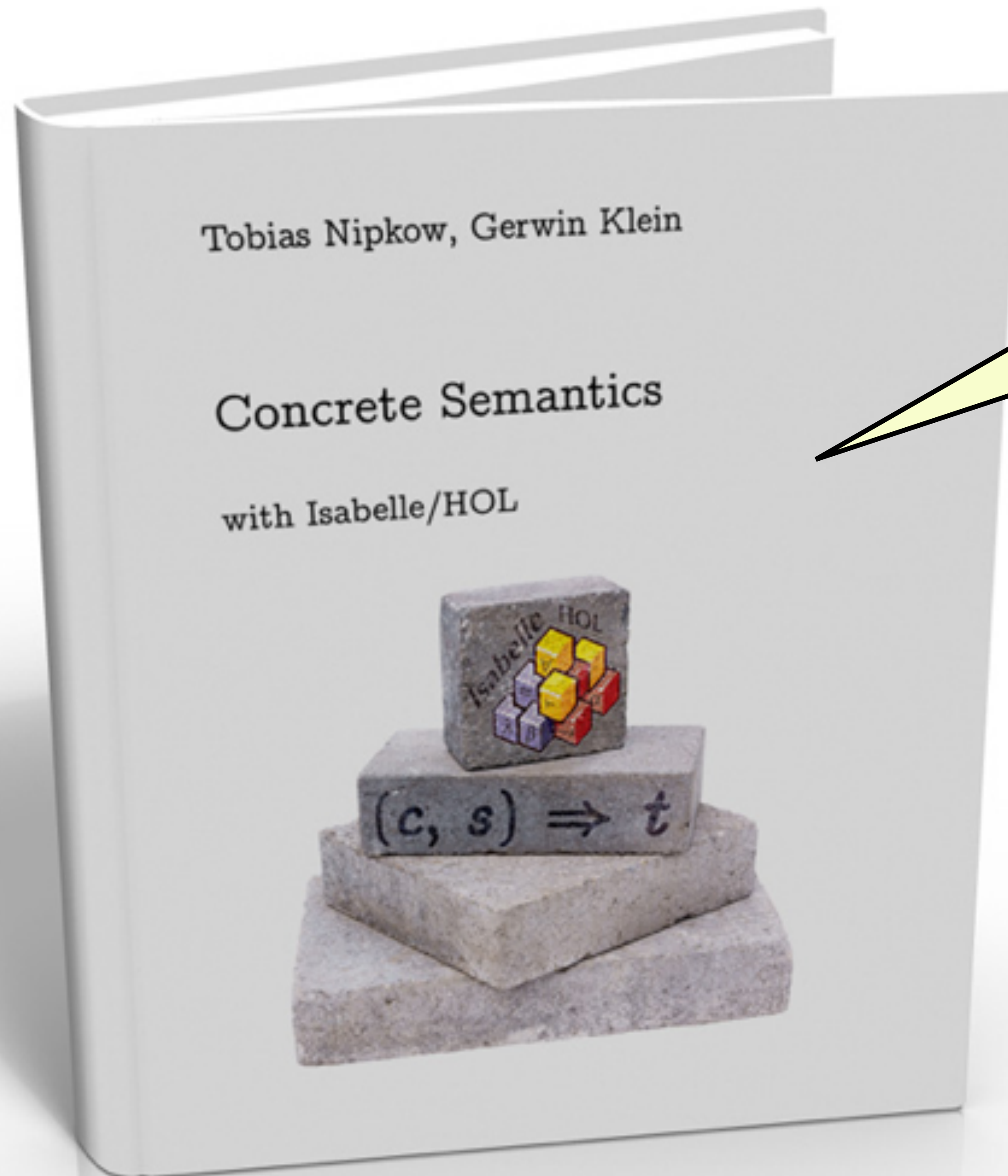
Functional Big-step Semantics

FM talk, 11 Mar 2015

Magnus Myréen

Books



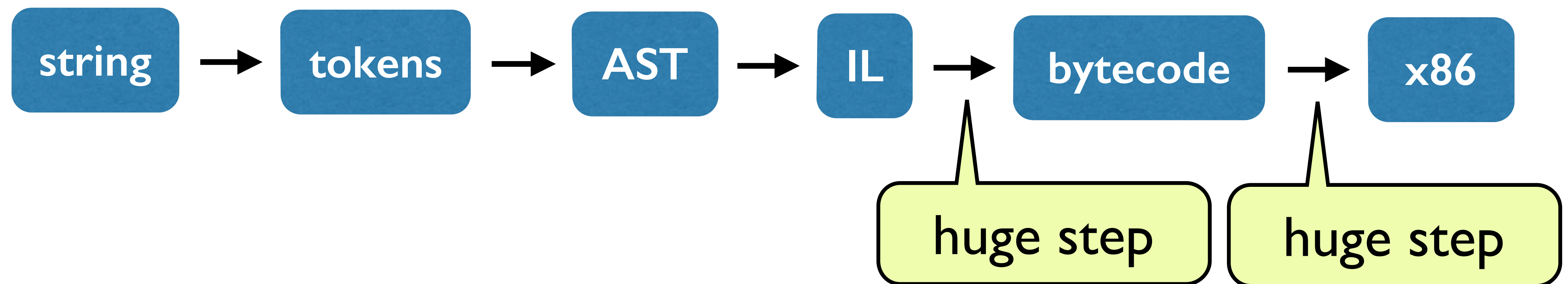
