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
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.
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.”

“[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

Compiles C source code to assembly.

Has good performance numbers

Proved correct in Coq.

Leroy et al. Source: 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).

https://cakeml.org/

later lecture zooms in
A spectrum

Verified compilers

- robust, inflexible
- proved to always work correctly

Proof-producing compilers

- more flexible, but can be fragile
- produces a proof for each run

Pilsner

- CompCert C compiler
- CakeML compiler
- CompCert TSO

Fiat

- Cogent
  - Translation validation for a verified OS kernel
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
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

but feel free to ask
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)

\[ \text{HOL} = \text{lambda calculus with simple ML-like types} \]

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:

\[ \vdash \text{PERM} \ L_1 \ L_2 \iff \forall x. \ \text{FILTER} \ ((=) \ x) \ L_1 = \text{FILTER} \ ((=) \ x) \ L_2 \]
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):

FILTER P [] = []
FILTER P (h::t) = if P h then h::FILTER P t else FILTER P t

Inductive relations (also co-inductive relations):

(S1) \begin{align*}
(t_1,s) \Downarrow_t (Rval\ n_1,s_1) \\
(t_2,s_1) \Downarrow_t r \\
(\text{Seq } t_1\ t_2,s) \Downarrow_t r
\end{align*}

(S2) \begin{align*}
(t_1,s) \Downarrow_t (r,s_1) \\
\neg\text{is\_Rval } r \\
(\text{Seq } t_1\ t_2,s) \Downarrow_t (r,s_1)
\end{align*}
Ingredient 2: Language models
Syntax

The abstract syntax is defined as a datatype.

A toy source language:

\[
\begin{align*}
  t &= \text{Dec} \ \text{string} \ t \\
   &\mid \text{Exp} \ e \\
   &\mid \text{Break} \\
   &\mid \text{Seq} \ t \ t \\
   &\mid \text{If} \ e \ t \ t \\
   &\mid \text{For} \ e \ e \ t \\
  \\
  e &= \text{Var} \ \text{string} \\
   &\mid \text{Num} \ \text{int} \\
   &\mid \text{Add} \ e \ e \\
   &\mid \text{Assign} \ \text{string} \ e
\end{align*}
\]

- 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:

\[
\begin{align*}
t &= \text{Dec} \text{ string } t \\
  &| \quad \text{Exp } e \\
  &| \quad \text{Break} \\
  &| \quad \text{Seq } t \ t \\
  &| \quad \text{If } e \ t \ t \\
  &| \quad \text{For } e \ e \ t
\end{align*}
\]

\[
\begin{align*}
e &= \text{Var} \text{ string} \\
  &| \quad \text{Num} \text{ int} \\
  &| \quad \text{Add } e \ e \\
  &| \quad \text{Assign} \text{ string } e
\end{align*}
\]

A toy target language:

\[
\begin{align*}
\text{inst} &= \text{Add} \text{ reg } \text{ reg } \text{ reg} \\
  &| \quad \text{Int} \text{ reg } \text{ int} \\
  &| \quad \text{Jmp} \text{ offset} \\
  &| \quad \text{JmpIf} \text{ reg } \text{ offset}
\end{align*}
\]

an assembly program is a list of instructions
Operational Semantics — the options

There are 4 main variants one can choose from.

- **Relational**
  - Used for concurrency
  - Common in ACL2

- **Functional**
  - CakeML uses this style
  - Avoids treating diverging behaviours separately

- **Small-step**
  - Most common for compiler proofs

- **Big-step**
  - Requires defining:
    - Inductive rel. for terminating behaviours
    - Co-inductive rel. for diverging behaviours
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)) = ... 
```

- Expression evaluation returns \textsf{Rval} or \textsf{Rfail} and new state
- State is a simple assoc-list
- Assignments update the state
- Expression evaluation returns \textsf{Rval} or \textsf{Rfail} and new state
Towards a functional big-step semantics (2)

... and the evaluation of statements (type t):

```sml
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.

fun run_e s (Var x) =  
  (case lookup x s of  
    NONE => (Rfail, s)  
    | SOME v => (Rval v, s))

fun run_e s (Num i) = (Rval i, s)

fun run_e s (Assign (x, e)) =  
  (case run_e s e of  
    (Rval n1, s1) => (Rval n1, (x, n1)::s1)  
    | r => r)

fun run_e s (Add (e1, e2)) = ...

