A Generic Framework for Verified Compilers

Martin Desharnais

May 26, 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

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 Unit

lemma *wfP-unit*: *wfP* (λ() (). *False*)
by (*simp add*: *Nitpick.case-unit-unfold wfP-eq-minimal*)

interpretation *well-founded* λ() (). *False*
apply *unfold-locales*
by (*auto intro*: *wfP-unit*)

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* }
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 ⋀*P*. ∀ *x*. (∀ *y*. *lex-prodp y x* ⟶ *P y*) ⟶ *P x* ⟹ (⋀*x*. *P x*)
proof −
fix *P*
assume ∀ *x*. (∀ *y*. *lex-prodp y x* ⟶ *P y*) ⟶ *P x*
hence *hyps*: (⋀*y1 y2*. *lex-prodp* (*y1*, *y2*) (*x1*, *x2*) ⟹ *P* (*y1*, *y2*)) ⟹ *P* (*x1*, *x2*) **for** *x1 x2*
by *fast*
show (⋀*x*. *P x*)
apply (*simp only*: *split-paired-all*)
apply (*atomize* (*full*))
apply (*rule allI*)
apply (*rule wfP-induct-rule*[*OF assms*(*1*), *of* λ*y*. ∀ *b*. *P* (*y*, *b*)])
apply (*rule allI*)
apply (*rule wfP-induct-rule*[*OF assms*(*2*), *of* λ*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*

1.3 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*)
by (*induction xs*) (*simp-all add*: *lexp-tail*)

lemma *lexp-lex*: *lexp order xs ys* ⟷ (*xs*, *ys*) ∈ *lex* {(*x*, *y*). *order x y*}
proof
assume *lexp order xs ys*
thus (*xs*, *ys*) ∈ *lex* {(*x*, *y*). *order x y*}
by (*induction xs ys rule*: *lexp.induct*) *simp-all*
next
assume (*xs*, *ys*) ∈ *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* ⟹ *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* ⇒ *'a* ⇒ *bool*) ⇒ *'a* ⇒ *bool* **for** *r* **where**
inf-step: *r x y* ⟹ *inf r y* ⟹ *inf r x*

coinductive *inf-wf* :: (*'a* ⇒ *'a* ⇒ *bool*) ⇒ (*'b* ⇒ *'b* ⇒ *bool*) ⇒ *'b* ⇒ *'a* ⇒ *bool*
for *r order* **where**
inf-wf: *order n m* ⟹ *inf-wf r order n x* ⟹ *inf-wf r order m x* |
inf-wf-step: r^{++} *x y* ⟹ *inf-wf r order n y* ⟹ *inf-wf r order m x*

lemma *inf-wf-to-step-inf-wf*:
assumes *well-founded order*
shows *inf-wf r order n x* ⟹ ∃ *y m*. *r x y* ∧ *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 *right-unique r*
shows *r x y* $\Longrightarrow$ *inf r x* $\Longrightarrow$ *inf r y*
using *right-uniqueD*[*OF* ‹*right-unique r*›]
by (*metis inf.cases*)

lemma *star-inf*:
assumes *right-unique r*
shows r^{**} *x y* $\Longrightarrow$ *inf r x* $\Longrightarrow$ *inf r y*
proof (*induction y rule*: *rtranclp-induct*)
case *base*
then show *?case* **by** *assumption*
next
case (*step y z*)
then show *?case*
using *step-inf*[*OF* ‹*right-unique r*›] **by** *metis*
qed

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 (*induction x arbitrary*: *z rule*: *converse-rtranclp-induct*)
case *base*
then show *?case*
by (*auto elim*: *converse-rtranclpE*)
next
case (*step x w*)
hence $R^{**}\ w\ z$
using *right-uniqueD*[*OF* ‹*right-unique R*›]
by (*metis converse-rtranclpE*)
with *step* **show** *?case*
by *simp*
qed

lemma *tranclp-complete-run-right-unique*:
fixes $R :: 'a \Rightarrow 'a \Rightarrow bool$ **and** $x\ y\ z :: 'a$
assumes *right-unique R*
shows $R^{++}\ x\ y \Longrightarrow (\nexists w.\ R\ y\ w) \Longrightarrow R^{++}\ x\ z \Longrightarrow (\nexists w.\ R\ z\ w) \Longrightarrow y = z$
using *right-uniqueD*[*OF* ‹*right-unique R*›, *of x*]
by (*auto dest*!: *tranclpD intro*!: *rtranclp-complete-run-right-unique*[*OF* ‹*right-unique R*›, *of - y z*])

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* :: $('a \Rightarrow 'a \Rightarrow bool) \Rightarrow 'a \Rightarrow bool$ **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 :: $'state \Rightarrow 'state \Rightarrow bool$ (**infix** $\rightarrow$ *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 *inf-not-finished*: *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* **and**
s1 →* *s2* **and** *s1* →* *s3* **and** *finished step s2* **and** *finished step s3*
shows *s2* = *s3*
proof −
have *right-unique step*
using *deterministic* **by** (*auto intro*: *right-uniqueI*)
with *assms* **show** *?thesis*
by (*auto simp*: *finished-def intro*: *rtranclp-complete-run-right-unique*)
qed

lemma *step-converges-or-diverges*: (∃ *s'*. *s* →* *s'* ∧ *finished step s'*) ∨ *s* →^∞
by (*smt* (*verit*, *del-insts*) *finished-def inf.coinduct rtranclp.intros*(*2*) *rtranclp.rtrancl-refl*)

