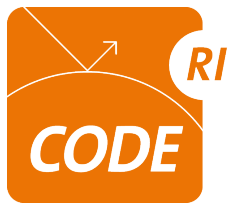
A Generic Framework for Verified Compilers Using Isabelle/HOL's Locales

Martin Desharnais Stefan Brunthaler



Research Institute
Cyber Defence
Universität der Bundeswehr München

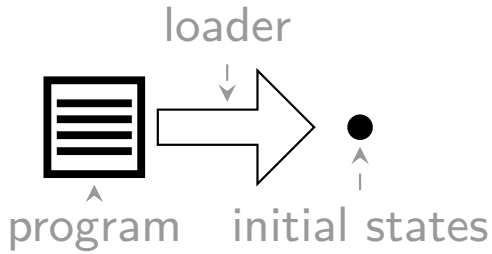
Isabelle Workshop 2020

30th June 2020

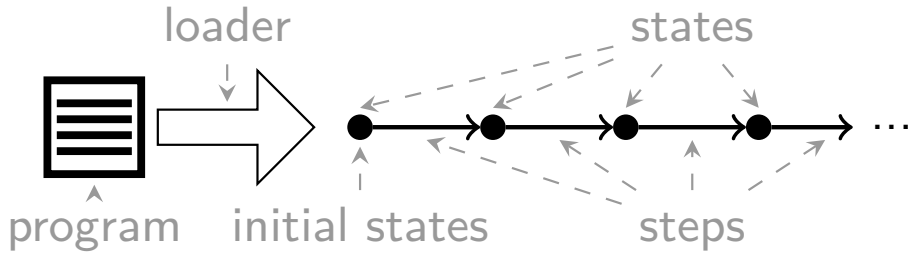
Language semantics

program

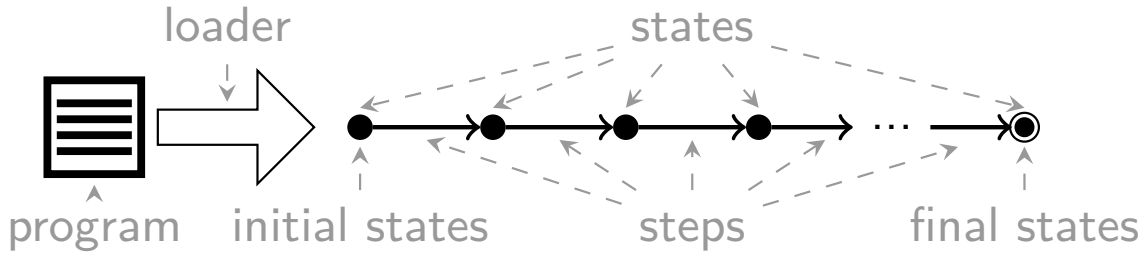
Language semantics



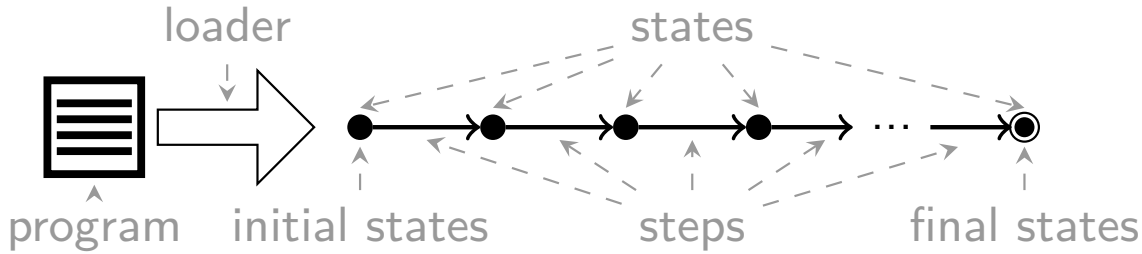
Language semantics



Language semantics

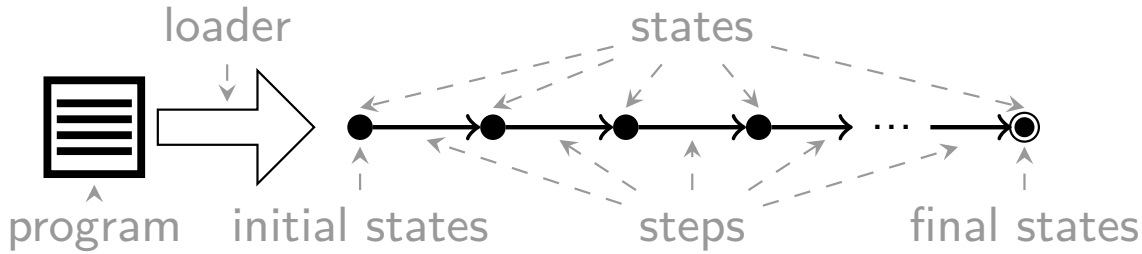


Language semantics



locale semantics =
  fixes
    step :: *'state* ⇒ *'state* ⇒ bool **and**
    final :: *'state* ⇒ bool

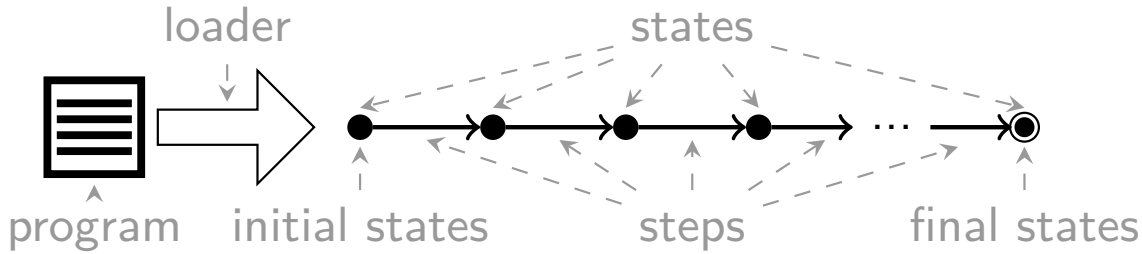
Language semantics



locale semantics =
  fixes
    step :: *'state* ⇒ *'state* ⇒ bool **and**
    final :: *'state* ⇒ bool