Big-step semantics are defined as inductively defined relation.

Functions are better!

me

Context: CakeML verified compiler

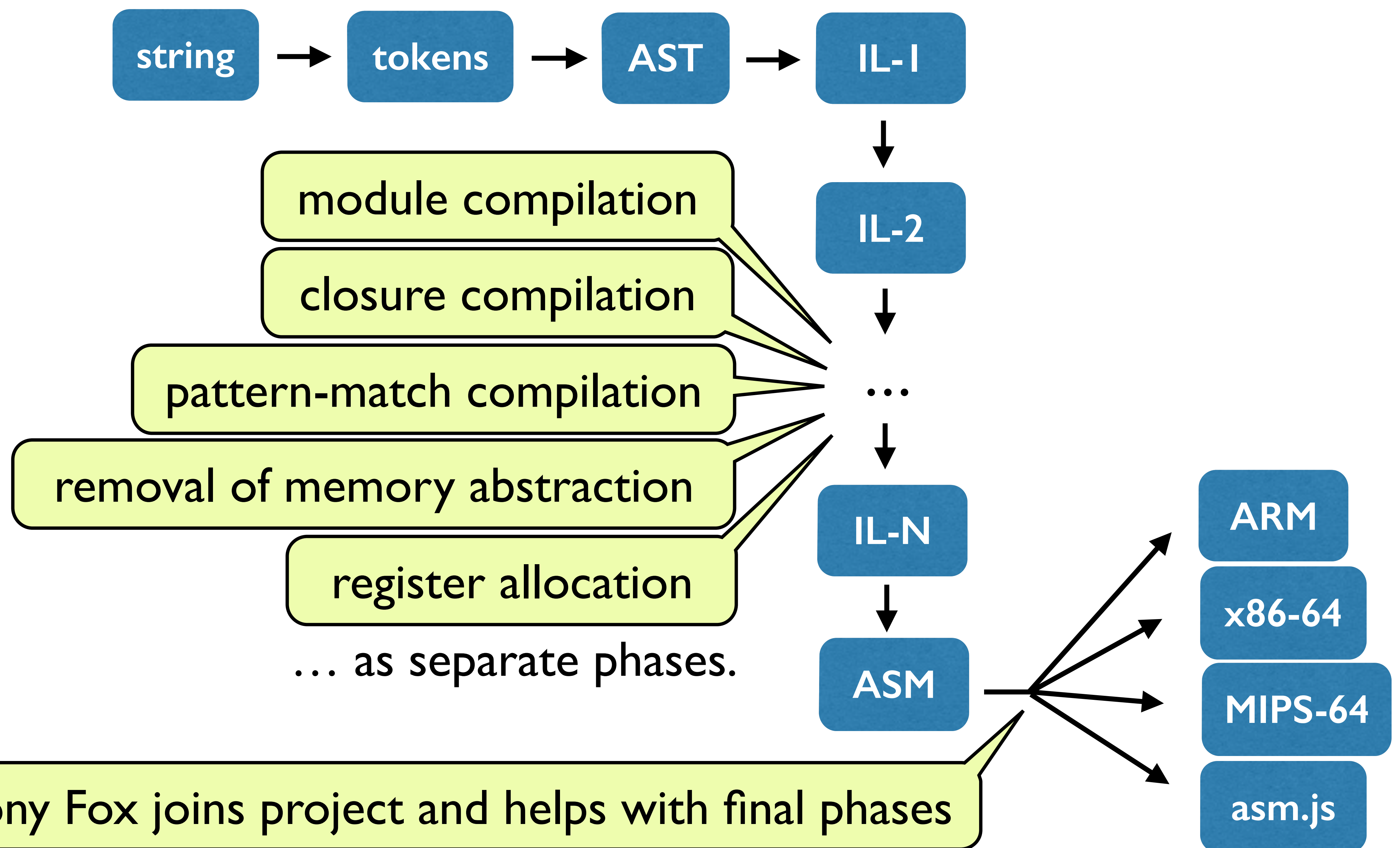
Old compiler:



Bytecode simplified proofs of read-eval-print loop, but made optimisation impossible.

Context: CakeML verified compiler

Refactored compiler: split into more conventional compiler phases



… a different example.

Pretty-Big-Step Semantics

Arthur Charguéraud

INRIA

arthur.chargueraud@inria.fr

ESOP'12

Abstract. In spite of the popularity of small-step semantics, big-step semantics remain used by many researchers. However, big-step semantics suffer from a serious duplication problem, which appears as soon as the semant[...] many prem[...] This duplic[...] to full-blown languages, results in formal definitions growing far bigger than necessary. Moreover, it leads to unsatisfactory redundancy in proofs. In this paper, we address the problem by introducing pretty-big-step semantics. Pretty-big-step semantics preserve the spirit of big-step

Example language with C-like For and Break

How should I define a language?

Syntax:

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

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

Datatype for results:

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

How should I define a language?

Semantics as an interpreter in SML:

```
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 (Add (e1, e2)) =
     (case run_e s e1 of
        (Rval n1, s1) =>
           (case run_e s1 e2 of
              (Rval n2, s2) => (Rval (n1+n2), s2)
            | r => r)
      | r => r)
  | run_e s (Assign (x, e)) =
     (case run_e s e of
        (Rval n1, s1) => (Rval n1, (x,n1)::s1)
      | r => r)
```

continues …

How should I define a language?

Semantics as an interpreter in SML (continued):

```
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
        (Rval n1,                         else t1)
      | r => r)
  | run_t s (For (e1, e2, t)) =
     (case run_e s e1 of
        (Rval n1, s1) =>
          if n1 = 0 then (
           (case run_t s1
              (Rval _, s2)
                (case run_
                   (Rval _, s3) => run_t s3 (For (e1, e2, t))
                 | r => r)
            | (Rbreak, s2) => (Rval 0, s2)
            | r => r)
      | r => r)
```

Is this a good definition?

For HOL proofs, unfortunately not.

This function can't be defined in HOL.

Define big-step semantics

(conventional approach)

$$\frac{(t_1\,,s)\ \Downarrow_t\ (\texttt{Rval}\ \ n_1\,,s_1) \qquad (t_2\,,s_1)\ \Downarrow_t\ r}{(\texttt{Seq}\ \ t_1\ \ t_2\,,s)\ \Downarrow_t\ r}$$

$$\frac{(t_1\,,s)\ \Downarrow_t\ (r\,,s_1) \qquad \neg\texttt{is_Rval}\ \ r}{(\texttt{Seq}\ \ t_1\ \ t_2\,,s)\ \Downarrow_t\ (r\,,s_1)}$$

$$\frac{}{(\texttt{Break}\,,s)\ \Downarrow_t\ (\texttt{Rbreak}\,,s)}$$

Problem of duplication...

Define big-step semantics

(conventional approach)

$$\frac{(e_1, s) \Downarrow_e (\texttt{Rval}\ 0, s_1)}{(\texttt{For}\ e_1\ e_2\ t, s) \Downarrow_t (\texttt{Rval}\ 0, s_1)}$$

$$\frac{(e_1, s) \Downarrow_e (r, s_1) \quad \neg\texttt{is_Rval}\ r}{(\texttt{For}\ e_1\ e_2\ t, s) \Downarrow_t (r, s_1)}$$