3.1 Behaviour of a dynamic execution

inductive *state-behaves* :: *'state* ⇒ *'state behaviour* ⇒ *bool* (**infix** ↓ *50*) **where**
state-terminates:
s1 →* *s2* ⟹ *finished step s2* ⟹ *final s2* ⟹ *s1* ↓ (*Terminates s2*) |
state-diverges:
s1 →^∞ ⟹ *s1* ↓ *Diverges* |
state-goes-wrong:
s1 →* *s2* ⟹ *finished step s2* ⟹ ¬ *final s2* ⟹ *s1* ↓ (*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 (*rule right-uniqueI*)
  fix *s b1 b2*
  assume $s \downarrow b1$ $s \downarrow b2$
  thus $b1 = b2$
    by (*auto simp*: *finished-def simp del*: *not-ex*
        elim!: *state-behaves.cases*
        dest: *rtranclp-complete-run-right-unique*[*OF* ‹*right-unique* $(\rightarrow)$›, *of s*]
        dest: *final-finished star-inf*[*OF* ‹*right-unique* $(\rightarrow)$›, *THEN inf-not-finished*])
qed

lemma *left-total-state-behaves*: *left-total* $(\downarrow)$
proof (*rule left-totalI*)
  fix *s*
  show $\exists\, b.\ s \downarrow b$
    using *step-converges-or-diverges*[*of s*]
  proof (*elim disjE exE conjE*)
    fix s'
    assume $s \rightarrow^* s'$ **and** *finished* $(\rightarrow)$ s'
    thus $\exists\, b.\ s \downarrow b$
      by (*cases final* s') (*auto intro*: *state-terminates state-goes-wrong*)
  next
    assume $s \rightarrow^\infty$
    thus $\exists\, b.\ s \downarrow b$
      by (*auto intro*: *state-diverges*)
  qed
qed

3.2 Safe states

definition *safe* **where**
  safe $s \longleftrightarrow (\forall\, s'.\ step^{**}\ s\ s' \longrightarrow final\ s' \vee (\exists\, s''.\ step\ s'\ s''))$

lemma *final-safeI*: *final* $s \Longrightarrow$ *safe s*
  by (*metis final-finished finished-star safe-def*)

lemma *step-safe*: *step* $s\ s' \Longrightarrow$ *safe* $s \Longrightarrow$ *safe* s'
  by (*simp add*: *converse-rtranclp-into-rtranclp safe-def*)

lemma *steps-safe*: $step^{**}\ s\ s' \Longrightarrow$ *safe* $s \Longrightarrow$ *safe* s'
  by (*meson rtranclp-trans safe-def*)

lemma *safe-state-behaves-not-wrong*:

assumes *safe s* **and** $s \downarrow b$
shows $\neg$ *is-wrong b*
using ‹$s \downarrow b$›
proof (*cases rule*: *state-behaves.cases*)
case (*state-goes-wrong s2*)
then show *?thesis*
using ‹*safe s*› **by** (*auto simp*: *safe-def finished-def*)
qed *simp-all*

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*
unfolding *prog-behaves-def*
using *right-unique-state-behaves*[*OF right-unique-step*] *right-unique-load*

by (*auto intro*: *right-unique-OO*)