parameters

Language semantics

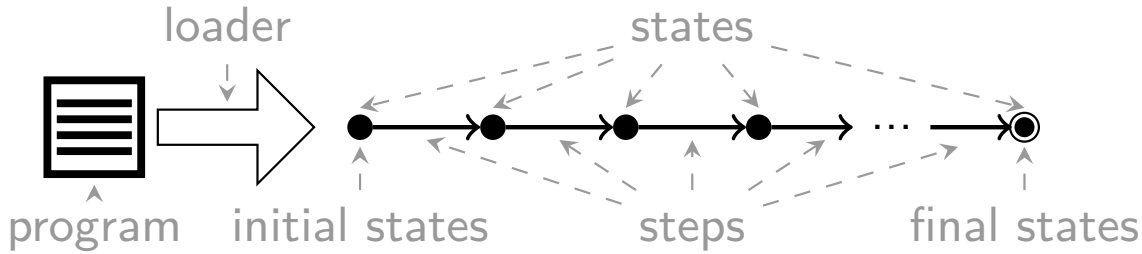


type variables

locale semantics =
  fixes
    step :: *'state* ⇒ *'state* ⇒ bool **and**
    final :: *'state* ⇒ bool

parameters

Language semantics



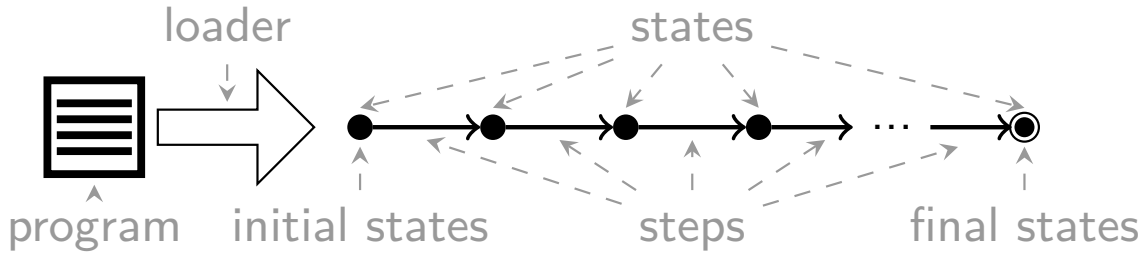
type variables

locale semantics =
  fixes
    step :: *'state* ⇒ *'state* ⇒ bool **and**
    final :: *'state* ⇒ bool

parameters

locale language =
  semantics → ◉ **for**
    → **and** ◉ :: *'state* ⇒ bool +
  fixes load :: *'prog* ⇒ *'state* option

Language semantics



type variables

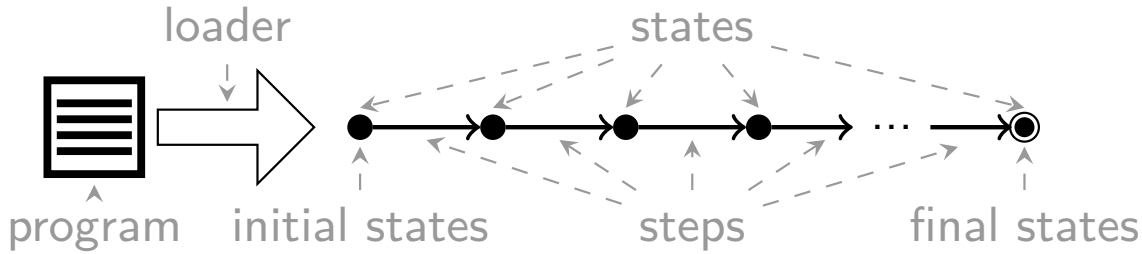
locale semantics =
  fixes
    step :: *'state* ⇒ *'state* ⇒ bool **and**
    final :: *'state* ⇒ bool

parameters

locale language =
  semantics → ◉ **for**
    → **and** ◉ :: *'state* ⇒ bool +
  fixes load :: *'prog* ⇒ *'state* option

imports

Language semantics



type variables

locale semantics =
  fixes
    step :: *'state* ⇒ *'state* ⇒ bool **and**
    final :: *'state* ⇒ bool

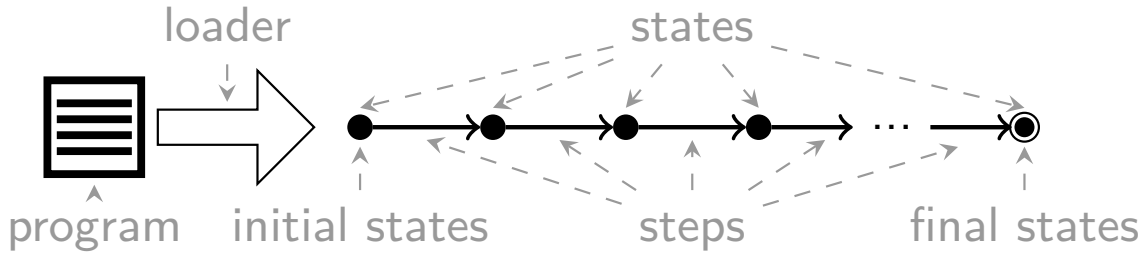
parameters

locale language =
  semantics ⟶ ◉ **for**
    ⟶ **and** ◉ :: *'state* ⇒ bool +
  fixes load :: *'prog* ⇒ *'state* option

predicate

imports

Language semantics



type variables

locale semantics =
  fixes
    step :: *'state* ⇒ *'state* ⇒ bool **and**
    final :: *'state* ⇒ bool