Below, evaluation of a **Break** statement returns **Rbreak**, which is propagated to the enclosing **For** loop. A **For** loop returns a normal **Rval** result if the body of the loop returns **Rbreak**.

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)
| 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))

These SML functions make use of catch-all patterns in case-expressions in order to conveniently propagate non-**Rval** results. We use the same approach in our functional semantics (§2.3) to keep them concise. The case expressions above are idiomatic for SML, but in a language with syntactic support for monadic computations, such as Haskell with do-notation, one would package the propagation of exceptional results inside a monadic bind operator.

... but now this recursive call introduces potential non-termination
Why is non-terminination an issue?

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

\[ f \ n = f \ n + 1 \]

then \[ f \ n - f \ n = f \ n + 1 - f \ n \]

and \[ 0 = 1 \]
We can force our functions to terminate by inserting a clock.

**ML:**
\[
\text{run_t} \ s \ (\text{For} \ (e_1, e_2, t)) = \\
\text{...} \\
\text{(Rval } _, \ s_3) \Rightarrow \text{run_t} \ s_3 \ (\text{For} \ (e_1, e_2, t))
\]

**HOL:**
\[
\text{eval_t} \ s \ (\text{For} \ e_1 \ e_2 \ t) = \\
\text{...} \\
\text{(Rval } _, s_3) \Rightarrow \\
\text{if } s_3.\text{clock} \neq 0 \text{ then} \\
\text{eval_t} \ (\text{dec_clock} \ s_3) \ (\text{For} \ e_1 \ e_2 \ t) \\
\text{else} \ (\text{Rtimeout}, s_3)
\]

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:**

\[
\begin{align*}
\text{eval}_t \ s \ (\text{Exp} \ e) &= \text{eval}_e \ s \ e \\
\text{eval}_t \ s \ (\text{Dec} \ x \ t) &= \text{eval}_t \ (\text{store} \ x \ s) \\
\text{eval}_t \ s \ \text{Break} &= (\text{Rbreak}, s) \\
\text{eval}_t \ s \ (\text{Seq} \ t_1 \ t_2) &= \\
&\quad \text{case} \ \text{eval}_t \ s \ t_1 \ \text{of} \\
&\qquad (\text{Rval}, s_1) \Rightarrow \text{eval}_t \ s_1 \ t_2 \\
&\qquad | \ r \Rightarrow r \\
\text{eval}_t \ s \ (\text{If} \ e \ t_1 \ t_2) &= \\
&\quad \text{case} \ \text{eval}_e \ s \ e \ \text{of} \\
&\qquad (\text{Rval} \ n_1, s_1) \Rightarrow \text{eval}_t \ s_1 \ (\text{if} \ n_1 = 0 \ \text{then} \ t_2 \ \text{else} \ t_1) \\
&\qquad | \ r \Rightarrow r
\end{align*}
\]

... 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.

\[
\text{semantics } t =
\begin{align*}
\text{if } & \exists c \; v \; s. \quad \text{sem}_t (s\text{\_with\_clock } c) \; t = (\text{Rval } v,s) \quad \text{then Terminate} \\
\text{else if } & \forall c. \; \exists s. \quad \text{sem}_t (s\text{\_with\_clock } c) \; t = (\text{Rtimeout},s) \quad \text{then Diverge} \\
\text{else } & \text{Crash}
\end{align*}
\]

- **Terminates if there is a clock that is sufficient**
- **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

\[
\begin{align*}
\text{phase1 (Exp } e) &= \text{Exp } e \\
\text{phase1 (Dec } x \ t) &= \text{Seq (Exp (Assign } x \ (\text{Num } 0))) \ (\text{phase1 } t) \\
\text{phase1 Break} &= \text{Break} \\
\text{phase1 (Seq } t_1 \ t_2) &= \text{Seq (phase1 } t_1) \ (\text{phase1 } t_2) \\
\text{phase1 (If } e \ t_1 \ t_2) &= \text{If } e \ (\text{phase1 } t_1) \ (\text{phase1 } t_2) \\
\text{phase1 (For } e_1 \ e_2 \ t) &= \text{Loop (If } e_1 \ (\text{Seq (phase1 } t) \ (\text{Exp } e_2)) \ \text{Break)}
\end{align*}
\]

where \( \text{Loop } t = \text{For (Num } 1) \ (\text{Num } 1) \ t \)