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* ⟷ (∄*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* ⟷
(∀ *i s1 s2 s1'*. *match i s1 s2* ⟶ $\mathcal{R}_1$ *s1 s1'* ⟶
(∃ *s2' i'*. $\mathcal{R}_2^{++}$ *s2 s2'* ∧ *match i' s1' s2'*) ∨ (∃ *i'*. *match i' s1' s2* ∧ *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:
∀ *i s1 s2*. *match i s1 s2* ⟶ *stuck-state step1 s1* ⟷ *stuck-state step2 s2* **and**
well-founded-order: *wfP order* **and**
sim: *simulation step1 step2 match order*
shows *match i s1 s2* ⟹ *step1 s1 s1'* ⟹
∃ *m s1'' n s2'' i'*. (*step1* ^^ *m*) *s1' s1''* ∧ (*step2* ^^ *Suc n*) *s2 s2''* ∧ *match i' s1'' s2''*
using *well-founded-order*
proof (*induction i arbitrary*: *s1 s1' rule*: *wfP-induct-rule*)
case (*less i*)
show *?case*
using *sim*[*unfolded simulation-def*, *rule-format*, *OF* ‹*match i s1 s2*› ‹*step1 s1 s1'*›]
proof (*elim disjE exE conjE*)
show ⋀*s2' i'*. *step2*$^{++}$ *s2 s2'* ⟹ *match i' s1' s2'* ⟹ *?thesis*
by (*metis Suc-pred relpowp-0-I tranclp-power*)
next
fix *i'*
assume *match i' s1' s2* **and** *order i' i*

```
    have ¬ stuck-state step1 s1
      using ‹step1 s1 s1'› stuck-state-def by metis
    hence ¬ stuck-state step2 s2
      using ‹match i s1 s2› matching-states-agree-on-stuck by metis
    hence ¬ stuck-state step1 s1'
      using ‹match i' s1' s2› matching-states-agree-on-stuck by metis

    then obtain s1'' where step1 s1' s1''
      by (metis stuck-state-def)

    obtain m s1''' n s2' i'' where
      (step1 ^^ m) s1'' s1''' and
      (step2 ^^ Suc n) s2 s2' and
      match i'' s1''' s2'
      using less.IH[OF ‹order i' i› ‹match i' s1' s2› ‹step1 s1' s1''›] by metis

    show ?thesis
    proof (intro exI conjI)
      show (step1 ^^ Suc m) s1' s1'''
        using ‹(step1 ^^ m) s1'' s1'''› ‹step1 s1' s1''› by (metis relpowp-Suc-I2)
    next
      show (step2 ^^ Suc n) s2 s2'
        using ‹(step2 ^^ Suc n) s2 s2'› .
    next
      show match i'' s1''' s2'
        using ‹match i'' s1''' s2'› .
    qed
  qed
qed

context begin

private inductive match-bisim
  for R1 :: 'a ⇒ 'a ⇒ bool and R2 :: 'b ⇒ 'b ⇒ bool and
    match :: 'c ⇒ 'a ⇒ 'b ⇒ bool and order :: 'c ⇒ 'c ⇒ bool
where
  bisim-stuck: stuck-state R1 s1 ⟹ stuck-state R2 s2 ⟹ match i s1 s2 ⟹
    match-bisim R1 R2 match order (0, 0) s1 s2 |

  bisim-steps: match i s1₀ s2₀ ⟹ R1** s1₀ s1 ⟹ (R1 ^^ Suc m) s1 s1' ⟹
    R2** s2₀ s2 ⟹ (R2 ^^ Suc n) s2 s2' ⟹ match i' s1' s2' ⟹
    match-bisim R1 R2 match order (m, n) s1 s2

theorem 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
  matching-states-agree-on-stuck:
   ∀ i s1 s2. match i s1 s2 ⟶ stuck-state step1 s1 ⟷ stuck-state step2 s2 and
  well-founded-order: 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 ⟹
    (∃ i. stuck-state step1 s1 ∧ stuck-state step2 s2 ∧ match i s1 s2) ∨
    (∃ i s1' s2'. step1^++ s1 s1' ∧ step2^++ s2 s2' ∧ match i s1' s2') and
  wfP ORDER and
   right-unique step1 ⟹ simulation step1 step2 (λi s1 s2. MATCH i s1 s2)
ORDER and
   right-unique step2 ⟹ simulation step2 step1 (λi s2 s1. MATCH i s1 s2)
ORDER
proof −
 define MATCH :: nat × nat ⇒ 's1 ⇒ 's2 ⇒ bool where
  MATCH = match-bisim step1 step2 match order

 define ORDER :: nat × nat ⇒ nat × nat ⇒ bool where
  ORDER = lex-prodp ((<) :: nat ⇒ nat ⇒ bool) ((<) :: nat ⇒ nat ⇒ bool)

 have MATCH-if-match: ⋀i s1 s2. match i s1 s2 ⟹ ∃ j. MATCH j s1 s2
 proof −
  fix i s1 s2
  assume match i s1 s2

  have stuck-state step1 s1 ⟷ stuck-state step2 s2
   using ‹match i s1 s2› matching-states-agree-on-stuck by metis
  hence (stuck-state step1 s1 ∧ stuck-state step2 s2) ∨ (∃ s1' s2'. step1 s1 s1' ∧
step2 s2 s2')
   by (metis stuck-state-def)
  thus ∃ j. MATCH j s1 s2
  proof (elim disjE conjE exE)
   show stuck-state step1 s1 ⟹ stuck-state step2 s2 ⟹ ∃ j. MATCH j s1 s2
    by (metis MATCH-def ‹match i s1 s2› bisim-stuck)
  next
   fix s1' s2'
   assume step1 s1 s1' and step2 s2 s2'

   obtain m n s1'' s2'' i' where
    (step1 ^^ m) s1' s1'' and
    (step2 ^^ Suc n) s2 s2'' and
    match i' s1'' s2''
    using finite-progress[OF assms ‹match i s1 s2› ‹step1 s1 s1'›] by metis
```

```
    show ∃j. MATCH j s1 s2
    proof (intro exI)
      show MATCH (m, n) s1 s2
        unfolding MATCH-def
      proof (rule bisim-steps)
        show match i s1 s2
          using ‹match i s1 s2› .
      next
        show step1** s1 s1
          by simp
      next
        show (step1 ^^ Suc m) s1 s1''
          using ‹step1 s1 s1'› ‹(step1 ^^ m) s1' s1''› by (metis relpowp-Suc-I2)
      next
        show step2** s2 s2
          by simp
      next
        show (step2 ^^ Suc n) s2 s2''
          using ‹(step2 ^^ Suc n) s2 s2''› .
      next
        show match i' s1'' s2''
          using ‹match i' s1'' s2''› .
      qed
    qed
  qed
qed

show thesis
proof (rule that)
  show ⋀i s1 s2. match i s1 s2 ⟹ ∃j. MATCH j s1 s2
    using MATCH-if-match .
next
  fix j :: nat × nat and s1 :: 's1 and s2 :: 's2
  assume MATCH j s1 s2