parameters

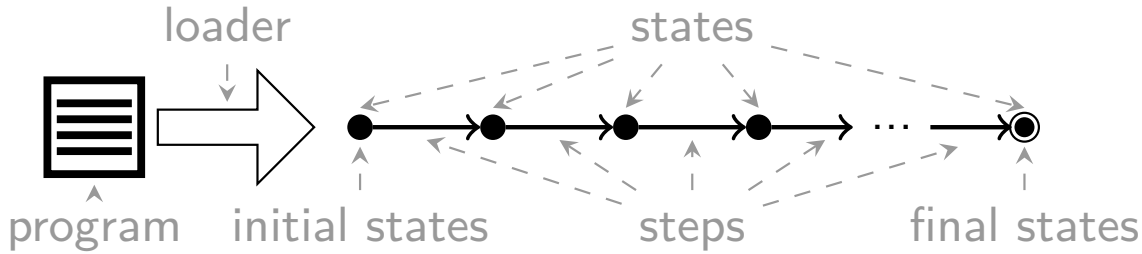
predicate

locale language =
  semantics ⟶ ◉ **for**
    ⟶ **and** ◉ :: *'state* ⇒ bool +
  fixes load :: *'prog* ⇒ *'state* option

imports

arbitrary bound variables

Language semantics



type variables

locale semantics =
  fixes
    step :: *'state* ⇒ *'state* ⇒ bool **and**
    final :: *'state* ⇒ bool

parameters

predicate

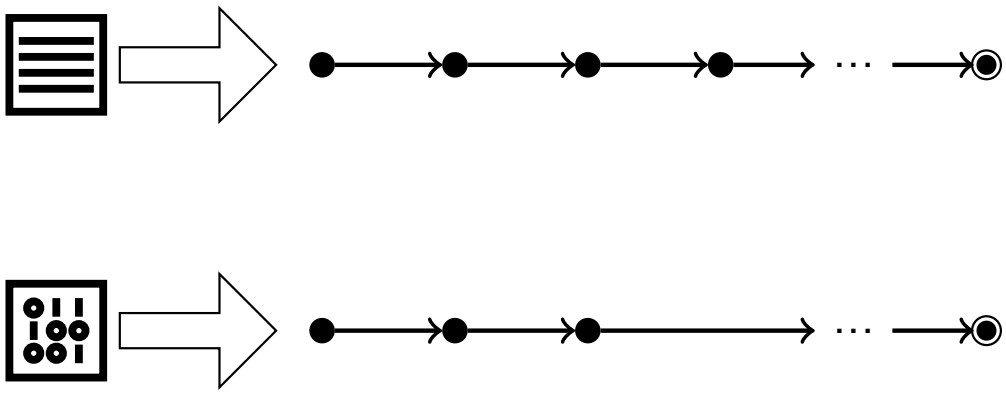
locale language =
  semantics ⟶ ◉ **for**
    ⟶ **and** ◉ :: *'state* ⇒ bool +
  fixes load :: *'prog* ⇒ *'state* option

