A Generic Framework for Verified Compilers

Martin Desharnais

April 20, 2020

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

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 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 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* }
by (*auto simp*: *lex-prod-def lex-prodp-def*)

lemma *lex-prodp-wfP*:
assumes
wfP r1 **and**

wfP r2
shows *wfP lex-prodp*
proof (*rule wfPUNIVI*)
show $\bigwedge P.\ \forall x.\ (\forall y.\ lex\text{-}prodp\ y\ x \longrightarrow P\ y) \longrightarrow P\ x \Longrightarrow (\bigwedge x.\ P\ x)$
proof −
fix *P*
assume $\forall x.\ (\forall y.\ lex\text{-}prodp\ y\ x \longrightarrow P\ y) \longrightarrow P\ x$
hence *hyps*: $(\bigwedge y1\ y2.\ lex\text{-}prodp\ (y1,\ y2)\ (x1,\ x2) \Longrightarrow P\ (y1,\ y2)) \Longrightarrow P\ (x1,\ x2)$ **for** *x1 x2*
by *fast*
show $(\bigwedge x.\ P\ x)$
apply (*simp only*: *split-paired-all*)
apply (*atomize* (*full*))
apply (*rule allI*)
apply (*rule wfP-induct-rule*[*OF assms*(*1*), *of* $\lambda y.\ \forall b.\ P\ (y,\ b)$])
apply (*rule allI*)
apply (*rule wfP-induct-rule*[*OF assms*(*2*), *of* $\lambda b.\ P\ (x,\ b)$ **for** *x*])
using *hyps*[*unfolded lex-prodp-def*, *simplified*]
by *blast*
qed
qed

end

lemma *lex-prodp-well-founded*:
assumes
well-founded r1 **and**
well-founded r2
shows *well-founded* (*lex-prodp r1 r2*)
using *well-founded.intro lex-prodp-wfP assms*[*THEN well-founded.wf*] **by** *auto*

1.2 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*)
by (*induction xs*) (*simp-all add*: *lexp-tail*)

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

thus $(xs,\ ys) \in lex\ \{(x,\ y).\ order\ x\ y\}$
by (*induction xs ys rule*: *lexp.induct*) *simp-all*
next
assume $(xs,\ ys) \in lex\ \{(x,\ y).\ order\ x\ y\}$
thus *lexp order xs ys*
by (*auto intro*!: *lexp-prepend intro*: *lexp-head simp*: *lex-conv*)
qed

lemma *lex-list-wfP*: *wfP order* $\Longrightarrow$ *wfP* (*lexp order*)
by (*simp add*: *lexp-lex wf-lex wfP-def*)

lemma *lex-list-well-founded*:
assumes *well-founded order*
shows *well-founded* (*lexp order*)
using *well-founded.intro assms*(*1*)[*THEN well-founded.wf*, *THEN lex-list-wfP*]
by *auto*

end

2 Infinitely Transitive Closure

theory *Inf*
imports *Well-founded*
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-wf r order n x* $\Longrightarrow$ *inf-wf r order m x* |
inf-wf-step: r^{++} *x y* $\Longrightarrow$ *inf-wf r order n y* $\Longrightarrow$ *inf-wf r order m x*

lemma *inf-wf-to-step-inf-wf*:
assumes *well-founded order*
shows *inf-wf r order n x* $\Longrightarrow \exists y\ m.\ r\ x\ y \wedge$ *inf-wf r order m y*
proof (*induction n arbitrary*: *x rule*: *well-founded.induct*[*OF assms*(*1*)])
case (*1 n*)
from *1.prems*(*1*) **show** *?case*
proof (*induction rule*: *inf-wf.cases*)
case (*inf-wf m n' x'*)
then show *?case* **using** *1.IH* **by** *simp*
next
case (*inf-wf-step x' y m n'*)
then show *?case*
by (*metis converse-tranclpE inf-wf.inf-wf-step*)
qed
qed

lemma *inf-wf-to-inf*:
assumes *well-founded order*
shows *inf-wf r order n x* $\Longrightarrow$ *inf r x*
proof (*coinduction arbitrary*: *x n rule*: *inf.coinduct*)
case (*inf x n*)
then obtain *y m* **where** *r x y* **and** *inf-wf r order m y*
using *inf-wf-to-step-inf-wf*[*OF assms*(*1*) *inf*(*1*)] **by** *metis*
thus *?case* **by** *auto*
qed