$$\frac{(e_1, s) \Downarrow_e (\texttt{Rval}\ n_1, s_1) \quad n_1 \neq 0 \quad (t, s_1) \Downarrow_t (\texttt{Rval}\ n_2, s_2) \quad (e_2, s_2) \Downarrow_e (\texttt{Rval}\ n_3, s_3) \quad (\texttt{For}\ e_1\ e_2\ t, s_3) \Downarrow_t r}{(\texttt{For}\ e_1\ e_2\ t, s) \Downarrow_t r}$$

$$\frac{(e_1, s) \Downarrow_e (\texttt{Rval}\ n_1, s_1) \quad n_1 \neq 0 \quad (t, s_1) \Downarrow_t (\texttt{Rbreak}, s_2)}{(\texttt{For}\ e_1\ e_2\ t, s) \Downarrow_t (\texttt{Rval}\ 0, s_2)}$$

$$\frac{(e_1, s) \Downarrow_e (\texttt{Rval}\ n_1, s_1) \quad n_1 \neq 0 \quad (t, s_1) \Downarrow_t (\texttt{Rval}\ n_2, s_2) \quad (e_2, s_2) \Downarrow_e (r, s_3) \quad \neg\texttt{is_Rval}\ r}{(\texttt{For}\ e_1\ e_2\ t, s) \Downarrow_t (r, s_3)}$$

$$\frac{(e_1, s) \Downarrow_e (\texttt{Rval}\ n_1, s_1) \quad n_1 \neq 0 \quad (t, s_1) \Downarrow_t (r, s_2) \quad \neg\texttt{is_Rval}\ r \quad r \neq \texttt{Rbreak}}{(\texttt{For}\ e_1\ e_2\ t, s) \Downarrow_t (r, s_2)}$$

Not suitable for proofs of divergence preservation.

Define big-step semantics

(conventional approach)

Induction theorem:

$$
\begin{array}{l}
\vdash (\forall s\ e\ r.\ (e,s) \Downarrow_e r \Rightarrow P\ (\texttt{Exp}\ e,s)\ r) \land \\
\quad (\forall s\ x\ t\ r. \\
\quad\quad P\ (t,s\ \texttt{with store := } s\texttt{.store |+ }(x,0))\ r \Rightarrow \\
\quad\quad P\ (\texttt{Dec}\ x\ t,s)\ r) \land (\forall s.\ P\ (\texttt{Break},s)\ (\texttt{Rbreak},s)) \land \\
\quad (\forall s\ s_1\ t_1\ t_2\ n_1\ r. \\
\quad\quad P\ (t_1,s)\ (\texttt{Rval}\ n_1,s_1) \land P\ (t_2,s_1)\ r \Rightarrow \\
\quad\quad P\ (\texttt{Seq}\ t_1\ t_2,s)\ r) \land \\
\quad (\forall s\ s_1\ t_1\ t_2\ r. \\
\quad\quad P\ (t_1,s)\ (r,s_1) \land \neg\texttt{is_Rval}\ r \Rightarrow P\ (\texttt{Seq}\ t_1\ t_2,s)\ (r,s_1)) \land \\
\quad (\forall s\ s_1\ e\ t_1\ t_2\ r. \\
\quad\quad (e,s) \Downarrow_e (\texttt{Rval}\ 0,s_1) \land P\ (t_2,s_1)\ r \Rightarrow P\ (\texttt{If}\ e\ t_1\ t_2,s)\ r) \land \\
\quad (\forall s\ s_1\ e\ t_1\ t_2\ n\ r. \\
\quad\quad (e,s) \Downarrow_e (\texttt{Rval}\ n,s_1) \land n \neq 0 \land P\ (t_1,s_1)\ r \Rightarrow \\
\quad\quad P\ (\texttt{If}\ e\ t_1\ t_2,s)\ r) \land \\
\quad (\forall s\ s_1\ e\ t_1\ t_2\ r. \\
\quad\quad (e,s) \Downarrow_e (r,s_1) \land \neg\texttt{is_Rval}\ r \Rightarrow P\ (\texttt{If}\ e\ t_1\ t_2,s)\ (r,s_1)) \land \\
\quad (\forall s\ s_1\ e_1\ e_2\ t. \\
\quad\quad (e_1,s) \Downarrow_e (\texttt{Rval}\ 0,s_1) \Rightarrow P\ (\texttt{For}\ e_1\ e_2\ t,s)\ (\texttt{Rval}\ 0,s_1)) \land \\
\quad (\forall s\ s_1\ e_1\ e_2\ t\ r. \\
\quad\quad (e_1,s) \Downarrow_e (r,s_1) \land \neg\texttt{is_Rval}\ r \Rightarrow P\ (\texttt{For}\ e_1\ e_2\ t,s)\ (r,s_1)) \land \\
\quad (\forall s\ s_1\ s_2\ s_3\ e_1\ e_2\ t\ n_1\ n_2\ n_3\ r. \\
\quad\quad (e_1,s) \Downarrow_e (\texttt{Rval}\ n_1,s_1) \land n_1 \neq 0 \land P\ (t,s_1)\ (\texttt{Rval}\ n_2,s_2) \land \\
\quad\quad (e_2,s_2) \Downarrow_e (\texttt{Rval}\ n_3,s_3) \land P\ (\texttt{For}\ e_1\ e_2\ t,s_3)\ r \Rightarrow \\
\quad\quad P\ (\texttt{For}\ e_1\ e_2\ t,s)\ r) \land \\
\quad (\forall s\ s_1\ s_2\ e_1\ e_2\ t\ n_1. \\
\quad\quad (e_1,s) \Downarrow_e (\texttt{Rval}\ n_1,s_1) \land n_1 \neq 0 \land P\ (t,s_1)\ (\texttt{Rbreak},s_2) \Rightarrow \\
\quad\quad P\ (\texttt{For}\ e_1\ e_2\ t,s)\ (\texttt{Rval}\ 0,s_2)) \land \\
\quad (\forall s\ s_1\ s_2\ s_3\ e_1\ e_2\ t\ n_1\ n_2\ r. \\
\quad\quad (e_1,s) \Downarrow_e (\texttt{Rval}\ n_1,s_1) \land n_1 \neq 0 \land P\ (t,s_1)\ (\texttt{Rval}\ n_2,s_2) \land \\
\quad\quad (e_2,s_2) \Downarrow_e (r,s_3) \land \neg\texttt{is_Rval}\ r \Rightarrow \\
\quad\quad P\ (\texttt{For}\ e_1\ e_2\ t,s)\ (r,s_3)) \land \\
\quad (\forall s\ s_1\ s_2\ e_1\ e_2\ t\ n_1\ r. \\
\quad\quad (e_1,s) \Downarrow_e (\texttt{Rval}\ n_1,s_1) \land n_1 \neq 0 \land P\ (t,s_1)\ (r,s_2) \land \neg\texttt{is_Rval}\ r \land \\
\quad\quad r \neq \texttt{Rbreak} \Rightarrow \\
\quad\quad P\ (\texttt{For}\ e_1\ e_2\ t,s)\ (r,s_2)) \Rightarrow \\
\quad \forall t\ s\ r.\ (t,s) \Downarrow_t r \Rightarrow P\ (t,s)\ r
\end{array}
$$

Define big-step semantics

(conventional approach)

Induction theorem:

$$
\begin{array}{l}
(\forall s\ s_1\ e_1\ e_2\ t.\\
\quad (e_1,s) \Downarrow_e (\texttt{Rval}\ 0, s_1) \Rightarrow P\ (\texttt{For}\ e_1\ e_2\ t, s)\ (\texttt{Rval}\ 0, s_1)) \wedge\\
(\forall s\ s_1\ e_1\ e_2\ t\ r.\\
\quad (e_1,s) \Downarrow_e (r, s_1) \wedge \neg\texttt{is_Rval}\ r \Rightarrow P\ (\texttt{For}\ e_1\ e_2\ t, s)\ (r, s_1)) \wedge\\
(\forall s\ s_1\ s_2\ s_3\ e_1\ e_2\ t\ n_1\ n_2\ n_3\ r.\\
\quad (e_1,s) \Downarrow_e (\texttt{Rval}\ n_1, s_1) \wedge n_1 \neq 0 \wedge P\ (t, s_1)\ (\texttt{Rval}\ n_2, s_2) \wedge\\
\quad (e_2,s_2) \Downarrow_e (\texttt{Rval}\ n_3, s_3) \wedge P\ (\texttt{For}\ e_1\ e_2\ t, s_3)\ r \Rightarrow\\
\quad P\ (\texttt{For}\ e_1\ e_2\ t, s)\ r) \wedge\\
(\forall s\ s_1\ s_2\ e_1\ e_2\ t\ n_1.\\
\quad (e_1,s) \Downarrow_e (\texttt{Rval}\ n_1, s_1) \wedge n_1 \neq 0 \wedge P\ (t, s_1)\ (\texttt{Rbreak}, s_2) \Rightarrow\\
\quad P\ (\texttt{For}\ e_1\ e_2\ t, s)\ (\texttt{Rval}\ 0, s_2)) \wedge\\
(\forall s\ s_1\ s_2\ s_3\ e_1\ e_2\ t\ n_1\ n_2\ r.\\
\quad (e_1,s) \Downarrow_e (\texttt{Rval}\ n_1, s_1) \wedge n_1 \neq 0 \wedge P\ (t, s_1)\ (\texttt{Rval}\ n_2, s_2) \wedge\\
\quad (e_2,s_2) \Downarrow_e (r, s_3) \wedge \neg\texttt{is_Rval}\ r \Rightarrow\\
\quad P\ (\texttt{For}\ e_1\ e_2\ t, s)\ (r, s_3)) \wedge\\
(\forall s\ s_1\ s_2\ e_1\ e_2\ t\ n_1\ r.\\
\quad (e_1,s) \Downarrow_e (\texttt{Rval}\ n_1, s_1) \wedge n_1 \neq 0 \wedge P\ (t, s_1)\ (r, s_2) \wedge \neg\texttt{is_Rval}\ r \wedge\\
\quad r \neq \texttt{Rbreak} \Rightarrow\\
\quad P\ (\texttt{For}\ e_1\ e_2\ t, s)\ (r, s_2)) \Rightarrow\\
\forall t\ s\ r.\ (t,s) \Downarrow_t r \Rightarrow P\ (t,s)\ r
\end{array}
$$

A lot of duplication in proofs!

Hrmm…

I prefer the SML code…

Why can't it be used as the definition of the semantics?

It doesn't terminate for all inputs…

Making the interpreter terminate

```
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)
```

Making the interpreter terminate

```
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)
```

Making the interpreter terminate

```
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) =>
                      if !clock <= 0 then
                        raise TimeOutException
                      else
                        (clock := !clock - 1;
                         run_t s3 (For (e1, e2, t)))
                 | r => r)
            | (Rbreak, s2) => (Rval 0, s2)
            | r => r)
```

As a logic function

```
sem_t s (For e1 e2 t) =
    ...
      (Rval n1,s3) ⇒
        if s3.clock ≠ 0 then
          sem_t (dec_clock s3) (For e1 e2 t)
        else (Rtimeout,s3)
```

As a logic function

```
sem_t s (Exp e) = sem_e s e
sem_t s (Dec x t) = sem_t (store_var x 0 s) t
sem_t s Break = (Rbreak,s)
sem_t s (Seq t1 t2) =
case sem_t s t1 of
  (Rval v5,s1) ⇒ sem_t s1 t2
| r ⇒ r)
sem_t s (If e t1 t2) =
case sem_e s e of
  (Rval n1,s1) ⇒ sem_t s1 (if n1 = 0 then t2 else t1)
| r ⇒ r)
sem_t s (For e1 e2 t) =
case sem_e s e1 of
  (Rval 0,s1) ⇒ (Rval 0,s1)
| (Rval n1,s1) ⇒
    (case sem_t s1 t of
       (Rval n1,s2) ⇒
         (case sem_e s2 e2 of
            (Rval n1,s3) ⇒
              if s3.clock ≠ 0 then
                sem_t (dec_clock s3) (For e1 e2 t)
              else (Rtimeout,s3)
          | r ⇒ r)
     | (Rbreak,s2) ⇒ (Rval 0,s2)
     | r ⇒ r)
| r ⇒ r)
```

No duplication!