imports

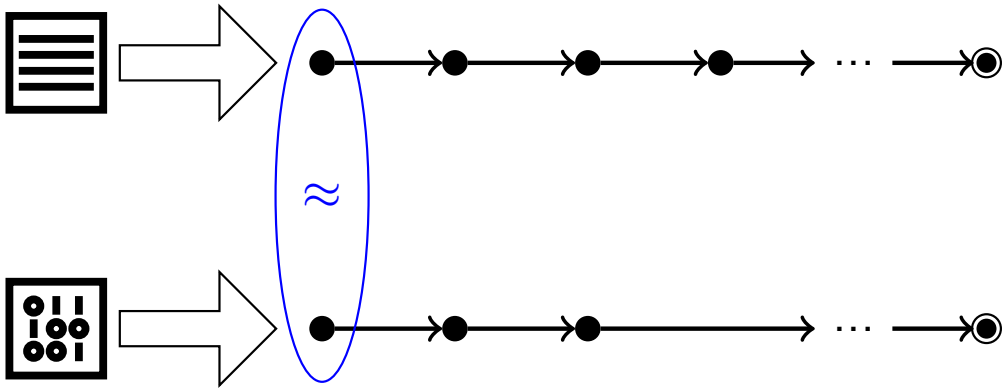
arbitrary bound variables

new parameters

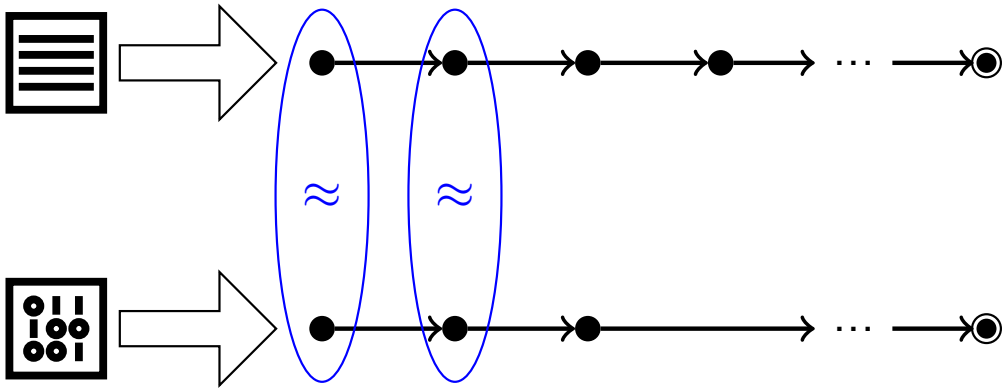
Simulation



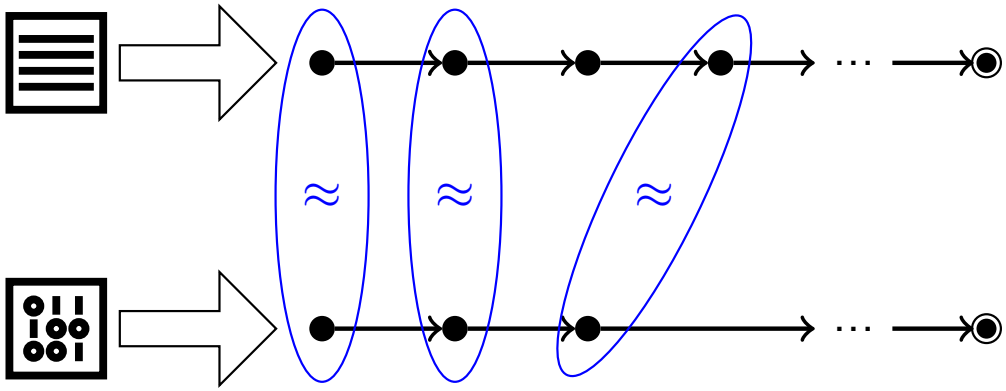
Simulation



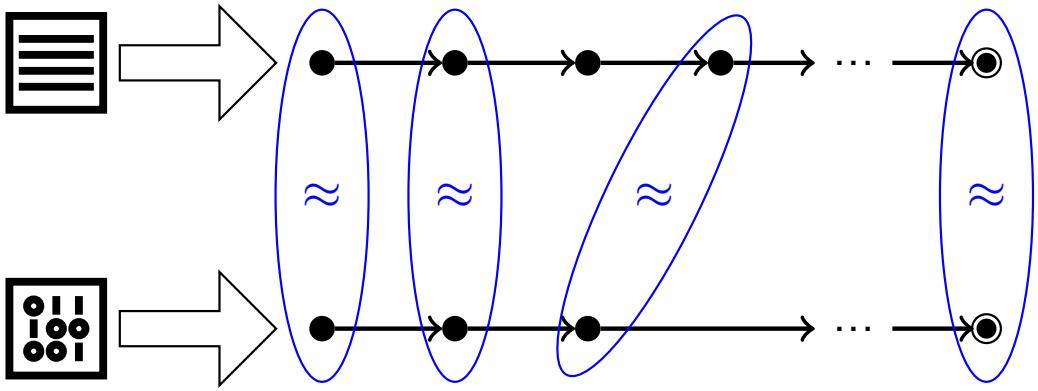
Simulation



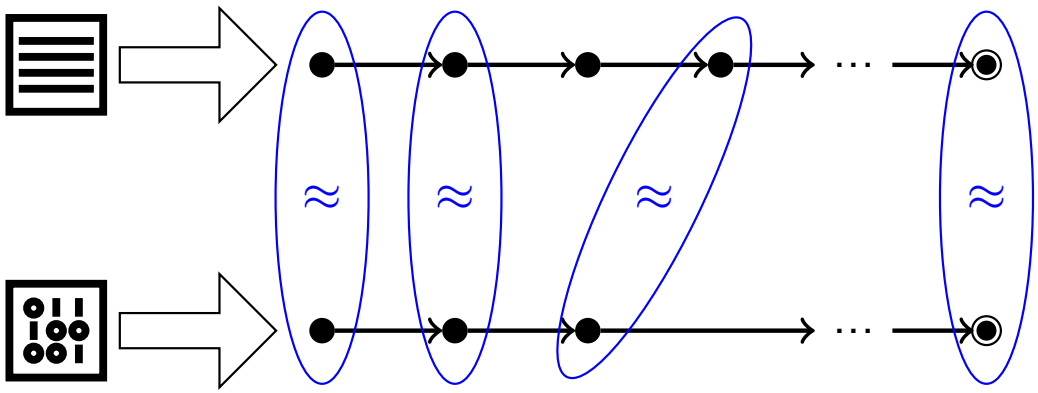
Simulation



Simulation

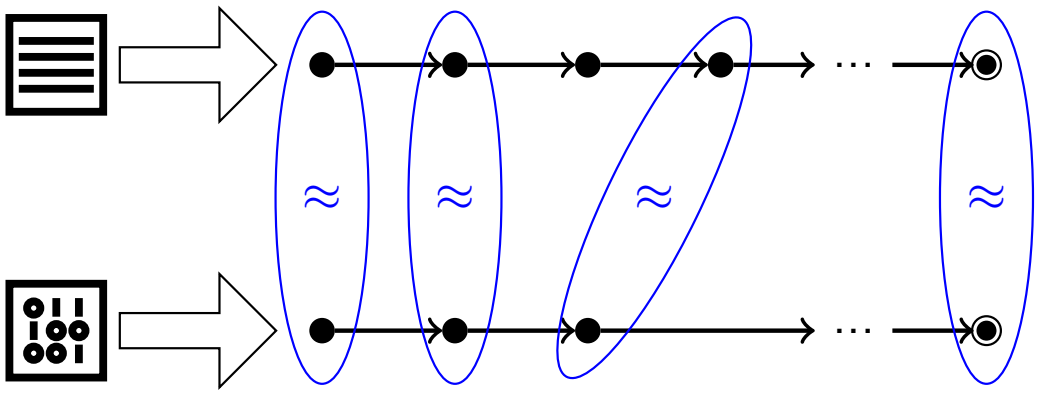


Simulation



locale backward-simulation =
  L1: semantics $\rightarrow_1$ $\circledcirc_1$ +
  L2: semantics $\rightarrow_2$ $\circledcirc_2$ +
  well-founded $\sqsubset$ **for**
    $\rightarrow_1$ **and** $\circledcirc_1$ :: *'state*$_1$ $\Rightarrow$ bool **and**
    $\rightarrow_2$ **and** $\circledcirc_2$ :: *'state*$_2$ $\Rightarrow$ bool **and**
    $\sqsubset$ :: *'index* $\Rightarrow$ *'index* $\Rightarrow$ bool +
  fixes match :: *'index* $\Rightarrow$ *'state*$_1$ $\Rightarrow$ *'state*$_2$ $\Rightarrow$ bool
  assumes
    match-final: match $i\ s_1\ s_2 \Longrightarrow \circledcirc_2\ s_2 \Longrightarrow \circledcirc_1\ s_1$ **and**
    simulation: match $i\ s_1\ s_2 \Longrightarrow s_2 \rightarrow_2 s_2' \Longrightarrow$
      $(\exists i'\ s_1'.\ s_1 \rightarrow_1^+ s_1' \wedge \text{match}\ i'\ s_1'\ s_2') \vee (\exists i'.\ \text{match}\ i'\ s_1\ s_2' \wedge i' \sqsubset i)$