lemma *step-inf*:
assumes
deterministic: $\bigwedge x\ y\ z.\ r\ x\ y \Longrightarrow r\ x\ z \Longrightarrow y = z$
shows *r x y* $\Longrightarrow$ *inf r x* $\Longrightarrow$ *inf r y*
by (*metis deterministic inf.cases*)

lemma *star-inf*:
assumes
deterministic: $\bigwedge x\ y\ z.\ r\ x\ y \Longrightarrow r\ x\ z \Longrightarrow y = z$
shows $r^{**}\ x\ y \Longrightarrow$ *inf r x* $\Longrightarrow$ *inf r y*
proof (*induction y rule*: *rtranclp-induct*)
case *base*
then show *?case* **.**
next
case (*step y z*)
then show *?case*
using *step-inf deterministic* **by** *metis*
qed

end

3 The Dynamic Representation of a Language

theory *Semantics*
imports *Main Behaviour Inf* **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 (*induction y rule*: *rtranclp-induct*)
case *base*
then show *?case* **by** *simp*
next
case (*step y z*)

then show *?case*
using *assms* **by** (*auto simp*: *finished-def*)
qed

locale *semantics* =
fixes
step :: *'state* ⇒ *'state* ⇒ *bool* (**infix** → *50*) **and**
final :: *'state* ⇒ *bool*
assumes
final-finished: *final s* ⟹ *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 (→)—usually written as an infix → arrow—and final states *final*.

lemma *finished-step*:
step s s' ⟹ ¬*finished step s*
by (*auto simp add*: *finished-def*)

abbreviation *eval* :: *'state* ⇒ *'state* ⇒ *bool* (**infix** →* *50*) **where**
eval ≡ *step***

abbreviation *inf-step* :: *'state* ⇒ *bool* **where**
inf-step ≡ *inf step*

notation
inf-step (*'*(→^∞*'*) [] *50*) **and**
inf-step (- →^∞ [*55*] *50*)

lemma *finished-inf*: *s* →^∞ ⟹ ¬ *finished step s*
using *inf.cases finished-step* **by** *metis*

lemma *eval-deterministic*:
assumes
deterministic: ⋀*x y z*. *step x y* ⟹ *step x z* ⟹ *y* = *z*
shows *s1* →* *s2* ⟹ *s1* →* *s3* ⟹ *finished step s2* ⟹ *finished step s3* ⟹ *s2* = *s3*
proof(*induction s1 arbitrary*: *s3 rule*: *converse-rtranclp-induct*)
case *base*
then show *?case* **by** (*simp add*: *finished-star*)
next
case (*step y z*)
then show *?case*
by (*metis converse-rtranclpE deterministic finished-step*)
qed

3.1 Behaviour of a dynamic execution

inductive *behaves* :: *'state* ⇒ *'state behaviour* ⇒ *bool* (**infix** ⇓ *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 *behaves-deterministic*:
assumes
deterministic: $\bigwedge x\ y\ z.$ *step x y* $\Longrightarrow$ *step x z* $\Longrightarrow$ $y = z$
shows *s* $\Downarrow$ *b1* $\Longrightarrow$ *s* $\Downarrow$ *b2* $\Longrightarrow$ *b1* $=$ *b2*
proof (*induction s b1 rule*: *behaves.induct*)
case (*state-terminates s1 s2*)
show *?case* **using** *state-terminates.prems state-terminates.hyps*
proof (*induction s1 b2 rule*: *behaves.induct*)
case (*state-terminates s1 s3*)
then show *?case*
using *eval-deterministic deterministic* **by** *simp*
next
case (*state-diverges s1*)
then show *?case*
using *deterministic star-inf*[*THEN finished-inf*] **by** *simp*
next
case (*state-goes-wrong s1 s3*)
then show *?case*
using *eval-deterministic deterministic* **by** *blast*
qed
next
case (*state-diverges s1*)
show *?case* **using** *state-diverges.prems state-diverges.hyps*
proof (*induction s1 b2 rule*: *behaves.induct*)
case (*state-terminates s1 s2*)
then show *?case*
using *deterministic star-inf*[*THEN finished-inf*] **by** *simp*
next
case (*state-diverges s1*)
then show *?case* **by** *simp*
next
case (*state-goes-wrong s1 s2*)
then show *?case*
using *deterministic star-inf*[*THEN finished-inf*] **by** *simp*
qed
next
case (*state-goes-wrong s1 s2*)
show *?case* **using** *state-goes-wrong.prems state-goes-wrong.hyps*
proof (*induction s1 b2*)

case (*state-terminates s1 s3*)
then show *?case*
using *eval-deterministic deterministic* **by** *blast*
next
case (*state-diverges s1*)
then show *?case*
using *deterministic star-inf*[*THEN finished-inf*] **by** *simp*
next
case (*state-goes-wrong s1 s3*)
then show *?case*
using *eval-deterministic deterministic* **by** *simp*
qed
qed