The auto-generated induction theorem

$$
\begin{array}{l}
\vdash (\forall s\ e.\ P\ s\ (\texttt{Exp}\ e)) \land \\
\quad (\forall s\ x\ t.\ P\ (\texttt{store_var}\ x\ 0\ s)\ t \Rightarrow P\ s\ (\texttt{Dec}\ x\ t)) \land \\
\quad (\forall s.\ P\ s\ \texttt{Break}) \land \\
\quad (\forall s\ t_1\ t_2. \\
\qquad (\forall s_1\ v_5.\ (\texttt{sem_t}\ s\ t_1 = (\texttt{Rval}\ v_5, s_1)) \Rightarrow P\ s_1\ t_2) \land P\ s\ t_1 \Rightarrow \\
\qquad P\ s\ (\texttt{Seq}\ t_1\ t_2)) \land \\
\quad (\forall s\ e\ t_1\ t_2. \\
\qquad (\forall s_1\ n_1. \\
\qquad\quad (\texttt{sem_e}\ s\ e = (\texttt{Rval}\ n_1, s_1)) \Rightarrow \\
\qquad\quad P\ s_1\ (\textbf{if}\ n_1 = 0\ \textbf{then}\ t_2\ \textbf{else}\ t_1)) \Rightarrow \\
\qquad P\ s\ (\texttt{If}\ e\ t_1\ t_2)) \land \\
\quad (\forall s\ e_1\ e_2\ t. \\
\qquad (\forall s_1\ n_1. \\
\qquad\quad (\texttt{sem_e}\ s\ e_1 = (\texttt{Rval}\ n_1, s_1)) \land n_1 \neq 0 \Rightarrow \\
\qquad\quad P\ s_1\ t \land \\
\qquad\quad \forall s_2\ n_2\ s_3\ n_3. \\
\qquad\qquad (\texttt{sem_t}\ s_1\ t = (\texttt{Rval}\ n_2, s_2)) \land \\
\qquad\qquad (\texttt{sem_e}\ s_2\ e_2 = (\texttt{Rval}\ n_3, s_3)) \land s_3.\mathsf{clock} \neq 0 \Rightarrow \\
\qquad\qquad P\ (\texttt{dec_clock}\ s_3)\ (\texttt{For}\ e_1\ e_2\ t)) \Rightarrow \\
\qquad P\ s\ (\texttt{For}\ e_1\ e_2\ t)) \Rightarrow \\
\quad \forall s\ t.\ P\ s\ t
\end{array}
$$

No duplication!

Logical clock for divergence pres.

Big-step semantics:

- has an optional clock component
- clock 'ticks' decrements every time a function is applied
- once clock hits zero, execution stops with a TimeOut

Why do this?

- because now big-step semantics describes both terminating and non-terminating evaluations

for every exp env clock

there is some result

either: Result or TimeOut

$$\forall exp\ \ env\ \ clock.\ \exists res.\ (exp, env, \text{Some}\ clock) \Downarrow_{\text{ev}}\ res$$

produced by the semantics

Divergence

Evaluation diverges if

$$\forall clock.\ (exp, env, \text{Some } clock) \Downarrow_{\text{ev}} \text{TimeOut}$$

for all clock values

TimeOut happens

Compiler correctness proved in conventional forward direction:

$$(exp, env) \Downarrow_{\text{ev}} val \implies$$
$$\text{“the code for } exp \text{ is installed in } bs \text{ etc.”} \implies$$
$$\exists bs'.\ bs \rightarrow^* bs' \wedge \text{“}bs' \text{ contains } val\text{”}$$

Bytecode has clock

… that stays in sync with CakeML clock

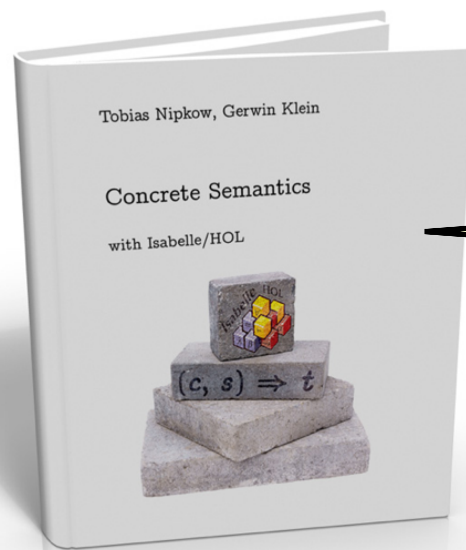
Theorem: bytecode diverges if and only if CakeML eval diverges

Non-determinism

How to handle it?

Partial solution: use oracle to factor out non-determinism

Summary



Big-step semantics are defined as inductively defined relation.

Functions are better!

me

Easier to read / understand

Avoid duplication

Better induction theorem

Proofs by rewriting (not covered here)

Naturally useful in proofs about divergence pres.

Down sides: must have clock,
non-deter requires special care.

Questions?