Verification of an ML compiler

Lecture 1:
An introduction to compiler verification

Marktoberdorf Summer School MOD 2017

Magnus O. Myreen, Chalmers University of Technology

Introduction

Your program crashes.

Where do you look for the fault?

- Do you look at *your source code*?
- Do *look at the code for the compiler* you used?

users want to rely on compilers

Verified compilers

What?

A verified compiler is a compiler that comes with a machine-checker proof. The proof states that the compiler ***preserves the behaviour of source programs***.

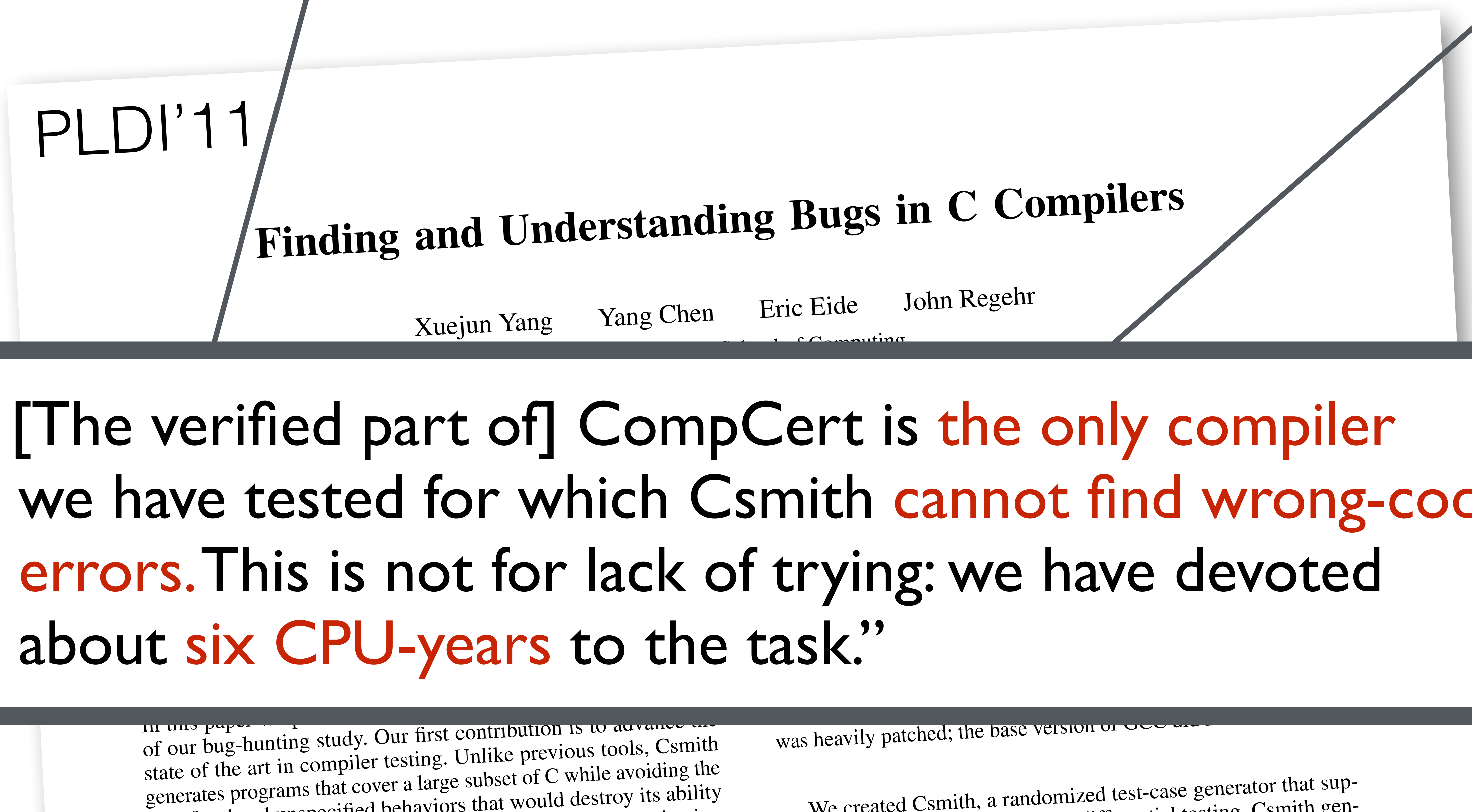
Traditional compiler development relies on testing.

Compiler verification is considered too costly.

cost reduction?

All (unverified) compilers have bugs

“ Every compiler we tested was found to crash and also to silently generate wrong code when presented with valid input. ”


PLDI’11

Finding and Understanding Bugs in C Compilers

Xuejun Yang Yang Chen Eric Eide John Regehr

of our bug-hunting study. Our first contribution is to advance the state of the art in compiler testing. Unlike previous tools, Csmith generates programs that cover a large subset of C while avoiding the

was heavily patched; the base version of GCC

We created Csmith, a randomized test-case generator that sup-

“ [The verified part of] CompCert is the only compiler we have tested for which Csmith cannot find wrong-code errors. This is not for lack of trying: we have devoted about six CPU-years to the task.”