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 option*

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*.

end

end

5 Simulations Between Dynamic Executions

theory *Simulation*
imports *Semantics Inf Well-founded*
begin

locale *backward-simulation* =
  L1: *semantics step1 final1* +
  L2: *semantics step2 final2* +
  well-founded (⊏)
  for
    step1 :: *'state1* ⇒ *'state1* ⇒ *bool* **and**
    step2 :: *'state2* ⇒ *'state2* ⇒ *bool* **and**
    final1 :: *'state1* ⇒ *bool* **and**
    final2 :: *'state2* ⇒ *bool* **and**
    order :: *'index* ⇒ *'index* ⇒ *bool* (**infix** ⊏ *70*) +
  fixes
    match :: *'index* ⇒ *'state1* ⇒ *'state2* ⇒ *bool*
  assumes
    match-final:
      match i s1 s2 ⟹ *final2 s2* ⟹ *final1 s1* **and**
    simulation:
      match i1 s1 s2 ⟹ *step2 s2 s2'* ⟹
        (∃ *i2 s1'*. *step1*$^{++}$ *s1 s1'* ∧ *match i2 s1' s2'*) ∨ (∃ *i2*. *match i2 s1 s2'* ∧ *i2* ⊏ *i1*)
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 (⊏) ordering.

end

locale *forward-simulation* =
  L1: *semantics step1 final1* +
  L2: *semantics step2 final2* +
  well-founded (⊏)
  for
    step1 :: *'state1* ⇒ *'state1* ⇒ *bool* **and**
    step2 :: *'state2* ⇒ *'state2* ⇒ *bool* **and**
    final1 :: *'state1* ⇒ *bool* **and**
    final2 :: *'state2* ⇒ *bool* **and**
    order :: *'index* ⇒ *'index* ⇒ *bool* (**infix** ⊏ *70*) +
  fixes
    match :: *'index* ⇒ *'state1* ⇒ *'state2* ⇒ *bool*
  assumes
    match-final:
      match i s1 s2 ⟹ *final1 s1* ⟹ *final2 s2* **and**
    simulation:

```
    match i1 s1 s2 ⟹ step1 s1 s1' ⟹
      (∃ i' s2'. step2^++ s2 s2' ∧ match i' s1' s2') ∨ (∃ i'. match i' s1 s2' ∧ i' ⊏
i1)

locale bisimulation =
  forward-simulation step1 step2 final1 final2 order match +
  backward-simulation step1 step2 final1 final2 order match
  for
    step1 :: 'state1 ⇒ 'state1 ⇒ bool and
    step2 :: 'state2 ⇒ 'state2 ⇒ bool and
    final1 :: 'state1 ⇒ bool and
    final2 :: 'state2 ⇒ bool and
    order :: 'index ⇒ 'index ⇒ bool and
    match :: 'index ⇒ 'state1 ⇒ 'state2 ⇒ bool

context backward-simulation begin

lemma lift-simulation-plus:
  step2^++ s2 s2' ⟹ match i1 s1 s2 ⟹
    (∃ i2 s1'. step1^++ s1 s1' ∧ match i2 s1' s2') ∨
    (∃ i2. match i2 s1 s2' ∧ order^++ i2 i1)
  thm tranclp-induct
proof(induction s2' arbitrary: i1 s1 rule: tranclp-induct)
  case (base s2')
  from simulation[OF base.prems(1) base.hyps(1)] show ?case
    by auto
next
  case (step s2' s2'')
  show ?case
    using step.IH[OF ‹match i1 s1 s2›]
  proof
    assume ∃ i2 s1'. step1^++ s1 s1' ∧ match i2 s1' s2'
    then obtain i2 s1' where step1^++ s1 s1' and match i2 s1' s2' by auto
    from simulation[OF ‹match i2 s1' s2'› ‹step2 s2' s2''›] show ?thesis
    proof
      assume ∃ i3 s1''. step1^++ s1' s1'' ∧ match i3 s1'' s2''
      then obtain i3 s1'' where step1^++ s1' s1'' and match i3 s1'' s2'' by auto
      then show ?thesis
        using tranclp-trans[OF ‹step1^++ s1 s1'›] by auto
    next
      assume ∃ i3. match i3 s1' s2'' ∧ i3 ⊏ i2
      then obtain i3 where match i3 s1' s2'' and i3 ⊏ i2 by auto
      then show ?thesis
        using ‹step1^++ s1 s1'› by auto
    qed
  next
    assume ∃ i2. match i2 s1 s2' ∧ (⊏)^++ i2 i1
    then obtain i3 where match i3 s1 s2' and (⊏)^++ i3 i1 by auto
    then show ?thesis
```

```
    using simulation[OF ‹match i3 s1 s2'› ‹step2 s2' s2''›] by auto
  qed
qed

lemma lift-simulation-eval:
  L2.eval s2 s2' ⟹ match i1 s1 s2 ⟹ ∃ i2 s1'. L1.eval s1 s1' ∧ match i2 s1' s2'
proof(induction s2 arbitrary: i1 s1 rule: converse-rtranclp-induct)
  case (base s2)
  thus ?case by auto
next
  case (step s2 s2'')
  from simulation[OF ‹match i1 s1 s2› ‹step2 s2 s2''›] show ?case
  proof
    assume ∃ i2 s1'. step1^++ s1 s1' ∧ match i2 s1' s2''
    thus ?thesis
      by (meson rtranclp-trans step.IH tranclp-into-rtranclp)
  next
    assume ∃ i2. match i2 s1 s2'' ∧ i2 ⊏ i1
    thus ?thesis
      by (auto intro: step.IH)
  qed
qed

lemma backward-simulation-inf:
  assumes
    match i s1 s2 and
    inf step2 s2
  shows inf step1 s1
proof −
  from assms have inf-wf step1 order i s1
  proof (coinduction arbitrary: i s1 s2)
    case inf-wf
    obtain s2' where step2 s2 s2' and inf step2 s2'
      using inf-wf(2) by (auto elim: inf.cases)
    from simulation[OF ‹match i s1 s2› ‹step2 s2 s2'›] show ?case
      using ‹inf step2 s2'› by auto
  qed
  thus ?thesis using inf-wf-to-inf
    by (auto intro: inf-wf-to-inf well-founded-axioms)
qed
```

