Semantic preservation for non-terminating programs

Lecture 13

MPhil ACS & Part III course, Functional Programming: Implementation, Specification and Verification

Magnus Myreen
Michaelmas term, 2013

From previous lectures

Correctness of ML-to-bytecode compilation:

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

From previous lectures

Correctness of ML-to-bytecode compilation:

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

Correctness of bytecode-to-machine-code compilation:

$bs \rightarrow^* bs' \implies$

```
{ PC (base + bs.pc) * BYTECODE (bs.stack,err) }
  base: to_mc bc.code
{ PC (base + bs'.pc) * BYTECODE (bs'.stack,err) v PC err * true }
```

Top-level correctness

Combination:

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

```
{ PC (base + bs.pc) * BYTECODE (bs.stack,err) }
  base: to_mc bc.code
{ PC (base + bs'.pc) * BYTECODE (bs'.stack,err) v PC err * true }
```

Top-level correctness

Combination:

if the big-step terminates with value

$(exp, env) \Downarrow_{\text{ev}} val \implies$

"the code for exp is installed in bs etc." $\implies$

$\exists bs'.$ "bs' contains val" $\wedge$

```
{ PC (base + bs.pc) * BYTECODE (bs.stack,err) }
  base: to_mc bc.code
{ PC (base + bs'.pc) * BYTECODE (bs'.stack,err) v PC err * true }
```

Top-level correctness

Combination:

if the big-step terminates with value

$(exp, env) \Downarrow_{\text{ev}} val \implies$

"the code for exp is installed in bs etc." $\implies$

$\exists bs'.$ "bs' contains val" $\wedge$

```
{ PC (base + bs.pc) * BYTECODE (bs.stack,err) }
  base: to_mc bc.code
{ PC (base + bs'.pc) * BYTECODE (bs'.stack,err) v PC err * true }
```

then machine code terminates with that value

Top-level correctness

Combination:

if the big-step terminates with value

$(exp, env) \Downarrow_{\text{ev}} val \implies$

"the code for exp is installed in bs etc." $\implies$

$\exists bs'.$ "bs' contains val" $\wedge$

```
{ PC (base + bs.pc) * BYTECODE (bs.stack,err) }
  base: to_mc bc.code
{ PC (base + bs'.pc) * BYTECODE (bs'.stack,err) v PC err * true }
```

then machine code terminates with that value

What about diverging (i.e. non-terminating) expressions?

Top-level correctness

Combination:

if the big-step terminates with value

$(exp, env) \Downarrow_{\text{ev}} val \implies$

"the code for exp is installed in bs etc." $\implies$

$\exists bs'.$ "bs' contains val" $\wedge$

```
{ PC (base + bs.pc) * BYTECODE (bs.stack,err) }
  base: to_mc bc.code
{ PC (base + bs'.pc) * BYTECODE (bs'.stack,err) v PC err * true }
```

then machine code terminates with that value

What about diverging (i.e. non-terminating) expressions?

This theorem allows the machine to do anything if exp diverges.

Diverging programs?

Let's define divergence and termination.

For the bytecode:

$$\text{diverges } bs \quad = \quad \forall bs'.\ bs \to^* bs' \implies \exists bs''.\ bs' \to bs''$$

$$\text{terminates } bs\ bs' \quad = \quad bs \to^* bs' \wedge \forall bs''.\ \neg(bs' \to bs'')$$

Diverging programs?

Let's define divergence and termination.

For the bytecode:

another step can always be taken

$$\text{diverges } bs \quad = \quad \forall bs'.\ bs \to^* bs' \implies \exists bs''.\ bs' \to bs''$$

$$\text{terminates } bs\ bs' \quad = \quad bs \to^* bs' \wedge \forall bs''.\ \neg(bs' \to bs'')$$

at some point, no more steps possible

Diverging programs?

Let's define divergence and termination.

For the bytecode:

another step can always be taken

$$\text{diverges } bs \quad = \quad \forall bs'.\ bs \to^* bs' \implies \exists bs''.\ bs' \to bs''$$

$$\text{terminates } bs\ bs' \quad = \quad bs \to^* bs' \wedge \forall bs''.\ \neg(bs' \to bs'')$$

at some point, no more steps possible

Theorem:

$$\forall bs.\ \text{diverges } bs \vee \exists bs'.\ \text{terminates } bs\ bs'$$

a bytecode state either diverges or terminates

Diverging ML programs?

Let's define divergence and termination.

Terminating executions:

$$(exp, env) \Downarrow_{\mathrm{ev}} val$$

big-step sem. convenient for compiler correctness proof

Diverging ML programs?

Let's define divergence and termination.

Terminating executions:

$$(exp, env) \Downarrow_{\mathrm{ev}} val$$

big-step sem. convenient for compiler correctness proof

Specification for diverging executions?

Possible approaches:

- use a small-step semantics
- use big-step semantics but defined co-inductively
- use big-step semantics with a logical clock

Diverging ML programs?

Let's define divergence and termination.

Terminating executions:

$$(exp, env) \Downarrow_{\mathrm{ev}} val$$

big-step sem. convenient for compiler correctness proof

Specification for diverging executions?