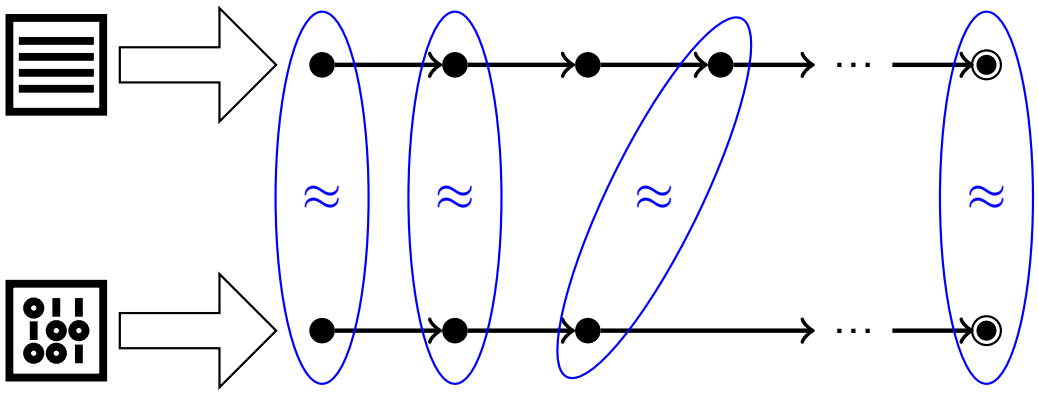
Simulation



locale backward-simulation =
  L1: semantics $\rightarrow_1$ $\circledcirc_1$ +
  L2: semantics $\rightarrow_2$ $\circledcirc_2$ +
  well-founded $\sqsubset$ **for**
    $\rightarrow_1$ **and** $\circledcirc_1$ :: *'state*$_1$ $\Rightarrow$ bool **and**
    $\rightarrow_2$ **and** $\circledcirc_2$ :: *'state*$_2$ $\Rightarrow$ bool **and**
    $\sqsubset$ :: *'index* $\Rightarrow$ *'index* $\Rightarrow$ bool +
  fixes match :: *'index* $\Rightarrow$ *'state*$_1$ $\Rightarrow$ *'state*$_2$ $\Rightarrow$ bool
  assumes
    match-final: match $i\ s_1\ s_2 \Longrightarrow \circledcirc_2\ s_2 \Longrightarrow \circledcirc_1\ s_1$ **and**
    simulation: match $i\ s_1\ s_2 \Longrightarrow s_2 \rightarrow_2 s_2' \Longrightarrow$
      $(\exists i'\ s_1'.\ s_1 \rightarrow_1^+ s_1' \wedge \text{match}\ i'\ s_1'\ s_2') \vee (\exists i'.\ \text{match}\ i'\ s_1\ s_2' \wedge i' \sqsubset i)$

multiple instances

Simulation

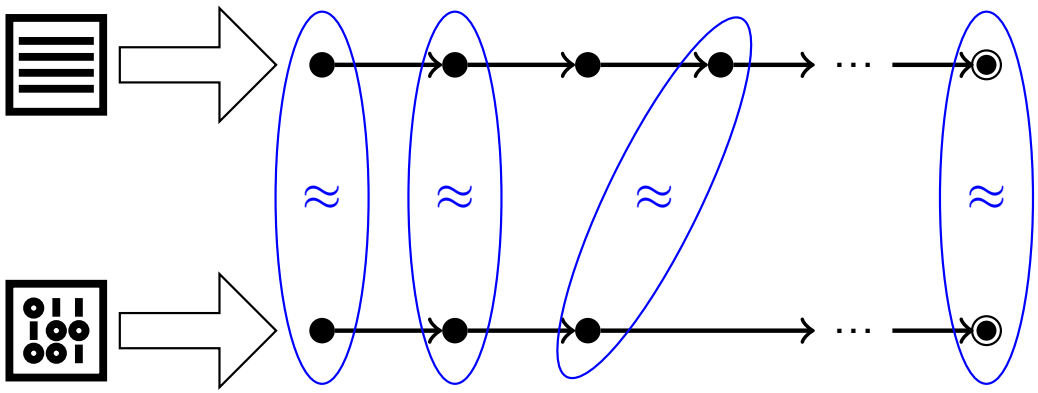


locale backward-simulation =
  L1: semantics $\rightarrow_1$ $\circledcirc_1$ +
  L2: semantics $\rightarrow_2$ $\circledcirc_2$ +
  well-founded $\sqsubset$ **for**
    $\rightarrow_1$ **and** $\circledcirc_1$ :: *'state*$_1$ $\Rightarrow$ bool **and**
    $\rightarrow_2$ **and** $\circledcirc_2$ :: *'state*$_2$ $\Rightarrow$ bool **and**
    $\sqsubset$ :: *'index* $\Rightarrow$ *'index* $\Rightarrow$ bool +
  fixes match :: *'index* $\Rightarrow$ *'state*$_1$ $\Rightarrow$ *'state*$_2$ $\Rightarrow$ bool
  assumes
    match-final: match $i\ s_1\ s_2 \Longrightarrow \circledcirc_2\ s_2 \Longrightarrow \circledcirc_1\ s_1$ **and**
    simulation: match $i\ s_1\ s_2 \Longrightarrow s_2 \rightarrow_2 s_2' \Longrightarrow$
      $(\exists i'\ s_1'.\ s_1 \rightarrow_1^+ s_1' \wedge \text{match}\ i'\ s_1'\ s_2') \vee (\exists i'.\ \text{match}\ i'\ s_1\ s_2' \wedge i' \sqsubset i)$