5.1 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.behaves s_2 $b_2 \Longrightarrow \neg$*is-wrong* $b_2 \Longrightarrow$ *match* i s_1 $s_2 \Longrightarrow$
    $\exists\, b_1$ i'. *L1.behaves* s_1 $b_1 \wedge$ *rel-behaviour* (*match* i') b_1 b_2
proof(*induction rule*: *L2.behaves.cases*)
  case (*state-terminates s2 s2'*)
  then obtain i' *s1'* **where** *L1.eval* s_1 *s1'* **and** *match* i' *s1' s2'*
    using *lift-simulation-eval* **by** *blast*
  hence *final1 s1'*
    by (*auto intro*: *state-terminates.hyps match-final*)
  hence *L1.behaves* s_1 (*Terminates s1'*)
    using *L1.final-finished*
    by (*simp add*: *L1.state-terminates* ‹*L1.eval* s_1 *s1'*›)
  moreover have *rel-behaviour* (*match i'*) (*Terminates s1'*) b_2
    by (*simp add*: ‹*match i' s1' s2'*› *state-terminates.hyps*)
  ultimately show *?case* **by** *blast*
next
  case (*state-diverges s2*)
  then show *?case*
    using *backward-simulation-inf L1.state-diverges* **by** *fastforce*
next
  case (*state-goes-wrong s2 s2'*)
  then show *?case* **using** ‹¬*is-wrong* b_2› **by** *simp*
qed