Motivations

Bugs in compilers are not tolerated by users

Bugs can be hard to find by testing

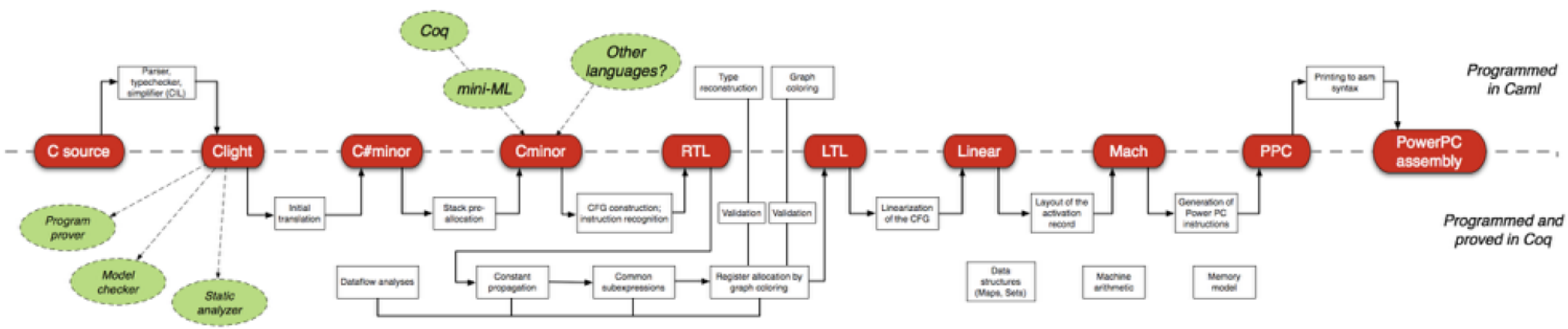
Verified compilers must be used for verification of source-level programs to imply guarantees at the level of verified machine code

Research question: how easy (cheap) can we make compiler verification?

State of the art

CompCert

CompCert C compiler



Leroy et al. ***Source:*** http://compcert.inria.fr/

Compiles C source code to assembly.

Has **good performance numbers**

Proved correct in Coq.

http://compcert.inria.fr/

CakeML compiler

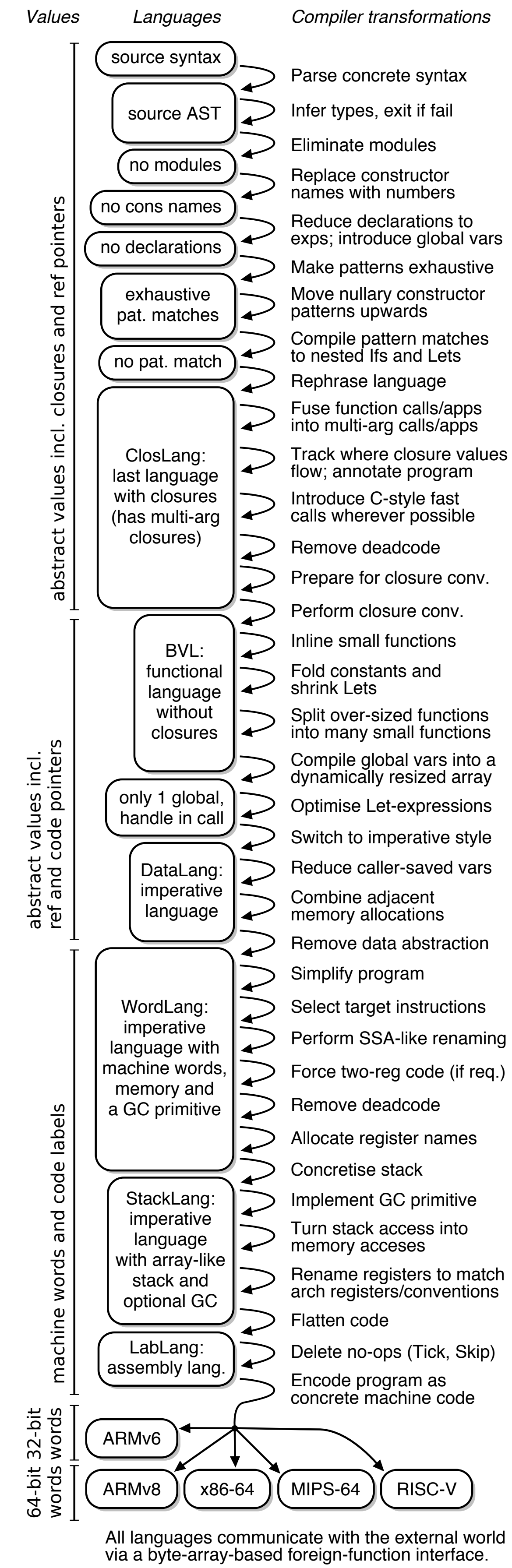
Compiles CakeML concrete syntax to machine code.

