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 $R :: 'a \Rightarrow 'a \Rightarrow bool$ (**infix** $\sqsubset$ *70*)
assumes
wf: *wfP* ($\sqsubset$)
begin

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

end

1.1 Lexicographic product

context
fixes
$r1 :: 'a \Rightarrow 'a \Rightarrow bool$ **and**
$r2 :: 'b \Rightarrow 'b \Rightarrow bool$
begin

definition $lex\text{-}prodp :: 'a \times 'b \Rightarrow 'a \times 'b \Rightarrow bool$ **where**
$lex\text{-}prodp\ x\ y \equiv r1\ (fst\ x)\ (fst\ y) \vee fst\ x = fst\ y \wedge r2\ (snd\ x)\ (snd\ y)$

lemma *lex-prodp-lex-prod*:
shows $lex\text{-}prodp\ x\ y \longleftrightarrow (x, y) \in lex\text{-}prod\ \{\ (x, y).\ r1\ x\ y\ \}\ \{\ (x, y).\ r2\ x\ y\ \}$
⟨*proof*⟩

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

wfP r2
shows *wfP lex-prodp*
⟨*proof*⟩

end

lemma *lex-prodp-well-founded*:
assumes
well-founded r1 **and**
well-founded r2
shows *well-founded* (*lex-prodp r1 r2*)
⟨*proof*⟩

1.2 Lexicographic list

context
fixes *order* :: *'a* ⇒ *'a* ⇒ *bool*
begin

inductive *lexp* :: *'a list* ⇒ *'a list* ⇒ *bool* **where**
lexp-head: *order x y* ⟹ *length xs* = *length ys* ⟹ *lexp* (*x* # *xs*) (*y* # *ys*) |
lexp-tail: *lexp xs ys* ⟹ *lexp* (*x* # *xs*) (*x* # *ys*)

end

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

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

lemma *lex-list-wfP*: *wfP order* ⟹ *wfP* (*lexp order*)
⟨*proof*⟩

lemma *lex-list-well-founded*:
assumes *well-founded order*
shows *well-founded* (*lexp order*)
⟨*proof*⟩

end

2 Infinitely Transitive Closure

theory *Inf*
imports *Well-founded*
begin

coinductive *inf* :: (*'a* ⇒ *'a* ⇒ *bool*) ⇒ *'a* ⇒ *bool* **for** *r* **where**
inf-step: *r x y* ⟹ *inf r y* ⟹ *inf r x*

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

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

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

lemma *step-inf*:
assumes
deterministic: $\bigwedge x\ y\ z.\ r\ x\ y \Longrightarrow r\ x\ z \Longrightarrow y = z$
shows $r\ x\ y \Longrightarrow inf\ r\ x \Longrightarrow inf\ r\ y$
⟨*proof*⟩

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*⟩

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*⟩

locale *semantics* =
fixes
step :: $'state \Rightarrow 'state \Rightarrow bool$ (**infix** $\rightarrow$ *50*) **and**
final :: $'state \Rightarrow bool$

assumes
final-finished: *final s* $\Longrightarrow$ *finished step s*
begin

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

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

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

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

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

lemma *finished-inf*: *s* $\rightarrow^\infty$ $\Longrightarrow$ $\neg$ *finished step s*
⟨*proof*⟩

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

3.1 Behaviour of a dynamic execution

inductive *behaves* :: *'state* $\Rightarrow$ *'state behaviour* $\Rightarrow$ *bool* (**infix** $\Downarrow$ *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*⟩

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 $\Longrightarrow$ *final2 s2* $\Longrightarrow$ *final1 s1* **and**
simulation:
match i1 s1 s2 $\Longrightarrow$ *step2 s2 s2'* $\Longrightarrow$
$(\exists\, i2\ s1'.\ step1^{++}\ s1\ s1' \wedge match\ i2\ s1'\ s2') \vee (\exists\, i2.\ match\ i2\ s1\ s2' \wedge i2 \sqsubset i1)$
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 ($\sqsubset$) ordering.

end

locale *forward-simulation* =
L1: *semantics step1 final1* +
L2: *semantics step2 final2* +
well-founded ($\sqsubset$)
for
step1 :: *'state1* $\Rightarrow$ *'state1* $\Rightarrow$ *bool* **and**
step2 :: *'state2* $\Rightarrow$ *'state2* $\Rightarrow$ *bool* **and**
final1 :: *'state1* $\Rightarrow$ *bool* **and**
final2 :: *'state2* $\Rightarrow$ *bool* **and**
order :: *'index* $\Rightarrow$ *'index* $\Rightarrow$ *bool* (**infix** $\sqsubset$ *70*) +
fixes
match :: *'index* $\Rightarrow$ *'state1* $\Rightarrow$ *'state2* $\Rightarrow$ *bool*
assumes
match-final:
match i s1 s2 $\Longrightarrow$ *final1 s1* $\Longrightarrow$ *final2 s2* **and**
simulation:
match i1 s1 s2 $\Longrightarrow$ *step1 s1 s1'* $\Longrightarrow$
$(\exists\, i'\ s2'.\ step2^{++}\ s2\ s2' \wedge match\ i'\ s1'\ s2') \vee (\exists\, i'.\ match\ i'\ s1\ s2' \wedge i' \sqsubset i1)$

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

match :: *'index* ⇒ *'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*⟩

lemma *lift-simulation-eval*:
L2.eval s2 s2' ⟹ *match i1 s1 s2* ⟹ ∃ *i2 s1'*. *L1.eval s1 s1'* ∧ *match i2 s1' s2'*
⟨*proof*⟩

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

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 ⟹ ¬*is-wrong* b_2 ⟹ *match i* s_1 s_2 ⟹
∃ b_1 *i'*. *L1.behaves* s_1 b_1 ∧ *rel-behaviour* (*match i'*) b_1 b_2
⟨*proof*⟩

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*⟩

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*⟩

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

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

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

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

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

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

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

end

6 Compiler Between Static Representations

theory *Compiler*

imports *Language Simulation*
begin

definition *option-comp* :: $('a \Rightarrow 'b\ option) \Rightarrow ('c \Rightarrow 'a\ option) \Rightarrow 'c \Rightarrow 'b\ option$
(**infix** $\Leftarrow$ *50*) **where**
$(f \Leftarrow g)\ x \equiv Option.bind\ (g\ x)\ f$

context
fixes $f :: ('a \Rightarrow 'a\ option)$
begin

fun *option-comp-pow* :: $nat \Rightarrow 'a \Rightarrow 'a\ option$ **where**
option-comp-pow 0 $= (\lambda\text{-}.\ None)$ |
option-comp-pow $(Suc\ 0) = f$ |
option-comp-pow $(Suc\ n) = (option\text{-}comp\text{-}pow\ n \Leftarrow f)$

end

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

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

6.1 Preservation of behaviour

corollary *behaviour-preservation*:
assumes
compiles: *compile p1* $=$ *Some p2* **and**
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*⟩

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*⟩

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

end

7 Fixpoint of Converging Program Transformations

theory *Fixpoint*
imports *Compiler*
begin

context
fixes
m :: *'a* ⇒ *nat* **and**
f :: *'a* ⇒ *'a option*
begin

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

end

lemma *fixpoint-to-comp-pow*:
fixpoint m f x = y $\Longrightarrow$ $\exists\, n.$ *option-comp-pow f n x = y*
⟨*proof*⟩

lemma *fixpoint-eq-comp-pow*:
$\exists\, n.$ *fixpoint m f x = option-comp-pow f n x*
⟨*proof*⟩

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

end

References

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