrow 'state2 \Rightarrow \text{bool}$ **and**

$\text{measure} :: 'state2 \Rightarrow \text{nat}$

assumes

$\text{lockstep-simulation} :: \bigwedge s1\ s2\ s2'. \text{match}\ s1\ s2 \implies \text{step2}\ s2\ s2' \implies (\exists s1'. \text{step1}\ s1\ s1' \wedge \text{match}\ s1'\ s2')$ **and**

$\text{match} :: \text{match}\ s1\ s2$ **and**

$\text{step} :: \text{step2}\ s2\ s2'$

shows $(\exists s1'. \text{step1}\ s1\ s1' \wedge \text{match}\ s1'\ s2') \vee \text{match}\ s1\ s2' \wedge \text{measure}\ s2' < \text{measure}\ s2$

$\langle \text{proof} \rangle$

```

    step2 :: 'state2 ⇒ 'state2 ⇒ bool
assumes
    lockstep-simulation:  $\bigwedge s1\ s2\ s2'.\ match\ s1\ s2 \implies step1\ s1\ s1' \implies (\exists s2'.\ step2\ s2\ s2' \wedge match\ s1'\ s2')$  and
    match: match s1 s2 and
    step: step1 s1 s1'
shows  $\exists s2'.\ step2^{++}\ s2\ s2' \wedge match\ s1'\ s2'$ 
⟨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  $\Leftarrow$  50) where
  (f  $\Leftarrow$  g) x  $\equiv$  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 = ( $\lambda$ -. None) |
  option-comp-pow (Suc 0) = f |
  option-comp-pow (Suc n) = (option-comp-pow n  $\Leftarrow$  f)

end

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

```

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: \neg *is-wrong* *b2*

shows \exists *b1* *i*. *L1*.*prog-behaves* *p1* *b1* \wedge *rel-behaviour* (*match* *i*) *b1* *b2*

<proof>

end

6.2 Composition of compilers

lemma *compiler-composition*:

assumes

compiler *step1* *step2* *final1* *final2* *load1* *load2* *order1* *match1* *compile1* **and**

compiler *step2* *step3* *final2* *final3* *load2* *load3* *order2* *match2* *compile2*

shows *compiler* *step1* *step3* *final1* *final3* *load1* *load3*

(*lex-prodp* *order1*⁺⁺ *order2*) (*rel-comp* *match1* *match2*) (*compile2* \Leftarrow *compile1*)

<proof>

lemma *compiler-composition-pow*:

assumes

compiler *step* *step* *final* *final* *load* *load* *order* *match* *compile*

shows *compiler* *step* *step* *final* *final* *load* *load*

(*lexp* *order*⁺⁺) (*rel-comp-pow* *match*) (*option-comp-pow* *compile* *n*)

<proof>

end

7 Fixpoint of Converging Program Transformations

theory *Fixpoint*

imports *Compiler*

begin

context

fixes

```

    m :: 'a ⇒ nat and
    f :: 'a ⇒ 'a option
begin

function fixpoint :: 'a ⇒ 'a option where
  fixpoint x = (
    case f x of
      None ⇒ None |
      Some x' ⇒ 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 step final final load load order match compile
  shows compiler step step final final load load
    (lexp order++) (rel-comp-pow match) (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.