Proved correct in HOL4.

Has **mostly good performance numbers** (later lecture)

Known as the first verified compiler to be bootstrapped.

I'm one of the six developers behind version 2 (diagram to the right).



later lecture zooms in

https://cakeml.org/

A spectrum

robust, inflexible

proved to always work correctly

more flexible, but can be fragile

produces a proof for each run

Verified compilers … *Proof-producing compilers*

Pilsner

Fiat

Cogent

CompCert C compiler

Translation validation for a verified OS kernel

CakeML compiler

CompCertTSO

These 4 lectures on

Verification of an ML compiler

will focus on key concepts rather than implementation details

The lectures:

Lecture 1: introduction to compiler verification

Lecture 2: data representation and garbage collection

Lecture 3: closures and call optimisations

Lecture 4: compiler bootstrapping

but feel free to ask

The lectures will not cover:

- Verification of parsing and ML-style type inference
- Optimisations (except call optimisations in lecture 3)
- Modelling and verification of I/O
- How to manage a large code base of proofs

The lectures:

- Lecture 1: introduction to compiler verification
- Lecture 2: data representation and garbage collection
- Lecture 3: closures and call optimisations
- Lecture 4: compiler bootstrapping

Let's get started!

Lecture 1: introduction to compiler verification

Questions

Should we write a new compiler for the verification?

yes, because then it has nice invariants

What should we prove about it?

that the generated programs behave "the same" as the source programs

Do we need to use mechanised proof?

yes

Proving a compiler correct

Ingredients:

- a formal logic for the proofs
- accurate models of
  - the source language
  - the target language
  - the compiler algorithm

Tools:

- a proof assistant (software)

Method:

- prove simulation theorem using induction

does this correspond with reality?

requires ***testing***

Ingredient 1: Formal logic

Formal logic

These lectures will use classical higher-order logic (HOL)

HOL = lambda calculus with simple ML-like types

Allows for quantification over functions and relations.

Example 1: Skolem

$$\vdash (\forall x.\ \exists y.\ P\ x\ y) \iff \exists f.\ \forall x.\ P\ x\ (f\ x)$$

Example 2: complete induction over nat:

$$\vdash (\forall n.\ (\forall m.\ m < n \Rightarrow P\ m) \Rightarrow P\ n) \Rightarrow \forall n.\ P\ n$$

Example 3: definition of list permutation:

$$\vdash \texttt{PERM}\ L_1\ L_2 \iff \forall x.\ \texttt{FILTER}\ (\texttt{(=)}\ x)\ L_1 = \texttt{FILTER}\ (\texttt{(=)}\ x)\ L_2$$

User-definitions in HOL

Datatypes:

```
datatype e = Var string
           | Num int
           | Add e e
           | Assign string e
```

```
datatype r = Rval int
           | Rbreak
           | Rfail
```

Recursive functions (that terminate or are tail-recursive):

$\texttt{FILTER}\ P\ \texttt{[]}\ \texttt{=}\ \texttt{[]}$

$\texttt{FILTER}\ P\ \texttt{(}h{::}t\texttt{)}\ \texttt{=}\ \textbf{if}\ P\ h\ \textbf{then}\ h{::}\texttt{FILTER}\ P\ t\ \textbf{else}\ \texttt{FILTER}\ P\ t$

Inductive relations (also co-inductive relations):

$$\text{(S1)}\ \frac{(t_1, s) \Downarrow_{\texttt{t}} (\texttt{Rval}\ n_1, s_1) \qquad (t_2, s_1) \Downarrow_{\texttt{t}} r}{(\texttt{Seq}\ t_1\ t_2, s) \Downarrow_{\texttt{t}} r} \qquad \text{(S2)}\ \frac{(t_1, s) \Downarrow_{\texttt{t}} (r, s_1) \qquad \neg\texttt{is\_Rval}\ r}{(\texttt{Seq}\ t_1\ t_2, s) \Downarrow_{\texttt{t}} (r, s_1)}$$

Ingredient 2: Language models

Syntax

The abstract syntax is defined as a datatype.

A toy *source* language:

```
t = Dec string t
  | Exp e
  | Break
  | Seq t t
  | If e t t
  | For e e t

e = Var string
  | Num int
  | Add e e
  | Assign string e
```

evaluates an expression

Break aborts a loop

For-loop

expressions can have side-effects (c.f. ML)

Syntax

The abstract syntax is defined as a datatype.