end

5.2 Composition of backward 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*))

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
      (*lex-prodp order1*$^{++}$ *order2*) (*rel-comp match1 match2*)
proof *intro-locales*
  show *semantics step1 final1*
    by (*auto intro*: *backward-simulation.axioms assms*)
next
  show *semantics step3 final3*
    by (*auto intro*: *backward-simulation.axioms assms*)
next
  show *well-founded* (*lex-prodp order1*$^{++}$ *order2*)
    using *assms*[*THEN backward-simulation.axioms*(*3*)]

```
  by (simp add: lex-prodp-well-founded well-founded.intro well-founded.wf wfP-trancl)
next
  show backward-simulation-axioms step1 step3 final1 final3
    (lex-prodp order1^++ order2) (rel-comp match1 match2)
  proof
    fix i s1 s3
    assume
      match: rel-comp match1 match2 i s1 s3 and
      final: final3 s3
    obtain i1 i2 s2 where match1 i1 s1 s2 and match2 i2 s2 s3 and i = (i1, i2)
      using match unfolding rel-comp-def by auto
    thus final1 s1
      using final assms[THEN backward-simulation.match-final]
      by simp
  next
    fix i s1 s3 s3'
    assume
      match: rel-comp match1 match2 i s1 s3 and
      step: step3 s3 s3'
    obtain i1 i2 s2 where match1 i1 s1 s2 and match2 i2 s2 s3 and i-def: i =
(i1, i2)
      using match unfolding rel-comp-def by auto
    from backward-simulation.simulation[OF assms(2) ‹match2 i2 s2 s3› step]
    show (∃ i' s1'. step1^++ s1 s1' ∧ rel-comp match1 match2 i' s1' s3') ∨
      (∃ i'. rel-comp match1 match2 i' s1 s3' ∧ lex-prodp order1^++ order2 i' i)
      (is (∃ i' s1'. ?STEPS i' s1') ∨ ?STALL)
    proof
      assume ∃ i2' s2'. step2^++ s2 s2' ∧ match2 i2' s2' s3'
      then obtain i2' s2' where step2^++ s2 s2' and match2 i2' s2' s3' by auto
      from backward-simulation.lift-simulation-plus[OF assms(1) ‹step2^++ s2 s2'›
‹match1 i1 s1 s2›]
      show ?thesis
      proof
        assume ∃ i2 s1'. step1^++ s1 s1' ∧ match1 i2 s1' s2'
        then obtain i2 s1' where step1^++ s1 s1' and match1 i2 s1' s2' by auto
        hence ?STEPS (i2, i2') s1'
          by (auto intro: ‹match2 i2' s2' s3'› simp: rel-comp-def)
        thus ?thesis by auto
      next
        assume ∃ i2. match1 i2 s1 s2' ∧ order1^++ i2 i1
        then obtain i2'' where match1 i2'' s1 s2' and order1^++ i2'' i1 by auto
        hence ?STALL
          unfolding rel-comp-def i-def lex-prodp-def
          using ‹match2 i2' s2' s3'› by auto
        thus ?thesis by simp
      qed
    next
      assume ∃ i2'. match2 i2' s2 s3' ∧ order2 i2' i2
      then obtain i2' where match2 i2' s2 s3' and order2 i2' i2 by auto
```

```
      hence ?STALL
        unfolding rel-comp-def i-def lex-prodp-def
        using ‹match1 i1 s1 s2› by auto
      thus ?thesis by simp
    qed
  qed
qed

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 step final final order match
  shows
    backward-simulation step step final final (lexp order^++) (rel-comp-pow match)
proof intro-locales
  show semantics step final
    by (auto intro: backward-simulation.axioms assms)
next
  show well-founded (lexp order^++)
    using backward-simulation.axioms(3)[OF assms] lex-list-well-founded
    using well-founded.intro well-founded.wf wfP-trancl by blast
next
 show backward-simulation-axioms step step final final (lexp order^++) (rel-comp-pow
match)
  proof unfold-locales
    fix is s1 s2
    assume rel-comp-pow match is s1 s2 and final s2
    thus final s1 thm rel-comp-pow.induct
    proof (induction is s1 s2 rule: rel-comp-pow.induct)
      case (1 x y)