  thus (∃i. stuck-state step1 s1 ∧ stuck-state step2 s2 ∧ match i s1 s2) ∨
    (∃i s1' s2'. step1++ s1 s1' ∧ step2++ s2 s2' ∧ match i s1' s2')
    unfolding MATCH-def
  proof (cases step1 step2 match order j s1 s2 rule: match-bisim.cases)
    case (bisim-stuck i)
    thus ?thesis
      by blast
  next
    case (bisim-steps m s1' i s2₀ n₁ n₂ s2' i')
    hence ∃i s1' s2'. step1++ s1 s1' ∧ step2++ s2 s2' ∧ match i s1' s2'
      by (metis tranclp-power zero-less-Suc)
    thus ?thesis ..
  qed
next
```

```
    show wfP ORDER
      unfolding ORDER-def
      using lex-prodp-wfP wfP-less well-founded-order by metis
  next
    assume right-unique step1
    show simulation step1 step2 MATCH ORDER
      unfolding simulation-def
    proof (intro allI impI)
      fix j :: nat × nat and s1 s1' :: 's1 and s2 :: 's2
      assume MATCH j s1 s2 and step1 s1 s1'
      hence match-bisim step1 step2 match order j s1 s2
        unfolding MATCH-def by metis
      thus (∃ s2' j'. step2^++ s2 s2' ∧ MATCH j' s1' s2') ∨ (∃ j'. MATCH j' s1'
s2 ∧ ORDER j' j)
      proof (cases step1 step2 match order j s1 s2 rule: match-bisim.cases)
        case (bisim-stuck i)
        hence False
          using ‹step1 s1 s1'› stuck-state-def by metis
        thus ?thesis ..
      next
        case (bisim-steps i s1_0 s2_0 m s1'' n s2' i')

        have (step1 ^^ m) s1' s1''
          using ‹step1 s1 s1'› ‹(step1 ^^ Suc m) s1 s1''› ‹right-unique step1›
          by (metis relpowp-Suc-D2 right-uniqueD)

        show ?thesis
        proof (cases m)
          case 0
          hence s1'' = s1'
            using ‹(step1 ^^ m) s1' s1''› by simp

          have step2^++ s2 s2'
            using ‹(step2 ^^ Suc n) s2 s2'› by (metis tranclp-power zero-less-Suc)

          moreover have ∃ j'. MATCH j' s1' s2'
            using ‹match i' s1'' s2'› ‹s1'' = s1'› MATCH-if-match by metis

          ultimately show ?thesis
            by metis
        next
          case (Suc m')
          define j' where
            j' = (m', n)

          have MATCH j' s1' s2
            unfolding MATCH-def j'-def
          proof (rule match-bisim.bisim-steps)
            show match i s1_0 s2_0
```

```
          using ‹match i s1₀ s2₀› .
        next
          show step1** s1₀ s1'
            using ‹step1** s1₀ s1› ‹step1 s1 s1'› by auto
        next
          show (step1 ^^ Suc m') s1' s1''
            using ‹(step1 ^^ m) s1' s1''› ‹m = Suc m'› by argo
        next
          show step2** s2₀ s2
            using ‹step2** s2₀ s2› .
        next
          show (step2 ^^ Suc n) s2 s2'
            using ‹(step2 ^^ Suc n) s2 s2'› .
        next
          show match i' s1'' s2'
            using ‹match i' s1'' s2'› .
        qed

        moreover have ORDER j' j
          unfolding ORDER-def ‹j' = (m', n)› ‹j = (m, n)› ‹m = Suc m'›
          by (simp add: lex-prodp-def)

        ultimately show ?thesis
          by metis
      qed
    qed
  qed
 next
  assume right-unique step2
  show simulation step2 step1 (λi s2 s1. MATCH i s1 s2) ORDER
    unfolding simulation-def
  proof (intro allI impI)
    fix j :: nat × nat and s1 :: 's1 and s2 s2' :: 's2
    assume MATCH j s1 s2 and step2 s2 s2'
    hence match-bisim step1 step2 match order j s1 s2
      unfolding MATCH-def by metis
     thus (∃ s1' j'. step1⁺⁺ s1 s1' ∧ MATCH j' s1' s2') ∨ (∃ j'. MATCH j' s1
s2' ∧ ORDER j' j)
    proof (cases step1 step2 match order j s1 s2 rule: match-bisim.cases)
      case (bisim-stuck i)
      hence stuck-state step2 s2
        by argo
      hence False
        using ‹step2 s2 s2'› stuck-state-def by metis
      thus ?thesis ..
    next
      case (bisim-steps i s1₀ s2₀ m s1' n s2'' i')
      show ?thesis
      proof (cases n)
```

```
  case 0
  hence s2'' = s2'
    using ‹step2 s2 s2'› ‹(step2 ^^ Suc n) s2 s2''› ‹right-unique step2›
    by (metis One-nat-def relpowp-1 right-uniqueD)

  have step1^++ s1 s1'
    using ‹(step1 ^^ Suc m) s1 s1'›
    by (metis less-Suc-eq-0-disj tranclp-power)

  moreover have ∃j'. MATCH j' s1' s2'
    using ‹match i' s1' s2''› ‹s2'' = s2'› MATCH-if-match by metis

  ultimately show ?thesis
    by metis
next
  case (Suc n')

  define j' where
    j' = (m, n')

  have MATCH j' s1 s2'
    unfolding MATCH-def j'-def
  proof (rule match-bisim.bisim-steps)
    show match i s1₀ s2₀
      using ‹match i s1₀ s2₀› .
  next
    show step1** s1₀ s1
      using ‹step1** s1₀ s1› .
  next
    show (step1 ^^ Suc m) s1 s1'
      using ‹(step1 ^^ Suc m) s1 s1'› .
  next
    show step2** s2₀ s2'
      using ‹step2** s2₀ s2› ‹step2 s2 s2'› by auto
  next
    show (step2 ^^ Suc n') s2' s2''
      using ‹step2 s2 s2'› ‹(step2 ^^ Suc n) s2 s2''› ‹right-unique step2›
      by (metis Suc relpowp-Suc-D2 right-uniqueD)
  next
    show match i' s1' s2''
      using ‹match i' s1' s2''› .
  qed

  moreover have ORDER j' j
    unfolding ORDER-def ‹j' = (m, n')› ‹j = (m, n)› ‹n = Suc n'›
    by (simp add: lex-prodp-def)

  ultimately show ?thesis
    by metis
```