A toy ***source*** language:

```
t = Dec string t
  | Exp e
  | Break
  | Seq t t
  | If e t t
  | For e e t

e = Var string
  | Num int
  | Add e e
  | Assign string e
```

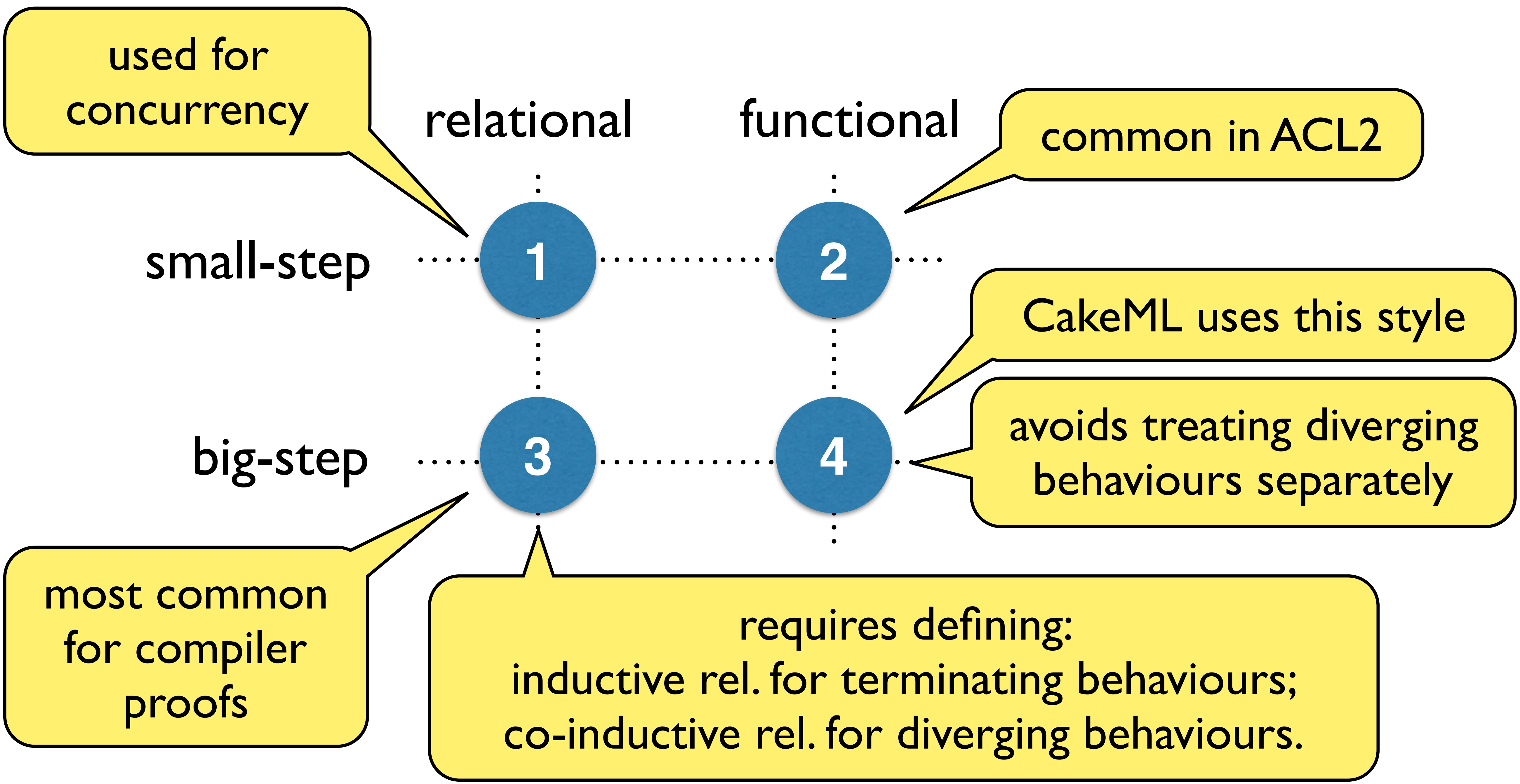
A toy ***target*** language:

```
inst
  = Add reg reg reg
  | Int reg int
  | Jmp offset
  | JmpIf reg offset
```

an assembly program is a list of instructions

Operational Semantics — the options

There are 4 main variants one can choose from.



Towards a functional big-step semantics (1)

We define an interpreter for the source *in Standard ML*:

```
fun lookup y [] = NONE
  | lookup y ((x,v)::xs) = if y = x then SOME v else lookup y xs

fun run_e s (Var x) =
      (case lookup x s of
         NONE => (Rfail,s)
       | SOME v => (Rval v,s))
  | run_e s (Num i) = (Rval i,s)
  | run_e s (Assign (x, e)) =
      (case run_e s e of
         (Rval n1, s1) => (Rval n1, (x,n1)::s1)
       | r => r)
  | run_e s (Add (e1, e2)) = ...
```

state is a simple assoc-list

assignments update the state

expression evaluation returns `Rval` or `Rfail` and new state

Towards a functional big-step semantics (2)

… and the evaluation of statements (type t):

```
fun run_t s (Exp e) = run_e s e
  | run_t s (Dec (x, t)) = run_t ((x,0)::s) t
  | run_t s Break = (Rbreak, s)
  | run_t s (Seq (t1, t2)) =
     (case run_t s t1 of
        (Rval _, s1) => run_t s1 t2
      | r => r)
  | run_t s (If (e, t1, t2)) =
     (case run_e s e of
        (Rval n1, s1) => run_t s1 (if n1 = 0 then t2 else t1)
      | r => r)
```

Break causes an Rbreak

Rbreak skips Seq-code

Towards a functional big-step semantics (3)

… so far everything we wrote in ML could be defined in HOL.