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
\[ \vdash \forall P. \]
\[
(\forall s e. \ P \ s \ (\text{Exp } e)) \land \\
(\forall s x t. \ P \ (\text{store_var } x \ 0 \ s) \ t \ \Rightarrow \ P \ s \ (\text{Dec } x \ t)) \land \\
(\forall s. \ P \ s \ \text{Break}) \land \\
(\forall s \ t_1 \ t_2. \\
(\forall v_2 \ s_1 \ v_5. \\
\quad (\text{eval}_t \ s \ t_1 = (v_2, s_1)) \land (v_2 = \text{Rval}) \land \ P \ s \ t_1 \ \Rightarrow \\
\quad P \ s \ (\text{Seq } t_1 \ t_2)) \land \\
(\forall s e \ t_1 \ t_2. \\
(\forall v_2 \ s_1 \ n_1. \\
\quad (\text{eval}_e \ s \ e = (v_2, s_1)) \land (v_2 = \text{Rval}) \land \ P \ s_1 \ (\text{if } n_1 = 0 \ \text{then } t_2 \ \text{else } t_1)) \ \Rightarrow \\
\quad P \ s \ (\text{If } e \ t_1 \ t_2)) \land \\
(\forall s e_1 \ e_2 \ t. \\
(\forall v_4 \ s_1 \ n_1 \ v_3 \ s_2 \ v_7 \ v_2 \ s_3 \ v_5. \\
\quad (\text{eval}_e \ s \ e_1 = (v_4, s_1)) \land (v_4 = \text{Rval} \ n_1) \land n_1 \neq 0 \land \\
\quad (\text{eval}_t \ s_1 \ t = (v_3, s_2)) \land (v_3 = \text{Rval} \ v_7) \land \\
\quad (\text{eval}_e \ s_2 \ e_2 = (v_2, s_3)) \land (v_2 = \text{Rval} \ v_5) \land s_3.\text{clock} \neq 0 \ \Rightarrow \\
\quad P \ (\text{dec_clock } s_3) \ (\text{For } e_1 \ e_2 \ t)) \land \\
(\forall v_4 \ s_1 \ n_1. \\
\quad (\text{eval}_e \ s \ e_1 = (v_4, s_1)) \land (v_4 = \text{Rval} \ n_1) \land n_1 \neq 0 \ \Rightarrow \\
\quad P \ s_1 \ t) \ \Rightarrow \\
\quad P \ s \ (\text{For } e_1 \ e_2 \ t)) \ \Rightarrow \\
\forall v \ v_1. \ P \ v \ v_1
\]
Simulation proof

Particularly simple for phase 1:
\[ \forall s \ t. \ \text{eval}_t \ s \ (\text{phase1} \ t) = \text{eval}_t \ s \ t \]

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:
\[ \forall t. \ \text{semantics} \ (\text{phase1} \ t) = \text{semantics} \ t \]
Simulation theorems *in general*

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

\[
\text{evaluate code } s_1 = (\text{res}, s_2) \land \text{res} \neq \text{Rfail} \land \\
\text{state}_\text{rel} \ s_1 \ t_1 \Rightarrow \\
\exists t_2. \text{evaluate (compile code) } t_1 = (\text{res}, t_2) \land \\
\text{state}_\text{rel} \ s_2 \ t_2
\]

*non-failing* evaluation of source intermediate language (IL)

... w.r.t. some relation between states (*state_rel*)
Simulation theorems *in general*

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

\[
\text{evaluate } \text{code } s_1 = (\text{res},s_2) \land \text{res} \neq \text{Rfail} \land \\
\text{state}_\text{rel} \ s_1 \ t_1 \Rightarrow \\
\exists t_2. \ \text{evaluate } (\text{compile code}) \ t_1 = (\text{res},t_2) \land \\
\text{state}_\text{rel} \ s_2 \ t_2
\]