qed
qed
qed
qed
qed

end

definition *safe-state* **where**
safe-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-state* $\mathcal{R}$ $\mathcal{F}$ *s* $\Longrightarrow$ *safe-state* $\mathcal{R}$ $\mathcal{F}$ *s'*
by (*simp add*: *converse-rtranclp-into-rtranclp safe-state-def*)

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'*
by (*simp add*: *rtranclp-induct step-preserves-safe-state*)

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'*
by (*simp add*: *step-preserves-safe-state tranclp-induct*)

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*
by (*metis* (*full-types*) *assms converse-rtranclpE right-uniqueD safe-state-def*)

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*
by (*smt* (*verit*, *best*) *assms converse-rtranclp-induct safe-state-before-step-if-safe-state-after*)

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*
by (*meson assms safe-state-before-rtranclp-step-if-safe-state-after tranclp-into-rtranclp*)

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*
using *assms* **by** (*simp add*: *safe-state-def*)

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:
$\forall\, i\ s1\ s2.\ match\ i\ s1\ s2 \longrightarrow safe\text{-}state\ step1\ final1\ s1 \wedge safe\text{-}state\ step2\ final2\ s2$
shows $\forall\, i\ s1\ s2.\ match\ i\ s1\ s2 \longrightarrow stuck\text{-}state\ step1\ s1 \longleftrightarrow stuck\text{-}state\ step2\ s2$
using *assms* **by** (*metis rtranclp.rtrancl-refl safe-state-def stuck-state-def*)

corollary *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
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**
order-well-founded: *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
$\bigwedge i\ s1\ s2.\ match\ i\ s1\ s2 \Longrightarrow (\exists\, j.\ MATCH\ j\ s1\ s2)$
$\bigwedge j\ s1\ s2.\ MATCH\ j\ s1\ s2 \Longrightarrow final1\ s1 \longleftrightarrow final2\ s2$ **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 −
have *matching-states-agree-on-stuck*:
$\forall\, i\ s1\ s2.\ match\ i\ s1\ s2 \longrightarrow stuck\text{-}state\ step1\ s1 \longleftrightarrow stuck\text{-}state\ step2\ s2$
using *matching-states-agree-on-stuck-if-they-agree-on-final*[*OF final1-stuck final2-stuck*
matching-states-agree-on-final matching-states-are-safe] **.**

obtain
MATCH :: $nat \times nat \Rightarrow 's1 \Rightarrow 's2 \Rightarrow bool$ **and**
ORDER :: $nat \times nat \Rightarrow nat \times nat \Rightarrow bool$
where

```
    MATCH-if-match: ⋀i s1 s2. match i s1 s2 ⟹ ∃j. MATCH j s1 s2 and
    MATCH-spec: ⋀j s1 s2. MATCH j s1 s2 ⟹
      (∃i. stuck-state step1 s1 ∧ stuck-state step2 s2 ∧ match i s1 s2) ∨
      (∃i s1' s2'. step1^++ s1 s1' ∧ step2^++ s2 s2' ∧ match i s1' s2') and
    wfP ORDER and
    simulation step1 step2 MATCH ORDER and
    simulation step2 step1 (λi s2 s1. MATCH i s1 s2) ORDER
    using ‹right-unique step1› ‹right-unique step2›
    using lift-strong-simulation-to-bisimulation[
        OF matching-states-agree-on-stuck
        order-well-founded sim]
    by (smt (verit))

  show thesis
  proof (rule that)
    show ⋀i s1 s2. match i s1 s2 ⟹ ∃j. MATCH j s1 s2
      using MATCH-if-match .
  next
    show ⋀j s1 s2. MATCH j s1 s2 ⟹ final1 s1 = final2 s2
      using MATCH-spec
     by (metis converse-tranclpE final1-stuck final2-stuck matching-states-agree-on-final)
  next
    show ⋀j s1 s2. MATCH j s1 s2 ⟹ stuck-state step1 s1 = stuck-state step2 s2
      using MATCH-spec
      by (metis stuck-state-def tranclpD)
  next
    fix j s1 s2
    assume MATCH j s1 s2
    then show safe-state step1 final1 s1 ∧ safe-state step2 final2 s2
      apply −
      apply (drule MATCH-spec)
      apply (elim disjE exE conjE)
      using matching-states-are-safe apply metis
      using safe-state-before-tranclp-step-if-safe-state-after
      by (metis assms(1) assms(2) matching-states-are-safe)
  next
    show wfP ORDER
      using ‹wfP ORDER› .
  next
    show simulation step1 step2 (λi s1 s2. MATCH i s1 s2) ORDER
      using ‹simulation step1 step2 (λi s1 s2. MATCH i s1 s2) ORDER› .
  next
    show simulation step2 step1 (λi s2 s1. MATCH i s1 s2) ORDER
      using ‹simulation step2 step1 (λi s2 s1. MATCH i s1 s2) ORDER› .
  qed
qed

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 (⊏)
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 i s1 s2 ⟹ *step2 s2 s2'* ⟹
(∃ *i' s1'*. *step1*$^{++}$ *s1 s1'* ∧ *match i' s1' s2'*) ∨ (∃ *i'*. *match i' s1 s2'* ∧ *i'* ⊏ *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 (⊏) ordering.

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 match-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
```

5.1.1 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 s2 b2 ⟹ ¬is-wrong b2 ⟹ match i s1 s2 ⟹
    ∃ b1 i'. L1.state-behaves s1 b1 ∧ rel-behaviour (match i') b1 b2
proof(induction rule: L2.state-behaves.cases)
  case (state-terminates s2 s2')
  then obtain i' s1' where L1.eval s1 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.state-behaves s1 (Terminates s1')
    using L1.final-finished
    by (simp add: L1.state-terminates ‹L1.eval s1 s1'›)
  moreover have rel-behaviour (match i') (Terminates s1') b2
    by (simp add: ‹match i' s1' s2'› state-terminates.hyps)
  ultimately show ?case by blast
next
  case (state-diverges s2)
  then show ?case
    using match-inf L1.state-diverges by fastforce
next
  case (state-goes-wrong s2 s2')
  then show ?case using ‹¬is-wrong b2› by simp
qed