multiple instances

new parameters

Simulation



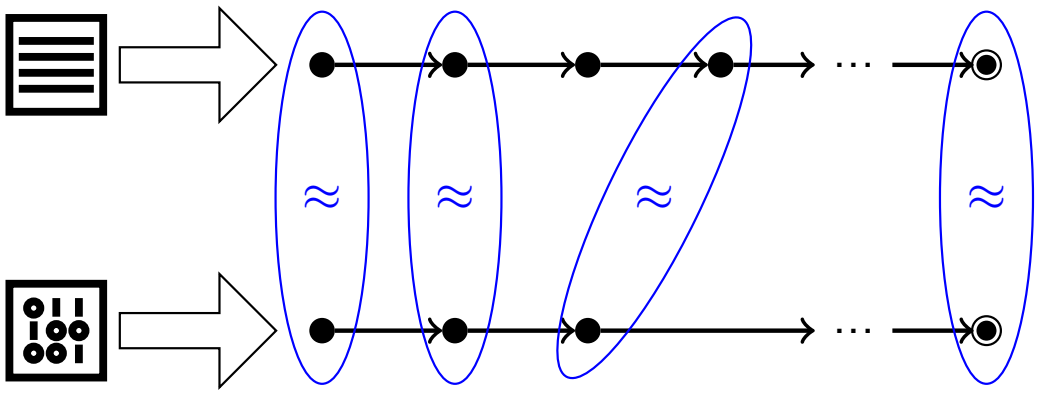
locale backward-simulation =
  L1: semantics $\rightarrow_1$ $\bullet_1$ +
  L2: semantics $\rightarrow_2$ $\bullet_2$ +
  well-founded $\sqsubset$ **for**
    $\rightarrow_1$ **and** $\bullet_1$ :: *'state*$_1$ $\Rightarrow$ bool **and**
    $\rightarrow_2$ **and** $\bullet_2$ :: *'state*$_2$ $\Rightarrow$ bool **and**
    $\sqsubset$ :: *'index* $\Rightarrow$ *'index* $\Rightarrow$ bool +
  fixes match :: *'index* $\Rightarrow$ *'state*$_1$ $\Rightarrow$ *'state*$_2$ $\Rightarrow$ bool
  assumes
    match-final: match $i\ s_1\ s_2 \Longrightarrow \bullet_2\ s_2 \Longrightarrow \bullet_1\ s_1$ **and**
    simulation: match $i\ s_1\ s_2 \Longrightarrow s_2 \rightarrow_2 s_2' \Longrightarrow$
      $(\exists i'\ s_1'.\ s_1 \rightarrow_1^+ s_1' \wedge \text{match}\ i'\ s_1'\ s_2') \vee (\exists i'.\ \text{match}\ i'\ s_1\ s_2' \wedge i' \sqsubset i)$

multiple instances

new parameters

assumptions

Simulation



locale backward-simulation =
  L1: semantics $\rightarrow_1$ $\bullet_1$ +
  L2: semantics $\rightarrow_2$ $\bullet_2$ +
  well-founded $\sqsubset$ **for**
    $\rightarrow_1$ **and** $\bullet_1$ :: *'state*$_1$ $\Rightarrow$ bool **and**
    $\rightarrow_2$ **and** $\bullet_2$ :: *'state*$_2$ $\Rightarrow$ bool **and**
    $\sqsubset$ :: *'index* $\Rightarrow$ *'index* $\Rightarrow$ bool +
  fixes match :: *'index* $\Rightarrow$ *'state*$_1$ $\Rightarrow$ *'state*$_2$ $\Rightarrow$ bool
  assumes
    match-final: match $i\ s_1\ s_2 \implies \bullet_2\ s_2 \implies \bullet_1\ s_1$ **and**
    simulation: match $i\ s_1\ s_2 \implies s_2 \rightarrow_2 s_2' \implies$
      $(\exists i'\ s_1'.\ s_1 \rightarrow_1^+ s_1' \wedge \text{match}\ i'\ s_1'\ s_2') \vee (\exists i'.\ \text{match}\ i'\ s_1\ s_2' \wedge i' \sqsubset i)$