      then show ?case by simp
    next
      case (2 i x y)
      then show ?case
        using backward-simulation.match-final[OF assms(1)] by simp
    next
      case (3 i1 i2 is x z)
      from 3.prems[simplified] obtain y where
        match: match i1 x y and rel-comp-pow match (i2 # is) y z
```

```
      by auto
    hence final y using 3.IH 3.prems by simp
    thus ?case
      using 3.prems match backward-simulation.match-final[OF assms(1)] by
auto
  qed
 next
  fix is s1 s3 s3'
  assume rel-comp-pow match is s1 s3 and step s3 s3'
  hence (∃ is' s1'. step^++ s1 s1' ∧ length is' = length is ∧ rel-comp-pow match
is' s1' s3') ∨
    (∃ is'. rel-comp-pow match is' s1 s3' ∧ lexp order^++ is' is)
  proof (induction is s1 s3 arbitrary: s3' rule: rel-comp-pow.induct)
    case 1
    then show ?case by simp
  next
    case (2 i s1 s3)
   from backward-simulation.simulation[OF assms(1) 2.prems[simplified]] show
?case
    proof
      assume ∃ i' s1'. step^++ s1 s1' ∧ match i' s1' s3'
      then obtain i' s1' where step^++ s1 s1' and match i' s1' s3' by auto
      hence step^++ s1 s1' ∧ rel-comp-pow match [i'] s1' s3' by simp
      thus ?thesis
        by (metis length-Cons)
    next
      assume ∃ i'. match i' s1 s3' ∧ order i' i
      then obtain i' where match i' s1 s3' and order i' i by auto
      hence rel-comp-pow match [i'] s1 s3' ∧ lexp order^++ [i'] [i]
        by (simp add: lexp-head tranclp.r-into-trancl)
      thus ?thesis by blast
    qed
  next
    case (3 i1 i2 is s1 s3)
    from 3.prems[simplified] obtain s2 where
      match i1 s1 s2 and 0: rel-comp-pow match (i2 # is) s2 s3
      by auto
    from 3.IH[OF 0 3.prems(2)] show ?case
    proof
      assume ∃ is' s2'. step^++ s2 s2' ∧ length is' = length (i2 # is) ∧
        rel-comp-pow match is' s2' s3'
      then obtain i2' is' s2' where
        step^++ s2 s2' and length is' = length is and rel-comp-pow match (i2' #
is') s2' s3'
        by (metis Suc-length-conv)
      from backward-simulation.lift-simulation-plus[OF assms(1) ‹step^++ s2 s2'›
‹match i1 s1 s2›]
      show ?thesis
      proof
```

```
      assume ∃ i2 s1'. step^++ s1 s1' ∧ match i2 s1' s2'
      thus ?thesis
        using ⟨rel-comp-pow match (i2' # is') s2' s3'⟩
        by (metis ⟨length is' = length is⟩ length-Cons rel-comp-pow.simps(3))
    next
      assume ∃ i2. match i2 s1 s2' ∧ order^++ i2 i1
      then obtain i1' where match i1' s1 s2' and order^++ i1' i1 by auto
      hence rel-comp-pow match (i1' # i2' # is') s1 s3'
        using ⟨rel-comp-pow match (i2' # is') s2' s3'⟩ by auto
      moreover have lexp order^++ (i1' # i2' # is') (i1 # i2 # is)
        using ⟨order^++ i1' i1⟩ ⟨length is' = length is⟩
        by (auto intro: lexp-head)
      ultimately show ?thesis by fast
    qed
  next
    assume ∃ i'. rel-comp-pow match i' s2 s3' ∧ lexp order^++ i' (i2 # is)
    then obtain i2' is' where
      rel-comp-pow match (i2' # is') s2 s3' and lexp order^++ (i2' # is') (i2 #
is)
      by (metis lexp.simps)
    thus ?thesis
      by (metis ⟨match i1 s1 s2⟩ lexp.simps(1) rel-comp-pow.simps(3))
  qed
 qed
 thus (∃ is' s1'. step^++ s1 s1' ∧ rel-comp-pow match is' s1' s3') ∨
   (∃ is'. rel-comp-pow match is' s1 s3' ∧ lexp order^++ is' is)
   by auto
qed
qed

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
unfolding plus-backward-simulation-def
proof safe
  fix s1 s2 s2'
  assume match s1 s2 and step2 s2 s2'
  then obtain s1' where step1 s1 s1' and match s1' s2'
    using assms(1) unfolding lockstep-backward-simulation-def by blast
```