end
```

5.2 Forward simulation

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

lemma *lift-simulation-plus*:
  step1^{++} *s1 s1'* ⟹ *match i s1 s2* ⟹
    (∃ *i' s2'*. *step2*$^{++}$ *s2 s2'* ∧ *match i' s1' s2'*) ∨
    (∃ *i'*. *match i' s1' s2* ∧ *order*$^{++}$ *i' i*)
proof (*induction s1' arbitrary*: *i s2 rule*: *tranclp-induct*)
  case (*base s1'*)
  with *simulation*[*OF base.prems*(*1*) *base.hyps*(*1*)] **show** *?case*
    by *auto*
next
  case (*step s1' s1''*)
  show *?case*
    using *step.IH*[*OF* ‹*match i s1 s2*›]
  proof (*elim disjE exE conjE*)
    fix *i' s2'*
    assume *step2*$^{++}$ *s2 s2'* **and** *match i' s1' s2'*

    have (∃ *i' s2'a*. *step2*$^{++}$ *s2' s2'a* ∧ *match i' s1'' s2'a*) ∨ (∃ *i'a*. *match i'a s1'' s2'* ∧ *i'a* $\sqsubset$ *i'*)
      using *simulation*[*OF* ‹*match i' s1' s2'*› ‹*step1 s1' s1''*›] **.**
    thus *?thesis*
    proof (*elim disjE exE conjE*)
      fix *i'' s2''*
      assume *step2*$^{++}$ *s2' s2''* **and** *match i'' s1'' s2''*
      thus *?thesis*
        using *tranclp-trans*[*OF* ‹*step2*$^{++}$ *s2 s2'*›] **by** *auto*
    next
      fix *i''*

```
    assume match i'' s1'' s2' and i'' ⊏ i'
    thus ?thesis
      using ‹step2^++ s2 s2'› by auto
  qed
 next
  fix i'
  assume match i' s1' s2 and (⊏)^++ i' i
  then show ?thesis
    using simulation[OF ‹match i' s1' s2› ‹step1 s1' s1''›] by auto
 qed
qed

lemma lift-simulation-eval:
 L1.eval s1 s1' ⟹ match i s1 s2 ⟹ ∃ i' s2'. L2.eval s2 s2' ∧ match i' s1' s2'
proof(induction s1 arbitrary: i s2 rule: converse-rtranclp-induct)
 case (base s2)
 thus ?case by auto
next
 case (step s1 s1'')
 show ?case
  using simulation[OF ‹match i s1 s2› ‹step1 s1 s1''›]
 proof (elim disjE exE conjE)
  fix i' s2'
  assume step2^++ s2 s2' and match i' s1'' s2'
  thus ?thesis
   by (auto intro: rtranclp-trans dest!: tranclp-into-rtranclp step.IH)
 next
  fix i'
  assume match i' s1'' s2 and i' ⊏ i
  thus ?thesis
   by (auto intro: step.IH)
 qed
qed

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

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(*induction rule*: *L1.state-behaves.cases*)
  case (*state-terminates s1 s1'*)
  then obtain *i' s2'* **where** *steps-s2*: *L2.eval s2 s2'* **and** *match-s1'-s2'*: *match i' s1' s2'*
    using *lift-simulation-eval* **by** *blast*
  hence *final2 s2'*
    by (*auto intro*: *state-terminates.hyps match-final*)
  hence *L2.state-behaves s2* (*Terminates s2'*)
    using *L2.final-finished L2.state-terminates*[*OF steps-s2*]
    by *simp*
  moreover have *rel-behaviour* (*match i'*) *b1* (*Terminates s2'*)
    by (*simp add*: ‹*match i' s1' s2'*› *state-terminates.hyps*)
  ultimately show *?case* **by** *blast*
next
  case (*state-diverges s2*)
  then show *?case*
    using *match-inf*[*THEN L2.state-diverges*] **by** *fastforce*
next
  case (*state-goes-wrong s2 s2'*)
  then show *?case* **using** ‹$\neg$*is-wrong b1*› **by** *simp*
qed

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 −
  obtain *b1* **where** *behaves-s1*: *L1.state-behaves s1 b1*
    using *L1.left-total-state-behaves*
    by (*auto elim*: *left-totalE*)

  have *not-wrong-b1*: $\neg$ *is-wrong b1*
    by (*rule L1.safe-state-behaves-not-wrong*[*OF safe-s1 behaves-s1*])

  obtain *i'* **where** *L2.state-behaves s2 b2* **and** *rel-b1-B2*: *rel-behaviour* (*match i'*) *b1 b2*
    using *simulation-behaviour*[*OF behaves-s1 not-wrong-b1 match-s1-s2*]
    using *L2.right-unique-state-behaves*[*OF right-unique2*, *THEN right-uniqueD*]
    using *behaves-s2*
    by *auto*

show *?thesis*
using *behaves-s1 rel-b1-B2* **by** *auto*
qed

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* ⇒ *'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*

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 step2 final1 final2 ORDER MATCH
proof −
have *simulation step1 step2 match lt*
using *fsim* **unfolding** *simulation-def* **by** *metis*

obtain
MATCH :: *nat* × *nat* ⇒ *'state1* ⇒ *'state2* ⇒ *bool* **and**
ORDER :: *nat* × *nat* ⇒ *nat* × *nat* ⇒ *bool*

where
($\bigwedge i\ s1\ s2.\ match\ i\ s1\ s2 \Longrightarrow \exists j.\ MATCH\ j\ s1\ s2$) **and**
($\bigwedge j\ s1\ s2.\ MATCH\ j\ s1\ s2 \Longrightarrow final1\ s1 = final2\ s2$) **and**
($\bigwedge j\ s1\ s2.\ MATCH\ j\ s1\ s2 \Longrightarrow stuck\text{-}state\ step1\ s1 = stuck\text{-}state\ step2\ s2$)
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**
fsim': *simulation step1 step2 MATCH ORDER* **and**
bsim': *simulation step2 step1* ($\lambda i\ s2\ s1.\ MATCH\ i\ s1\ s2$) *ORDER*
using *lift-strong-simulation-to-bisimulation'*[*OF assms*(1,2) *final1-stuck final2-stuck matching-states-agree-on-final matching-states-are-safe* ‹*wfP lt*› ‹*simulation step1 step2 match lt*›]
by *blast*