… but now this recursive call introduces potential non-termination

```
| run_t s (For (e1, e2, t)) =
   (case run_e s e1 of
      (Rval n1, s1) =>
        if n1 = 0 then (Rval 0, s1) else
         (case run_t s1 t of
            (Rval _, s2) =>
              (case run_e s2 e2 of
                 (Rval _, s3) => run_t s3 (For (e1, e2, t))
                | r => r)
          | (Rbreak, s2) => (Rval 0, s2)
          | r => r)
    | r => r)
```

Why is non-terimination an issue?

Suppose HOL allowed definitions of non-terminating functions, e.g.

$$f\ n = f\ n + 1$$

then $\quad f\ n - f\ n = f\ n + 1 - f\ n$

and $\quad 0 = 1$

Towards a functional big-step semantics (4)

We can force our functions to terminate by inserting a clock.

ML:

```
| run_t s (For (e1, e2, t)) =
    ...
      (Rval _, s3) => run_t s3 (For (e1, e2, t))
```

HOL:

$\texttt{eval\_t}\ s\ (\texttt{For}\ e_1\ e_2\ t)\ =$

$\quad \ldots$

$\quad\quad (\texttt{Rval}\ \_, s_3) \Rightarrow$

$\quad\quad\quad \textbf{if}\ s_3.\texttt{clock} \neq 0\ \textbf{then}$

$\quad\quad\quad\quad \texttt{eval\_t}\ (\texttt{dec\_clock}\ s_3)\ (\texttt{For}\ e_1\ e_2\ t)$

$\quad\quad\quad \textbf{else}\ (\texttt{Rtimeout}, s_3)$

in HOL, the state is a record containing a clock

which gets decremented

if it hits zero, then we abort with an *uncatchable* exception

Functional big-step semantics

Other parts do not require clock clutter:

HOL:

eval_t s (Exp e) = eval_e s e
eval_t s (Dec x t) = eval_t (st
eval_t s Break = (Rbreak,s)
eval_t s (Seq t_1 t_2) =
  case eval_t s t_1 **of**
    (Rval _,s_1) $\Rightarrow$ eval_t s_1 t_2
  | $r \Rightarrow r$
eval_t s (If e t_1 t_2) =
  case eval_e s e **of**
    (Rval n_1,s_1) $\Rightarrow$ eval_t s_1 (**if** n_1 = 0 **then** t_2 **else** t_1)
  | $r \Rightarrow r$

… since the size of
the input shrinks

Observational semantics

Compiler proof is to relate different *observational semantics*.

Without I/O, programs can only `Terminate`, `Diverge` or `Crash`.

terminates if there is a clock that is sufficient

semantics t =
if $\exists c\ v\ s.$ sem_t (s_with_clock c) t = (Rval v,s) **then** Terminate
else if $\forall c.\ \exists s.$ sem_t (s_with_clock c) t = (Rtimeout,s) **then** Diverge
else Crash

diverges if all clocks cause timeouts

crashes otherwise, e.g. when `Rbreak` propagates to the top

Exercise

Define a functional big-step semantics in your favourite prover (e.g. HOL4, Isabelle/HOL, Coq, ACL2)

There is a trick to the termination proof. *Ask me!*

Ingredient 3: Compiler function

Ingredient 3: Compiler function

Three phases:

phase 1: rewrite For to something simpler

phase 2: split expressions into instructions

phase 3: flatten to list of assembly instructions

Ingredient 3: Compiler function

Three phases:

phase 1: rewrite For to something simpler

```
phase1 (Exp e) = Exp e
phase1 (Dec x t) = Seq (Exp (Assign x (Num 0))) (phase1 t)
phase1 Break = Break
phase1 (Seq t1 t2) = Seq (phase1 t1) (phase1 t2)
phase1 (If e t1 t2) = If e (phase1 t1) (phase1 t2)
phase1 (For e1 e2 t) = Loop (If e1 (Seq (phase1 t) (Exp e2)) Break)
```

where `Loop t = For (Num 1) (Num 1) t`

Makes all loops simple while-true loops.