*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:

\[
\text{evaluate } \text{code } s_1 = (\text{res}, s_2) \land \text{res} \neq \text{Rfail} \land \text{state_rel } s_1 \ t_1 \Rightarrow \\
\exists t_2. \text{evaluate (compile code) } t_1 = (\text{res}, t_2) \land \text{state_rel } s_2 \ t_2
\]

*Sufficient to prove observational equivalence* for both terminating and diverging runs.
Phase 2 simulation theorem

\[ \forall s, e, n, res : s_1. \\
(\text{eval}_t s e = (res, s_1)) \land res \neq \text{Rfail} \land \text{phase2_subset } e \land \\
\text{possible_var_name } n s_1.\text{store} \sqsubseteq t.\text{store} \land \\
(t.\text{clock} = s_1.\text{clock}) \land \text{strlen } n \Rightarrow \\
\exists t_1. \\
(\text{eval}_t t (\text{flatten}_t e n) = (res, t_1)) \land \\
s_1.\text{store} \sqsubseteq t_1.\text{store} \land (t_1.\text{clock} = s_1.\text{clock}) \land \\
\text{possible_var_name } n s_1.\text{store} \land \\
\forall k, v. \\
\text{possible_var_name } k s.\text{store} \land t_\text{max } e < \text{strlen } k \land \\
\text{strlen } k < \text{strlen } n \land (\text{lookup } t.\text{store } k = \text{SOME } v) \Rightarrow \\
(\text{lookup } t_1.\text{store } k = \text{SOME } v) \]

source IL

not failing

restrict language syntax following phase 1

target IL

compiler

state relation

extra property
Composing top-level theorems

Each phase maintains observational equivalence:

\[
\text{semantics (phase1 } t) = \text{semantics } t
\]

\[
\text{semantics } t \neq \text{Crash} \land \text{phase2\_subset } t \Rightarrow (\text{semantics (phase2 } t) = \text{semantics } t)
\]

\[
\text{semantics } t \neq \text{Crash} \land \text{phase3\_subset } t \Rightarrow (\text{asm\_semantics (phase3 } 0 0 t) = \text{semantics } t)
\]

Here: \( \text{compile } t = \text{phase3 } 0 0 (\text{phase2 (phase1 } t)) \)
Composing top-level theorems

Result:

\[ \forall t. \ syntax\_ok\ t \implies (asm\_semantics\ (compile\ t) = semantics\ t) \]

where \( compile\ t = phase3\ 0\ 0\ (phase2\ (phase1\ t)) \)

- lemma: correct syntax implies no Crashes
- for ML: type-correct program implies no Crash
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:

\[
\text{evaluate code } s1 = (res, s2) \land res \neq \text{Rfail} \land \\
\text{state}_\text{rel } s1 \ t1 \Rightarrow \\
\exists t2. \ \text{evaluate} (\text{compile code}) t1 = (res, t2) \land \\
\text{state}_\text{rel } s2 \ t2
\]

in order to prove observational equivalence, i.e.

\[\vdash \forall t. \ \text{syntax}_\text{ok } t \Rightarrow (\text{asm}_\text{semantics} (\text{compile } t) = \text{semantics } t)\]
Extra slides
Comparing functional with relation big-step

In the functional version, Seq was specified by:

\[
\text{eval}_t \ s \ (\text{Seq} \ t_1 \ t_2) = \\
\text{case} \ \text{eval}_t \ s \ t_1 \ \text{of} \\
\quad (\text{Rval} \ _, s_1) \Rightarrow \text{eval}_t \ s_1 \ t_2 \\
\mid r \Rightarrow r
\]