have *bisimulation step1 step2 final1 final2 ORDER MATCH*
proof *unfold-locales*
show $\bigwedge i\ s1\ s2.\ MATCH\ i\ s1\ s2 \Longrightarrow final1\ s1 \Longrightarrow final2\ s2$
using ‹$\bigwedge s2\ s1\ j.\ MATCH\ j\ s1\ s2 \Longrightarrow final1\ s1 = final2\ s2$› **by** *metis*
next
show $\bigwedge i\ s1\ s2.\ MATCH\ i\ s1\ s2 \Longrightarrow final2\ s2 \Longrightarrow final1\ s1$
using ‹$\bigwedge s2\ s1\ j.\ MATCH\ j\ s1\ s2 \Longrightarrow final1\ s1 = final2\ s2$› **by** *metis*
next
show *wfP ORDER*
using ‹*wfP ORDER*› **.**
next
show $\bigwedge i\ s1\ s2\ s1'.\ MATCH\ i\ s1\ s2 \Longrightarrow step1\ s1\ s1' \Longrightarrow$
($\exists i'\ s2'.\ step2^{++}\ s2\ s2' \wedge MATCH\ i'\ s1'\ s2'$) $\vee$ ($\exists i'.\ MATCH\ i'\ s1'\ s2 \wedge ORDER\ i'\ i$)
using *fsim'* **unfolding** *simulation-def* **by** *metis*
next
show $\bigwedge i\ s1\ s2\ s2'.\ MATCH\ i\ s1\ s2 \Longrightarrow step2\ s2\ s2' \Longrightarrow$
($\exists i'\ s1'.\ step1^{++}\ s1\ s1' \wedge MATCH\ i'\ s1'\ s2'$) $\vee$ ($\exists i'.\ MATCH\ i'\ s1\ s2' \wedge ORDER\ i'\ i$)
using *bsim'* **unfolding** *simulation-def* **by** *metis*
next
show $\bigwedge s.\ final1\ s \Longrightarrow finished\ step1\ s$
by (*simp add*: *final1-stuck finished-def*)
next
show $\bigwedge s.\ final2\ s \Longrightarrow finished\ step2\ s$
by (*simp add*: *final2-stuck finished-def*)
qed

thus *thesis*
using *that* **by** *metis*
qed

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: ∀ *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*)
shows ∃ (*MATCH* :: *nat* × *nat* ⇒ *'state1* ⇒ *'state2* ⇒ *bool*)
(*ORDER* :: *nat* × *nat* ⇒ *nat* × *nat* ⇒ *bool*).
bisimulation step1 step2 final1 final2 ORDER MATCH
using *obtains-bisimulation-from-forward-simulation*[*OF assms*] **by** *metis*

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: ∀ *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**
bsim: ∀ *i s1 s2 s2'*. *match i s1 s2* ⟶ *step2 s2 s2'* ⟶
(∃ *i' s1'*. *step1*$^{++}$ *s1 s1'* ∧ *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 step2 final1 final2 ORDER MATCH
proof −
have *matching-states-agree-on-final'*: ∀ *i s2 s1*. (λ*i s2 s1*. *match i s1 s2*) *i s2 s1* ⟶ *final2 s2* ⟷ *final1 s1*
using *matching-states-agree-on-final* **by** *simp*

have *matching-states-are-safe'*:
∀ *i s2 s1*. (λ*i s2 s1*. *match i s1 s2*) *i s2 s1* ⟶ *safe-state step2 final2 s2* ∧

safe-state step1 final1 s1
using *matching-states-are-safe* **by** *simp*

have *simulation step2 step1* (λ*i s2 s1*. *match i s1 s2*) *lt*
using *bsim* **unfolding** *simulation-def* **by** *metis*

obtain
MATCH :: *nat* × *nat* ⇒ *'state2* ⇒ *'state1* ⇒ *bool* **and**
ORDER :: *nat* × *nat* ⇒ *nat* × *nat* ⇒ *bool*
where
(⋀*i s1 s2*. *match i s1 s2* ⟹ ∃*j*. *MATCH j s2 s1*) **and**
(⋀*j s1 s2*. *MATCH j s2 s1* ⟹ *final1 s1* = *final2 s2*) **and**
(⋀*j s1 s2*. *MATCH j s2 s1* ⟹ *stuck-state step1 s1* = *stuck-state step2 s2*)
and
(⋀*j s1 s2*. *MATCH j s2 s1* ⟹ *safe-state step1 final1 s1* ∧ *safe-state step2 final2 s2*) **and**
wfP ORDER **and**
fsim': *simulation step1 step2* (λ*i s1 s2*. *MATCH i s2 s1*) *ORDER* **and**
bsim': *simulation step2 step1* (λ*i s2 s1*. *MATCH i s2 s1*) *ORDER*
using *lift-strong-simulation-to-bisimulation'*[*OF assms*(*2*,*1*) *final2-stuck final1-stuck matching-states-agree-on-final' matching-states-are-safe'* ‹*wfP lt*› ‹*simulation step2 step1* (λ*i s2 s1*. *match i s1 s2*) *lt*›]
by (*smt* (*verit*))