Proving a compiler correct

Ingredients:

- a formal logic for the proofs
- accurate models of
  - the source language
  - the target language
  - the compiler algorithm

Tools:

- a proof assistant (software)

Method:

- prove simulation theorem using induction

Induction theorem

$$
\begin{array}{l}
\vdash \forall P.\\
\quad (\forall s\ e.\ P\ s\ (\texttt{Exp}\ e)) \land\\
\quad (\forall s\ x\ t.\ P\ (\texttt{store\_var}\ x\ 0\ s)\ t \Rightarrow P\ s\ (\texttt{Dec}\ x\ t)) \land\\
\quad (\forall s.\ P\ s\ \texttt{Break}) \land\\
\quad (\forall s\ t_1\ t_2.\\
\quad\quad (\forall v_2\ s_1\ v_5.\\
\quad\quad\quad (\texttt{eval\_t}\ s\ t_1 = (v_2,s_1)) \land (v_2 = \texttt{Rva}\\
\quad\quad P\ s\ t_1 \Rightarrow\\
\quad\quad P\ s\ (\texttt{Seq}\ t_1\ t_2)) \land\\
\quad (\forall s\ e\ t_1\ t_2.\\
\quad\quad (\forall v_2\ s_1\ n_1.\\
\quad\quad\quad (\texttt{eval\_e}\ s\ e = (v_2,s_1)) \land (v_2 = \texttt{Rva}\\
\quad\quad\quad P\ s_1\ (\textbf{if}\ n_1 = 0\ \textbf{then}\ t_2\ \textbf{else}\ t_1)) \Rightarrow\\
\quad\quad P\ s\ (\texttt{If}\ e\ t_1\ t_2)) \land\\
\quad (\forall s\ e_1\ e_2\ t.\\
\quad\quad (\forall v_4\ s_1\ n_1\ v_3\ s_2\ v_7\ v_2\ s_3\ v_5.\\
\quad\quad\quad (\texttt{eval\_e}\ s\ e_1 = (v_4,s_1)) \land (v_4 = \texttt{Rval}\ n_1) \land n_1 \neq 0 \land\\
\quad\quad\quad (\texttt{eval\_t}\ s_1\ t = (v_3,s_2)) \land (v_3 = \texttt{Rval}\ v_7) \land\\
\quad\quad\quad (\texttt{eval\_e}\ s_2\ e_2 = (v_2,s_3)) \land (v_2 = \texttt{Rval}\ v_5) \land s_3.\texttt{clock} \neq 0 \Rightarrow\\
\quad\quad\quad P\ (\texttt{dec\_clock}\ s_3)\ (\texttt{For}\ e_1\ e_2\ t)) \land\\
\quad\quad (\forall v_4\ s_1\ n_1.\\
\quad\quad\quad (\texttt{eval\_e}\ s\ e_1 = (v_4,s_1)) \land (v_4 = \texttt{Rval}\ n_1) \land n_1 \neq 0 \Rightarrow\\
\quad\quad\quad P\ s_1\ t) \Rightarrow\\
\quad\quad P\ s\ (\texttt{For}\ e_1\ e_2\ t)) \Rightarrow\\
\quad \forall v\ v_1.\ P\ v\ v_1
\end{array}
$$

Has structure of functional big-step interpreter.

One case for each program construct.

Simulation proof

Particularly simple for phase 1:

$$\vdash \forall s\ t.\ \texttt{eval\_t}\ s\ (\texttt{phase1}\ t) = \texttt{eval\_t}\ s\ t$$

Requires only 8 lines of HOL4 proof script (mostly rewriting).

We can lift this to the observational semantics
with a one-line proof by rewriting:

$$\vdash \forall t.\ \texttt{semantics}\ (\texttt{phase1}\ t) = \texttt{semantics}\ t$$

Simulation theorems *in general*

Usually, each compile phase requires a theorem of the form:

non-failing evaluation of source intermediate language (IL)

```
evaluate code s1 = (res,s2) ∧ res ≠ Rfail ∧
state_rel s1 t1 ⇒
∃t2. evaluate (compile code) t1 = (res,t2) ∧
     state_rel s2 t2
```

implies

similar evaluation in target IL

… w.r.t. some relation between states (**state_rel**)

Simulation theorems *in general*

Usually, each compile phase requires a theorem of the form:

```
evaluate code s1 = (res,s2) ∧ res ≠ Rfail ∧
state_rel s1 t1 ⇒

∃t2. evaluate (compile code) t1 = (res,t2) ∧
     state_rel s2 t2
```

usually: `state_rel` keeps clocks in sync

variant: target can consume more clock ticks

Simulation theorems *in general*

Usually, each compile phase requires a theorem of the form:

```
evaluate code s1 = (res,s2) ∧ res ≠ Rfail ∧
state_rel s1 t1 ⇒

∃t2. evaluate (compile code) t1 = (res,t2) ∧
     state_rel s2 t2
```

Sufficient to prove observational equivalence for both terminating and diverging runs.

Phase 2 simulation theorem

source IL

not failing

restrict language syntax
following phase 1

$\vdash \forall s\ e\ t\ n\ res\ s_1$.

(eval_t s e = (res,s_1)) $\wedge$ $res \neq$ Rfail $\wedge$ phase2_subset e $\wedge$

possible … s … tore $\sqsubseteq$ t.store $\wedge$

(t.cloc … $\wedge$ … trlen n $\Rightarrow$

target IL

compiler

state relation

$\exists t_1$.