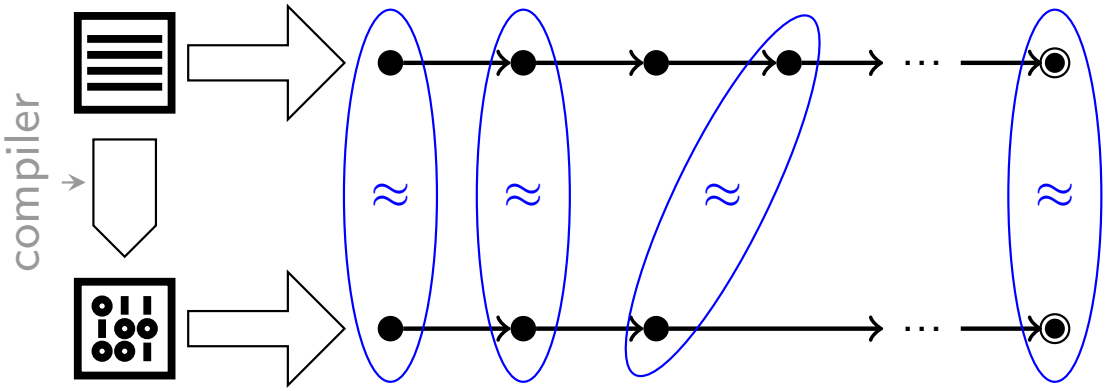
multiple instances

new parameters

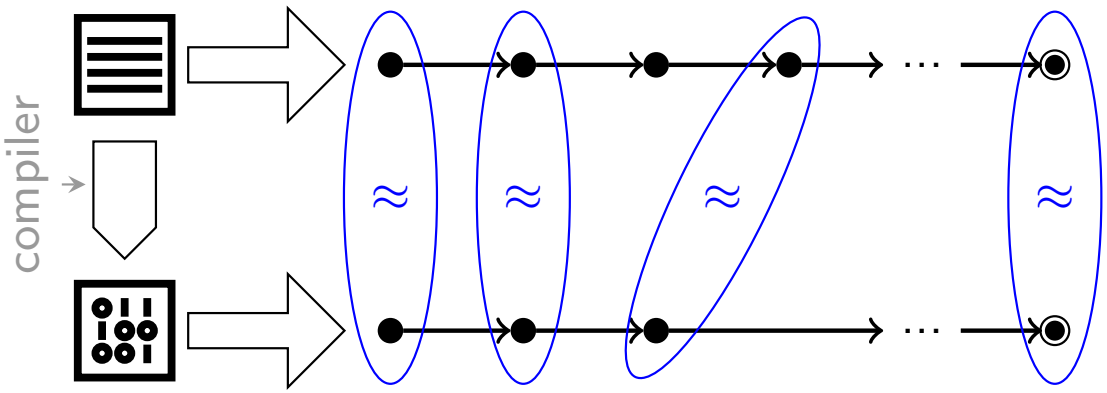
assumptions

transitive closure

Compiler



Compiler



locale compiler =
  L1: language $\rightarrow_1$ $\bullet_1$ load$_1$ +
  L2: language $\rightarrow_2$ $\bullet_2$ load$_2$ +
  backward-simulation $\rightarrow_1$ $\bullet_1$ $\rightarrow_2$ $\bullet_2$ $\sqsubset$ match **for**
    $\rightarrow_1$ **and** $\bullet_1$ **and** load$_1$ :: *'prog*$_1$ $\Rightarrow$ *'state*$_1$ $\Rightarrow$ bool **and**
    $\rightarrow_2$ **and** $\bullet_2$ **and** load$_2$ :: *'prog*$_2$ $\Rightarrow$ *'state*$_2$ $\Rightarrow$ bool **and**
    $\sqsubset$ **and** match :: *'index* $\Rightarrow$ *'state*$_1$ $\Rightarrow$ *'state*$_2$ $\Rightarrow$ bool
  fixes compile :: *'prog*$_1$ $\Rightarrow$ *'prog*$_2$ option
  assumes
    compile-load: compile p_1 = Some p_2 $\Longrightarrow$ load$_1$ p_1 = Some s_1 $\Longrightarrow$
      $\exists s_2\ i.$ load$_2$ p_2 = Some s_2 $\wedge$ match i s_1 s_2

Derived lemma (partial correctness)

context compiler **begin**

$$\frac{\text{compile}\ p_1 = \text{Some}\ p_2 \qquad \text{load}_1\ p_1 = \text{Some}\ s_1 \qquad \text{load}_2\ p_2 = \text{Some}\ s_2 \qquad s_2 \Downarrow b_2 \qquad \neg\ \text{is-wrong}\ b_2}{\exists b_1\ i.\ s_1 \Downarrow b_1 \wedge b_1 \simeq b_2}$$

end

Derived lemma (compiler composition)

$$\frac{\begin{array}{c}\mathsf{compiler} \rightarrow_1 \circledcirc_1\ \mathit{load}_1 \rightarrow_2 \circledcirc_2\ \mathit{load}_2 \sqsubset_{1\text{-}2} \mathit{match}_{1\text{-}2}\ \mathit{compile}_{1\text{-}2} \\ \mathsf{compiler} \rightarrow_2 \circledcirc_2\ \mathit{load}_2 \rightarrow_3 \circledcirc_3\ \mathit{load}_3 \sqsubset_{2\text{-}3} \mathit{match}_{2\text{-}3}\ \mathit{compile}_{2\text{-}3}\end{array}}{\mathsf{compiler} \rightarrow_1 \circledcirc_1\ \mathit{load}_1 \rightarrow_3 \circledcirc_3\ \mathit{load}_3\ [\ldots]\ [\ldots]\ (\mathit{compile}_{2\text{-}3} \Leftarrow \mathit{compile}_{1\text{-}2})}$$

Instantiation example

We used the framework to formalize optimizations of virtual machines.

Three bytecode languages

- **Std** standard (baseline)
- **Inca** inline caching
- **Ubx** unboxed data

Instantiation example: language Inca

datatype *'instr* fundef = ...
datatype (*'fenv*, *'menv*, *'fun*) prog = ...
datatype (*'fun*, *'operand*) frame = ...
datatype (*'fenv*, *'menv*, *'frame*) state = ...
datatype (*'dyn*, *'var*, *'fun*, *'op*, *'opinl*) instr = ...

Instantiation example: language Inca

datatype *'instr* fundef = ...
datatype (*'fenv*, *'menv*, *'fun*) prog = ...
datatype (*'fun*, *'operand*) frame = ...
datatype (*'fenv*, *'menv*, *'frame*) state = ...
datatype (*'dyn*, *'var*, *'fun*, *'op*, *'opinl*) instr = ...

locale inca =
  Fenv: env +
  Menv: env +
  dynval +
  nary-operation-inl

Instantiation example: language Inca

datatype *'instr* fundef = ...
datatype (*'fenv*, *'menv*, *'fun*) prog = ...
datatype (*'fun*, *'operand*) frame = ...
datatype (*'fenv*, *'menv*, *'frame*) state = ...
datatype (*'dyn*, *'var*, *'fun*, *'op*, *'opinl*) instr = ...

locale inca =
Fenv: env F-empty F-get F-add F-to-list +
Menv: env M-empty M-get M-add M-to-list +
dynval is-true is-false +
nary-operation-inl Op Arity InlOp Inl DeInl **for**

F-empty **and** F-add **and** F-to-list **and**
F-get :: *'fenv* ⇒ *'fun* ⇒
(*'dyn*, *'var*, *'fun*, *'op*, *'opinl*) instr fundef option **and**

M-empty **and** M-add **and** M-to-list **and**
M-get :: *'menv* ⇒ *'var* ⇒ *'dyn* option **and**

is-true **and** is-false :: *'dyn* ⇒ bool **and**

Op :: *'op* ⇒ *'dyn* list ⇒ *'dyn* **and** Arith **and**
InlOp **and** Inl **and** DeInl :: *'opinl* ⇒ *'op*

Instantiation example: language Inca (cont.)

context inca **begin**

inductive step :: ('*fenv*, '*menv*, ('*fun*, '*dyn*) frame) state ⇒
('*fenv*, '*menv*, ('*fun*, '*dyn*) frame) state ⇒ bool **where** ...

inductive final :: ('*fenv*, '*menv*, ('*fun*, '*dyn*) frame) state ⇒ bool **where** ...

definition load :: ('*fenv*, '*menv*, '*fun*) prog ⇒
('*fenv*, '*menv*, ('*fun*, '*dyn*) frame) state option **where** ...

sublocale inca-semantics: semantics step final ...
sublocale inca-language: language step final ...

end

Experience report on locales

Framework

- Small code base (1 kLOC)
- Tiny interface (the actual locales)