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

\[
\frac{(t_1, s) \downarrow_t (\text{Rval} \ n_1, s_1)}{(t_2, s_1) \downarrow_t r}{(\text{Seq} \ t_1 \ t_2, s) \downarrow_t r} \quad \text{(S1)}
\]

\[
\frac{(t_1, s) \downarrow_t (\text{Rval} \ n_1, s_1)}{(t_2, s_1) \uparrow_t}{(\text{Seq} \ t_1 \ t_2, s) \uparrow_t} \quad \text{(S1')} 
\]

\[
\frac{\neg \text{is_Rval} \ r}{(\text{Seq} \ t_1 \ t_2, s) \downarrow_t (r, s_1)} \quad \text{(S2)}
\]

\[
\frac{(t_1, s) \downarrow_t (\text{Rval} \ n_1, s_1)}{(t_2, s_1) \uparrow_t}{(\text{Seq} \ t_1 \ t_2, s) \uparrow_t} \quad \text{(S2')} 
\]
\[
\forall s \ s_1 \ e_1 \ e_2 \ t.
\]
\[
(e_1, s) \downarrow_e (\text{Rval } 0, s_1) \Rightarrow P \ (\text{For } e_1 \ e_2 \ t, s) \ (\text{Rval } 0, s_1)
\]
\[
(\forall s \ s_1 \ e_1 \ e_2 \ t \ r.
\]
\[
(e_1, s) \downarrow_e (r, s_1) \land \neg \text{is_Rval } r \Rightarrow P \ (\text{For } e_1 \ e_2 \ t, s) \ (r, s_1)
\]
\[
(\forall s \ s_1 \ s_2 \ s_3 \ e_1 \ e_2 \ t \ n_1 \ n_2 \ n_3 \ r.
\]
\[
(e_1, s) \downarrow_e (\text{Rval } n_1, s_1) \land n_1 \neq 0 \land P \ (t, s_1) \ (\text{Rval } n_2, s_2)
\]
\[
(e_2, s_2) \downarrow_e (\text{Rval } n_3, s_3) \land P \ (\text{For } e_1 \ e_2 \ t, s_3) \ r \Rightarrow
\]
\[
P \ (\text{For } e_1 \ e_2 \ t, s) \ r
\]
\[
(\forall s \ s_1 \ s_2 \ e_1 \ e_2 \ t \ n_1.
\]
\[
(e_1, s) \downarrow_e (\text{Rval } n_1, s_1) \land n_1 \neq 0 \land P \ (t, s_1) \ (\text{Rbreak}, s_2) \Rightarrow
\]
\[
P \ (\text{For } e_1 \ e_2 \ t, s) \ (\text{Rval } 0, s_2)
\]
\[
(\forall s \ s_1 \ s_2 \ s_3 \ e_1 \ e_2 \ t \ n_1 \ n_2 \ r.
\]
\[
(e_1, s) \downarrow_e (\text{Rval } n_1, s_1) \land n_1 \neq 0 \land P \ (t, s_1) \ (\text{Rval } n_2, s_2) \land
\]
\[
(e_2, s_2) \downarrow_e (r, s_3) \land \neg \text{is_Rval } r \Rightarrow
\]
\[
P \ (\text{For } e_1 \ e_2 \ t, s) \ (r, s_3)
\]
\[
(\forall s \ s_1 \ s_2 \ e_1 \ e_2 \ t \ n_1 \ r.
\]
\[
(e_1, s) \downarrow_e (\text{Rval } n_1, s_1) \land n_1 \neq 0 \land P \ (t, s_1) \ (r, s_2) \land
\]
\[
r \neq \text{Rbreak} \Rightarrow
\]
\[
P \ (\text{For } e_1 \ e_2 \ t, s) \ (r, s_2)
\]
\[
(\forall ts \ rs. \ ts \downarrow_t rs \Rightarrow P \ ts \ rs
\]

**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.

```
semantics t input (Terminate io_trace) ⇔
∃ c nd i s.
  (sem_t (init_st c nd input) t = (Rval i,s)) ∧
  (FILTER ISL s.io_trace = io_trace)

semantics t input Crash ⇔
∃ c nd r s.
  (sem_t (init_st c nd input) t = (r,s)) ∧
  ((r = Rbreak) ∨ (r = Rfail))

semantics t input (Diverge io_trace) ⇔
∃ nd.
  (∀ c. ∃ s. sem_t (init_st c nd input) t = (Rtimeout,s)) ∧
  (io_trace =
    ∨ c.
    fromList
      (FILTER ISL (SND (sem_t (init_st c nd input) t)).io_trace))
```