(eval_t t (flatten_t e n) = (res,t_1)) $\wedge$

s_1.store $\sqsubseteq$ t_1.store $\wedge$ (t_1.clock = s_1.clock) $\wedge$

possible_var_name n s_1.store $\wedge$

$\forall k\ v$.

possible_var_name k s.store $\wedge$ t_max e < strlen k $\wedge$

strlen k < strlen n $\wedge$ (lookup t.store k = SOME v) $\Rightarrow$

(lookup t_1.store k = SOME v)

extra property

Composing top-level theorems

Each phase maintains observational equivalence:

$$\texttt{semantics (phase1 } t\texttt{) = semantics } t$$

$$\texttt{semantics } t \neq \texttt{Crash} \wedge \texttt{phase2\_subset } t \Rightarrow$$
$$\texttt{(semantics (phase2 } t\texttt{) = semantics } t\texttt{)}$$

$$\texttt{semantics } t \neq \texttt{Crash} \wedge \texttt{phase3\_subset } t \Rightarrow$$
$$\texttt{(asm\_semantics (phase3 } 0\ 0\ t\texttt{) = semantics } t\texttt{)}$$

observational semantics of target assembly

Here: $\texttt{compile } t \texttt{ = phase3 } 0\ 0 \texttt{ (phase2 (phase1 } t\texttt{))}$

Composing top-level theorems

lemma: correct syntax implies no Crashes

for ML: type-correct program implies no Crash

Result:

$$\vdash \forall t.\ \texttt{syntax\_ok}\ t \Rightarrow (\texttt{asm\_semantics}\ (\texttt{compile}\ t)\ \texttt{=}\ \texttt{semantics}\ t)$$

where $\texttt{compile}\ t\ \texttt{=}\ \texttt{phase3}\ 0\ 0\ (\texttt{phase2}\ (\texttt{phase1}\ t))$

What we learnt

Ingredients: formal logic, compiler, language semantics

Tools: proof assistant

Method: *using functional big-step semantics* it suffices to prove theorems of the form:

```
evaluate code s1 = (res,s2) ∧ res ≠ Rfail ∧
state_rel s1 t1 ⇒
∃t2. evaluate (compile code) t1 = (res,t2) ∧
     state_rel s2 t2
```

in order to prove observational equivalence, i.e.

$$\vdash \forall t.\ \texttt{syntax\_ok}\ t \Rightarrow (\texttt{asm\_semantics}\ (\texttt{compile}\ t) = \texttt{semantics}\ t)$$

Extra slides

Comparing functional with relation big-step

In the functional version, Seq was specified by:

```
eval_t s (Seq t1 t2) =
case eval_t s t1 of
   (Rval _,s1) ⇒ eval_t s1 t2
| r ⇒ r
```

In the relational version, Seq is specified using ***four rules***:

$$(\mathrm{S1})\ \frac{(t_1\texttt{,}\,s)\ \Downarrow_{\texttt{t}}\ (\texttt{Rval}\ n_1\texttt{,}\,s_1) \qquad (t_2\texttt{,}\,s_1)\ \Downarrow_{\texttt{t}}\ r}{(\texttt{Seq}\ t_1\ t_2\texttt{,}\,s)\ \Downarrow_{\texttt{t}}\ r}$$

$$(\mathrm{S2})\ \frac{(t_1\texttt{,}\,s)\ \Downarrow_{\texttt{t}}\ (r\texttt{,}\,s_1) \qquad \neg\texttt{is\_Rval}\ r}{(\texttt{Seq}\ t_1\ t_2\texttt{,}\,s)\ \Downarrow_{\texttt{t}}\ (r\texttt{,}\,s_1)}$$

$$(\mathrm{S1}')\ \frac{(t_1\texttt{,}\,s)\ \Uparrow_{\texttt{t}}}{(\texttt{Seq}\ t_1\ t_2\texttt{,}\,s)\ \Uparrow_{\texttt{t}}}$$

$$(\mathrm{S2}')\ \frac{(t_1\texttt{,}\,s)\ \Downarrow_{\texttt{t}}\ (\texttt{Rval}\ n_1\texttt{,}\,s_1) \qquad (t_2\texttt{,}\,s_1)\ \Uparrow_{\texttt{t}}}{(\texttt{Seq}\ t_1\ t_2\texttt{,}\,s)\ \Uparrow_{\texttt{t}}}$$