then show $\exists s1'.\ step1^{++}\ s1\ s1' \wedge match\ s1'\ s2'$
by *auto*
qed

lemma *lockstep-to-plus-backward-simulation*:
fixes
$match :: 'state1 \Rightarrow 'state2 \Rightarrow bool$ **and**
$step1 :: 'state1 \Rightarrow 'state1 \Rightarrow bool$ **and**
$step2 :: 'state2 \Rightarrow 'state2 \Rightarrow bool$
assumes
lockstep-simulation: $\bigwedge s1\ s2\ s2'.\ match\ s1\ s2 \Longrightarrow step2\ s2\ s2' \Longrightarrow (\exists s1'.\ step1\ s1\ s1' \wedge match\ s1'\ s2')$ **and**
match: *match s1 s2* **and**
step: *step2 s2 s2'*
shows $\exists s1'.\ step1^{++}\ s1\ s1' \wedge match\ s1'\ s2'$
proof −
obtain *s1'* **where** *step1 s1 s1'* **and** *match s1' s2'*
using *lockstep-simulation*[*OF match step*] **by** *auto*
thus *?thesis* **by** *auto*
qed

lemma *lockstep-to-option-backward-simulation*:
fixes
$match :: 'state1 \Rightarrow 'state2 \Rightarrow bool$ **and**
$step1 :: 'state1 \Rightarrow 'state1 \Rightarrow bool$ **and**
$step2 :: 'state2 \Rightarrow 'state2 \Rightarrow bool$ **and**
$measure :: 'state2 \Rightarrow nat$
assumes
lockstep-simulation: $\bigwedge s1\ s2\ s2'.\ match\ s1\ s2 \Longrightarrow step2\ s2\ s2' \Longrightarrow (\exists s1'.\ step1\ s1\ s1' \wedge match\ s1'\ s2')$ **and**
match: *match s1 s2* **and**
step: *step2 s2 s2'*
shows $(\exists s1'.\ step1\ s1\ s1' \wedge match\ s1'\ s2') \vee match\ s1\ s2' \wedge measure\ s2' < measure\ s2$
using *lockstep-simulation*[*OF match step*] **..**

lemma *plus-to-star-backward-simulation*:
fixes
$match :: 'state1 \Rightarrow 'state2 \Rightarrow bool$ **and**
$step1 :: 'state1 \Rightarrow 'state1 \Rightarrow bool$ **and**
$step2 :: 'state2 \Rightarrow 'state2 \Rightarrow bool$ **and**
$measure :: 'state2 \Rightarrow nat$
assumes
star-simulation: $\bigwedge s1\ s2\ s2'.\ match\ s1\ s2 \Longrightarrow step2\ s2\ s2' \Longrightarrow (\exists s1'.\ step1^{++}\ s1\ s1' \wedge match\ s1'\ s2')$ **and**
match: *match s1 s2* **and**
step: *step2 s2 s2'*
shows $(\exists s1'.\ step1^{++}\ s1\ s1' \wedge match\ s1'\ s2') \vee match\ s1\ s2' \wedge measure\ s2' < measure\ s2$

using *star-simulation*[*OF match step*] **..**

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 −
obtain *s2'* **where** *step2 s2 s2'* **and** *match s1' s2'*
using *lockstep-simulation*[*OF match step*] **by** *auto*
thus *?thesis* **by** *auto*
qed

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 step2 final1 final2 order match
for
step1 **and** *step2* **and**
final1 **and** *final2* **and**

load1 :: *'prog1* ⇒ *'state1 option* **and**
load2 :: *'prog2* ⇒ *'state2 option* **and**
order :: *'index* ⇒ *'index* ⇒ *bool* **and**
match +
fixes
compile :: *'prog1* ⇒ *'prog2 option*
assumes
compile-load:
compile p1 = Some p2 ⟹ *load1 p1 = Some s1* ⟹ ∃ *s2 i. load2 p2 = Some s2* ∧ *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**
loads: *load1 p1 = Some s1 load2 p2 = Some s2* **and**
behaves: *L2.behaves s2 b2* **and**
not-wrong: ¬ *is-wrong b2*
shows ∃ *b1 i. L1.behaves s1 b1* ∧ *rel-behaviour* (*match i*) *b1 b2*
proof −
obtain *i* **where** *match i s1 s2*
using *compile-load*[*OF compiles*] *loads* **by** *auto*
then show *?thesis*
using *simulation-behaviour*[*OF behaves not-wrong*]
by *simp*
qed