have *bisimulation step1 step2 final1 final2 ORDER* (λ*i s1 s2*. *MATCH i s2 s1*)
proof *unfold-locales*
show ⋀*i s1 s2*. *MATCH i s2 s1* ⟹ *final1 s1* ⟹ *final2 s2*
using ‹⋀*s2 s1 j*. *MATCH j s2 s1* ⟹ *final1 s1* = *final2 s2*› **by** *metis*
next
show ⋀*i s1 s2*. *MATCH i s2 s1* ⟹ *final2 s2* ⟹ *final1 s1*
using ‹⋀*s2 s1 j*. *MATCH j s2 s1* ⟹ *final1 s1* = *final2 s2*› **by** *metis*
next
show *wfP ORDER*
using ‹*wfP ORDER*› **.**
next
show ⋀*i s1 s2 s1'*. *MATCH i s2 s1* ⟹ *step1 s1 s1'* ⟹
(∃*i' s2'*. *step2*$^{++}$ *s2 s2'* ∧ *MATCH i' s2' s1'*) ∨ (∃*i'*. *MATCH i' s2 s1'* ∧ *ORDER i' i*)
using *fsim'* **unfolding** *simulation-def* **by** *metis*
next
show ⋀*i s1 s2 s2'*. *MATCH i s2 s1* ⟹ *step2 s2 s2'* ⟹
(∃*i' s1'*. *step1*$^{++}$ *s1 s1'* ∧ *MATCH i' s2' s1'*) ∨ (∃*i'*. *MATCH i' s2' s1* ∧ *ORDER i' i*)
using *bsim'* **unfolding** *simulation-def* **by** *metis*
next
show ⋀*s*. *final1 s* ⟹ *finished step1 s*
by (*simp add*: *final1-stuck finished-def*)
next
show ⋀*s*. *final2 s* ⟹ *finished step2 s*

by (*simp add*: *final2-stuck finished-def*)
qed

thus *thesis*
using *that* **by** *metis*
qed

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: ∀ *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**
bsim: ∀ *i s1 s2 s2'*. *match i s1 s2* ⟶ *step2 s2 s2'* ⟶
(∃ *i' s1'*. *step1*$^{++}$ *s1 s1'* ∧ *match i' s1' s2'*) ∨ (∃ *i'*. *match i' s1 s2'* ∧ *lt i' i*)
shows ∃(*MATCH* :: *nat* × *nat* ⇒ *'state1* ⇒ *'state2* ⇒ *bool*)
(*ORDER* :: *nat* × *nat* ⇒ *nat* × *nat* ⇒ *bool*).
bisimulation step1 step2 final1 final2 ORDER MATCH
using *obtains-bisimulation-from-backward-simulation*[*OF assms*] **by** *metis*

5.4 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: $\bigwedge s1\ s2\ s2'.\ match\ s1\ s2 \Longrightarrow step1\ s1\ s1' \Longrightarrow (\exists s2'.\ step2\ s2\ s2' \wedge match\ s1'\ s2')$ **and**
match: *match s1 s2* **and**
step: *step1 s1 s1'*
shows $\exists s2'.\ step2^{++}\ s2\ s2' \wedge match\ s1'\ s2'$
proof −
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**
$(f \Leftarrow g)\ x \equiv Option.bind\ (g\ x)\ f$

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

fun *option-comp-pow* :: *nat* ⇒ *'a* ⇒ *'a option* **where**
option-comp-pow 0 = *(*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* ⇒ *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* ⟹ *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 −
obtain *s2* **where** *load2 p2 s2* **and** *L2.state-behaves s2 b2*
using *behaves L2.prog-behaves-def* **by** *auto*
obtain *i s1* **where** *load1 p1 s1 match i s1 s2*
using *compile-load*[*OF compiles* ‹*load2 p2 s2*›]
by *auto*
then show *?thesis*
using *simulation-behaviour*[*OF* ‹*L2.state-behaves s2 b2*› *not-wrong* ‹*match i s1 s2*›]
by (*auto simp*: *L1.prog-behaves-def*)
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 s3
    assume
      compile: (compile2 ⇐ compile1) p1 = Some p3 and
      load: load3 p3 s3
    obtain p2 where c1: compile1 p1 = Some p2 and c2: compile2 p2 = Some
p3
      using compile by (auto simp: bind-eq-Some-conv option-comp-def)
    obtain s2 i' where l2: load2 p2 s2 and match2 i' s2 s3
      using assms(2)[THEN compiler.compile-load, OF c2 load]
      by auto
    moreover obtain s1 i where load1 p1 s1 and match1 i s1 s2
      using assms(1)[THEN compiler.compile-load, OF c1 l2]
      by auto
    ultimately show ∃ s1 i. load1 p1 s1 ∧ 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 back-
ward-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 s2
      assume
        option-comp-pow compile (Suc 0) p1 = Some p2 and
        load p2 s2
      thus ∃ s1 i. load p1 s1 ∧ 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 s3
      assume
        option-comp-pow compile  (Suc (Suc n')) p1 = Some p3 and
        load p3 s3
      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)
      obtain s2 i' where load p2 s2 and rel-comp-pow match i' s2 s3
        using compiler.compile-load[OF 3.IH comp-IH ‹load p3 s3›]
        by auto
      moreover obtain s1 i where load p1 s1 and match i s1 s2
        using compiler.compile-load[OF assms comp ‹load p2 s2›]
        by auto
      moreover have rel-comp-pow match (i # i') s1 s3
        using ‹rel-comp-pow match i' s2 s3› ‹match i s1 s2› rel-comp-pow.elims(2)
by fastforce
      ultimately show ∃ s1 i. load p1 s1 ∧ rel-comp-pow match i s1 s3
        by blast
    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
$m :: 'a \Rightarrow nat$ **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'
$)$
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 $(x', x) \in$ *measure* m **by** *simp*
qed

end

lemma *fixpoint-to-comp-pow*:
fixpoint $m\ f\ x = y \Longrightarrow \exists\, n.$ *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:
 ∃ 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 s2
  assume fixpoint m compile p1 = Some p2 and load p2 s2
  obtain n where fixpoint m compile p1 = option-comp-pow compile n p1
    using fixpoint-eq-comp-pow by metis

  thus ∃ s1 i. load p1 s1 ∧ rel-comp-pow match i s1 s2
    using ‹fixpoint m compile p1 = Some p2› assms compiler.compile-load com-
piler-composition-pow
    using ‹load p2 s2› 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.