Induction from relational big-step

$$
\begin{array}{l}
\vdash \;\ldots \;\wedge\; \ldots \;\wedge\; \ldots \;\wedge\; \ldots \;\wedge\; \ldots \;\wedge\; \ldots \;\wedge\; \ldots \;\wedge\; \ldots \;\wedge \\
\quad (\forall s\ s_1\ e_1\ e_2\ t. \\
\qquad (e_1, s) \Downarrow_{\mathsf{e}} (\texttt{Rval}\ 0, s_1) \Rightarrow P\ (\texttt{For}\ e_1\ e_2\ t, s)\ (\texttt{Rval}\ 0, s_1)) \wedge \\
\quad (\forall s\ s_1\ e_1\ e_2\ t\ r. \\
\qquad (e_1, s) \Downarrow_{\mathsf{e}} (r, s_1) \wedge \neg \texttt{is\_Rval}\ r \Rightarrow P\ (\texttt{For}\ e_1\ e_2\ t, s)\ (r, s_1)) \wedge \\
\quad (\forall s\ s_1\ s_2\ s_3\ e_1\ e_2\ t\ n_1\ n_2\ n_3\ r. \\
\qquad (e_1, s) \Downarrow_{\mathsf{e}} (\texttt{Rval}\ n_1, s_1) \wedge n_1 \neq 0 \wedge P\ (t, s_1)\ (\texttt{Rval}\ n_2, s_2) \wedge \\
\qquad (e_2, s_2) \Downarrow_{\mathsf{e}} (\texttt{Rval}\ n_3, s_3) \wedge P\ (\texttt{For}\ e_1\ e_2\ t, s_3)\ r \Rightarrow \\
\qquad P\ (\texttt{For}\ e_1\ e_2\ t, s)\ r) \wedge \\
\quad (\forall s\ s_1\ s_2\ e_1\ e_2\ t\ n_1. \\
\qquad (e_1, s) \Downarrow_{\mathsf{e}} (\texttt{Rval}\ n_1, s_1) \wedge n_1 \neq 0 \wedge P\ (t, s_1)\ (\texttt{Rbreak}, s_2) \Rightarrow \\
\qquad P\ (\texttt{For}\ e_1\ e_2\ t, s)\ (\texttt{Rval}\ 0, s_2)) \wedge \\
\quad (\forall s\ s_1\ s_2\ s_3\ e_1\ e_2\ t\ n_1\ n_2\ r. \\
\qquad (e_1, s) \Downarrow_{\mathsf{e}} (\texttt{Rval}\ n_1, s_1) \wedge n_1 \neq 0 \wedge P\ (t, s_1)\ (\texttt{Rval}\ n_2, s_2) \wedge \\
\qquad (e_2, s_2) \Downarrow_{\mathsf{e}} (r, s_3) \wedge \neg \texttt{is\_Rval}\ r \Rightarrow \\
\qquad P\ (\texttt{For}\ e_1\ e_2\ t, s)\ (r, s_3 \\
\quad (\forall s\ s_1\ s_2\ e_1\ e_2\ t\ n_1\ r. \\
\qquad (e_1, s) \Downarrow_{\mathsf{e}} (\texttt{Rval}\ n_1, s_1) \wedge n_1 \neq 0 \wedge P\ (t, s_1)\ (r, s_2) \wedge \\
\qquad r \neq \texttt{Rbreak} \Rightarrow \\
\qquad P\ (\texttt{For}\ e_1\ e_2\ t, s)\ (r, s_2)) \Rightarrow \\
\quad \forall ts\ rs.\ ts \Downarrow_{\mathsf{t}} rs \Rightarrow P\ ts\ rs
\end{array}
$$

It has one rule for each case in the relation

Six cases for For!

Observational semantics with I/O

Defining the observational semantics when there is I/O.

$$
\begin{array}{l}
\texttt{semantics}\ t\ \mathit{input}\ (\texttt{Terminate}\ \mathit{io\_trace}) \iff \\
\exists\, c\ \mathit{nd}\ i\ s\,. \\
\quad (\texttt{sem\_t}\ (\texttt{init\_st}\ c\ \mathit{nd}\ \mathit{input})\ t = (\texttt{Rval}\ i, s)) \wedge \\
\quad (\texttt{FILTER ISL}\ s.\texttt{io\_trace} = \mathit{io\_trace}) \\
\texttt{semantics}\ t\ \mathit{input}\ \texttt{Crash} \iff \\
\exists\, c\ \mathit{nd}\ r\ s\,. \\
\quad (\texttt{sem\_t}\ (\texttt{init\_st}\ c\ \mathit{nd}\ \mathit{input})\ t = (r, s)) \wedge \\
\quad ((r = \texttt{Rbreak}) \vee (r = \texttt{Rfail})) \\
\texttt{semantics}\ t\ \mathit{input}\ (\texttt{Diverge}\ \mathit{io\_trace}) \iff \\
\exists\, \mathit{nd}\,. \\
\quad (\forall\, c.\ \exists\, s.\ \texttt{sem\_t}\ (\texttt{init\_st}\ c\ \mathit{nd}\ \mathit{input})\ t = (\texttt{Rtimeout}, s)) \wedge \\
\quad (\mathit{io\_trace} = \\
\quad\ \bigvee c\,. \\
\qquad \texttt{fromList} \\
\qquad\quad (\texttt{FILTER ISL}\ (\texttt{SND}\ (\texttt{sem\_t}\ (\texttt{init\_st}\ c\ \mathit{nd}\ \mathit{input})\ t)).\texttt{io\_trace}))
\end{array}
$$