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* ⇐ *compile1*)
proof (*rule compiler.intro*)
show *language step1 final1*
using *assms*(*1*)[*THEN compiler.axioms*(*1*)] **.**
next

```
  show language step3 final3
    using assms(2)[THEN compiler.axioms(2)] .
next
  show backward-simulation step1 step3 final1 final3
     (lex-prodp order1^++ order2) (rel-comp match1 match2)
    using backward-simulation-composition[OF assms[THEN compiler.axioms(3)]]
.
next
  show compiler-axioms load1 load3 (rel-comp match1 match2) (compile2 ⇐ com-
pile1)
  proof unfold-locales
    fix p1 p3 s1
    assume
      compile: (compile2 ⇐ compile1) p1 = Some p3 and
      load: load1 p1 = Some s1
    obtain p2 where compile1 p1 = Some p2 and compile2 p2 = Some p3
      using compile by (auto simp: bind-eq-Some-conv option-comp-def)
    then obtain s2 i where load2 p2 = Some s2 and match1 i s1 s2
      using assms(1)[THEN compiler.compile-load] load
      by blast
    moreover obtain s3 i' where load3 p3 = Some s3 and match2 i' s2 s3
      using assms(2)[THEN compiler.compile-load, OF ‹compile2 p2 = Some p3›
‹load2 p2 = Some s2›]
      by auto
    ultimately show ∃ s3 i. load3 p3 = Some s3 ∧ rel-comp match1 match2 i s1
s3
      unfolding rel-comp-def by auto
  qed
qed

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 (induction n rule: option-comp-pow.induct)
  case 1
  show ?case
    using assms
   by (auto intro: compiler.axioms compiler.intro compiler-axioms.intro backward-simulation-pow)
next
  case 2
  show ?case
  proof (rule compiler.intro)
    show compiler-axioms load load (rel-comp-pow match) (option-comp-pow com-
pile (Suc 0))
    proof unfold-locales
      fix p1 p2 s1
      assume
```

```
      option-comp-pow compile (Suc 0) p1 = Some p2 and
      load p1 = Some s1
    thus ∃ s2 i. load p2 = Some s2 ∧ rel-comp-pow match i s1 s2
      using compiler.compile-load[OF assms(1)]
      by (metis option-comp-pow.simps(2) rel-comp-pow.simps(2))
  qed
 qed (auto intro: assms compiler.axioms backward-simulation-pow)
next
 case (3 n')
 show ?case
 proof (rule compiler.intro)
  show compiler-axioms load load (rel-comp-pow match) (option-comp-pow com-
pile (Suc (Suc n')))
  proof unfold-locales
   fix p1 p3 s1
   assume
    option-comp-pow compile (Suc (Suc n')) p1 = Some p3 and
    load p1 = Some s1
   then obtain p2 where
    comp: compile p1 = Some p2 and
    comp-IH: option-comp-pow compile (Suc n') p2 = Some p3
    by (auto simp: option-comp-def bind-eq-Some-conv)
   then obtain s2 i where load p2 = Some s2 and match i s1 s2
    using compiler.compile-load[OF assms(1) - ‹load p1 = Some s1›] by blast
   then obtain s3 i' where load p3 = Some s3 and rel-comp-pow match i' s2
s3
    using compiler.compile-load[OF 3.IH comp-IH] by blast
   moreover have rel-comp-pow match (i # i') s1 s3
    using ‹match i s1 s2› ‹rel-comp-pow match i' s2 s3›
    using rel-comp-pow.elims(2) by fastforce
   ultimately show ∃ s3 i. load p3 = Some s3 ∧ rel-comp-pow match i s1 s3
    by auto
  qed
 qed (auto intro: assms compiler.axioms backward-simulation-pow)
qed

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'
  )
by pat-completeness auto
termination
proof (relation measure m)
  show wf (measure m) by auto
next
  fix x x'
  assume f x = Some x' and m x' < m x
  thus (x', x) ∈ measure m by simp
qed

end

lemma fixpoint-to-comp-pow:
  fixpoint m f x = y ⟹ ∃ n. option-comp-pow f n x = y
proof (induction x arbitrary: y rule: fixpoint.induct[where f = f and m = m])
  case (1 x)
  show ?case
  proof (cases f x)
    case None
    then show ?thesis
      using 1.prems
    by (metis (no-types, lifting) fixpoint.simps option.case-eq-if option-comp-pow.simps(1))
  next
    case (Some a)
    show ?thesis
    proof (cases m a < m x)
      case True
      hence fixpoint m f a = y
        using 1.prems Some by simp
      then show ?thesis
        using 1.IH[OF Some True]
      by (metis Some bind.simps(2) old.nat.exhaust option-comp-def option-comp-pow.simps(1,3))
    next
      case False
      then show ?thesis
        using 1.prems Some
        apply simp
        by (metis option-comp-pow.simps(2))
    qed
  qed
```

qed

lemma *fixpoint-eq-comp-pow*:
  $\exists\, n.$ *fixpoint m f x* = *option-comp-pow f n x*
  by (*metis fixpoint-to-comp-pow*)

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 (*rule compiler.intro*)
  show *compiler-axioms load load* (*rel-comp-pow match*) (*fixpoint m compile*)
  proof *unfold-locales*
    fix *p1 p2 s1*
    assume *fixpoint m compile p1* = *Some p2* **and** *load p1* = *Some s1*
    obtain *n* **where** *fixpoint m compile p1* = *option-comp-pow compile n p1*
      using *fixpoint-eq-comp-pow* **by** *metis*

    thus $\exists$ *s2 i. load p2* = *Some s2* $\wedge$ *rel-comp-pow match i s1 s2*
        using ‹*fixpoint m compile p1* = *Some p2*› *assms compiler.compile-load compiler-composition-pow*
      using ‹*load p1* = *Some s1*› **by** *fastforce*
  qed
qed (*auto intro*: *assms compiler.axioms backward-simulation-pow*)

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.