Instantiations

- Optimization of virtual machines
- Three bytecode languages
- Two compilers
- Somewhat bigger (7 kLOC)

Experience report on locales

Framework

- Small code base (1 kLOC)
- Tiny interface (the actual locales)

Instantiations

- Optimization of virtual machines
- Three bytecode languages
- Two compilers
- Somewhat bigger (7 kLOC)

Pros

+ Clear, explicit abstractions
+ Separation of concerns (parameters, assumptions, derived results)
+ Multiple abstract data types (e.g. *'state*, *'prog*)

Experience report on locales

Framework

- Small code base (1 kLOC)
- Tiny interface (the actual locales)

Instantiations

- Optimization of virtual machines
- Three bytecode languages
- Two compilers
- Somewhat bigger (7 kLOC)

Pros

+ Clear, explicit abstractions
+ Separation of concerns (parameters, assumptions, derived results)
+ Multiple abstract data types (e.g. *'state*, *'prog*)

Cons

– Syntactical overhead (predicates, imports, type annotations)
– No parametric types or type aliases in locale definitions
– Different syntax when referencing parameters vs derived definitions

Syntactical overhead

Extending locales requires to explicitly provide parameters to *avoid name clashes* and *express sharing*.

Syntactical overhead

Extending locales requires to explicitly provide parameters to *avoid name clashes* and *express sharing*.

Parameters need type annotations to *name the type variables*.

Syntactical overhead

Extending locales requires to explicitly provide parameters to *avoid name clashes* and *express sharing*.

Parameters need type annotations to *name the type variables*.

Parametric data types + absence of type aliases = bloated type annotations.

Syntactical overhead

Extending locales requires to explicitly provide parameters to *avoid name clashes* and *express sharing*.

Parameters need type annotations to *name the type variables*.

Parametric data types + absence of type aliases = bloated type annotations.

Overhead grows with the number of locales, parameters, and type variables.
(e.g. Inca-Ubx simulation: 13 type variables, 27 parameters, 11 are shared)

Discussion

What I did (so far)

- Provide complete list of parameters
- Write minimal set of type annotations
- Copy and paste from one locale to the other

Discussion

What would be nice (alternatives)

1. Use locale namespaces to avoid name clashes (as for assumptions)?
2. Use named parameter instantiation (as for lemmas)?
3. Use named type variable instantiation?
4. Use positional type variable instantiation?
5. ???

Discussion

What would be nice (alternatives)

1. Use locale namespaces to avoid name clashes (as for assumptions)?
2. Use named parameter instantiation (as for lemmas)?
3. Use named type variable instantiation?
4. Use positional type variable instantiation?
5. ???

locale inca =
  Fenv: env[**where** get = F-get] + (1. and 2.)
  Menv: env[**where** get = M-get] + (1. and 2.)
  dynval[**where types** *'dyn* = *'dyn*] + (1. and 3.)
  nary-operation-inl[**of types** *'op 'opinl 'dyn*] **for** (1. and 4.)
    F-get :: *'fenv* ⇒ *'fun* ⇒ (*'dyn*, *'var*, *'fun*, *'op*, *'opinl*) instr fundef option **and**
    M-get :: *'menv* ⇒ *'fun* ⇒ *'dyn* option

Discussion

What would be nice (alternatives)

1. Use locale namespaces to avoid name clashes (as for assumptions)?
2. Use named parameter instantiation (as for lemmas)?
3. Use named type variable instantiation?
4. Use positional type variable instantiation?
5. ???

locale inca =
  Fenv: env[**where** get = F-get] + (1. and 2.)
  Menv: env[**where** get = M-get] + (1. and 2.)
  dynval[**where types** *'dyn* = *'dyn*] + (1. and 3.)
  nary-operation-inl[**of types** *'op 'opinl 'dyn*] **for** (1. and 4.)
    F-get :: *'fenv* ⇒ *'fun* ⇒ (*'dyn*, *'var*, *'fun*, *'op*, *'opinl*) instr fundef option **and**
    M-get :: *'menv* ⇒ *'fun* ⇒ *'dyn* option

Estimate savings (number of parameters)

inca: 66 % (1. and 2.), 100 % (plus 3. or 4.)
Inca-Ubs simulation: 52 % (1. and 2.), 60 % (plus 3. or 4.) (sharing)

Discussion

What would be nice (alternatives)

1. Use locale namespaces to avoid name clashes (as for assumptions)?
2. Use named parameter instantiation (as for lemmas)?
3. Use named type variable instantiation?
4. Use positional type variable instantiation?
5. ???

locale inca =
  Fenv: env[**where** get = F-get] + (1. and 2.)
  Menv: env[**where** get = M-get] + (1. and 2.)
  dynval[**where types** *'dyn* = *'dyn*] + (1. and 3.)
  nary-operation-inl[**of types** *'op 'opinl 'dyn*] **for** (1. and 4.)
    F-get :: *'fenv* ⇒ *'fun* ⇒ (*'dyn*, *'var*, *'fun*, *'op*, *'opinl*) instr fundef option **and**
    M-get :: *'menv* ⇒ *'fun* ⇒ *'dyn* option

Estimate savings (number of parameters)

inca: 66 % (1. and 2.), 100 % (plus 3. or 4.)
Inca-Ubs simulation: 52 % (1. and 2.), 60 % (plus 3. or 4.) (sharing)

Thank you

Derived results (behaviour)

context semantics **begin**

$$\frac{\mathsf{step}^*\ s_1\ s_2 \qquad \mathsf{finished\ step}\ s_2 \qquad \mathsf{final}\ s_2}{s_1 \Downarrow \mathsf{Terminates}\ s_2}$$

$$\frac{\mathsf{step}^*\ s_1\ s_2 \qquad \mathsf{finished\ step}\ s_2 \qquad \neg\,\mathsf{final}\ s_2}{s_1 \Downarrow \mathsf{Goes\text{-}wrong}\ s_2}$$

$$\frac{\mathsf{step}^\infty\ s_1}{s_1 \Downarrow \mathsf{Diverges}}$$

end