Possible approaches:

- use a small-step semantics
- use big-step semantics but defined co-inductively
- use big-step semantics with a logical clock

pragmatic and simple solution

Idea: add logical clock

Common technique: restrict executions using a clock

Idea: add logical clock

Common technique: restrict executions using a clock

Adaption to big-step semantics:

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

Idea: add logical clock

Common technique: restrict executions using a clock

Adaption to big-step semantics:

- add 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

Idea: add logical clock

Common technique: restrict executions using a clock

Adaption to big-step semantics:

- add 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

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

Idea: add logical clock

Common technique: restrict executions using a clock

Adaption to big-step semantics:

- add 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

Big-step semantics with clock

Clock turned off: $(exp, env, \mathsf{None}) \Downarrow_{\mathrm{ev}} res$

Clock turned on:

$$\frac{\begin{array}{l}(f, env, \mathsf{Some}\ (clock + 1)) \Downarrow_{\mathrm{ev}} \mathsf{Result}\ (\mathsf{Closure}\ \ n\ env_1\ exp) \\ (x, env, \mathsf{Some}\ (clock + 1)) \Downarrow_{\mathrm{ev}} \mathsf{Result}\ v \\ (exp, env_1[n \mapsto v], \mathsf{Some}\ clock) \Downarrow_{\mathrm{ev}} res\end{array}}{(\mathsf{App}\ f\ \ x, env, \mathsf{Some}\ (clock + 1)) \Downarrow_{\mathrm{ev}} res}$$

$$\frac{}{(\mathsf{App}\ f\ \ x, env, \mathsf{Some}\ 0) \Downarrow_{\mathrm{ev}} \mathsf{TimeOut}}$$

Big-step semantics with clock

same as semantics without clock

Clock turned off: $(exp, env, \mathsf{None}) \Downarrow_{\mathrm{ev}} res$

Clock turned on:

$$\frac{\begin{array}{l}(f, env, \mathsf{Some}\ (clock+1)) \Downarrow_{\mathrm{ev}} \mathsf{Result}\ (\mathsf{Closure}\ \ n\ env_1\ exp) \\ (x, env, \mathsf{Some}\ (clock+1)) \Downarrow_{\mathrm{ev}} \mathsf{Result}\ v \\ (exp, env_1[n \mapsto v], \mathsf{Some}\ clock) \Downarrow_{\mathrm{ev}} res\end{array}}{(\mathsf{App}\ f\ \ x, env, \mathsf{Some}\ (clock+1)) \Downarrow_{\mathrm{ev}} res}$$

$$\frac{}{(\mathsf{App}\ f\ \ x, env, \mathsf{Some}\ 0) \Downarrow_{\mathrm{ev}} \mathsf{TimeOut}}$$

Big-step semantics with clock

same as semantics without clock

Clock turned off: $(exp, env, \text{None}) \Downarrow_{\text{ev}} res$

Clock turned on:

clock is decremented 'ticked'

$$\frac{\begin{array}{l}(f, env, \text{Some } (clock + 1)) \Downarrow_{\text{ev}} \text{Result } (\text{Closure } \ n\ env_1\ exp) \\ (x, env, \text{Some } (clock + 1)) \Downarrow_{\text{ev}} \text{Result } v \\ (exp, env_1[n \mapsto v], \text{Some } clock) \Downarrow_{\text{ev}} res\end{array}}{(\text{App } f\ x, env, \text{Some } (clock + 1)) \Downarrow_{\text{ev}} res}$$

$$\frac{}{(\text{App } f\ x, env, \text{Some } 0) \Downarrow_{\text{ev}} \text{TimeOut}}$$

Big-step semantics with clock

same as semantics without clock

Clock turned off: $(exp, env, \text{None}) \Downarrow_{\text{ev}} res$

Clock turned on:

clock is decremented 'ticked'

$$\frac{\begin{array}{l}(f, env, \text{Some } (clock + 1)) \Downarrow_{\text{ev}} \text{Result } (\text{Closure } n\ env_1\ exp) \\ (x, env, \text{Some } (clock + 1)) \Downarrow_{\text{ev}} \text{Result } v \\ (exp, env_1[n \mapsto v], \text{Some } clock) \Downarrow_{\text{ev}} res\end{array}}{(\text{App } f\ x, env, \text{Some } (clock + 1)) \Downarrow_{\text{ev}} res}$$

$$\frac{}{(\text{App } f\ x, env, \text{Some } 0) \Downarrow_{\text{ev}} \text{TimeOut}}$$

zero clock returns TimeOut

Big-step semantics with clock

Component TimeOuts passed through:

$$\frac{(f, env, \mathsf{Some}\ (clock + 1)) \Downarrow_{\mathrm{ev}} \mathsf{TimeOut}}{(\mathsf{App}\ f\ \ x, env, \mathsf{Some}\ (clock + 1)) \Downarrow_{\mathrm{ev}} \mathsf{TimeOut}}$$

$$\frac{(x, env, \mathsf{Some}\ (clock + 1)) \Downarrow_{\mathrm{ev}} \mathsf{TimeOut}}{(\mathsf{App}\ f\ \ x, env, \mathsf{Some}\ (clock + 1)) \Downarrow_{\mathrm{ev}} \mathsf{TimeOut}}$$

Big-step semantics with clock

Component TimeOuts passed through:

f times out

$$\frac{(f, env, \text{Some } (clock + 1)) \Downarrow_{\text{ev}} \text{TimeOut}}{(\text{App } f\ x, env, \text{Some } (clock + 1)) \Downarrow_{\text{ev}} \text{TimeOut}}$$

$$\frac{(x, env, \text{Some } (clock + 1)) \Downarrow_{\text{ev}} \text{TimeOut}}{(\text{App } f\ x, env, \text{Some } (clock + 1)) \Downarrow_{\text{ev}} \text{TimeOut}}$$

Big-step semantics with clock

Component TimeOuts passed through:

f times out

$$\frac{(f, env, \text{Some } (clock + 1)) \Downarrow_{\text{ev}} \text{TimeOut}}{(\text{App } f\ x, env, \text{Some } (clock + 1)) \Downarrow_{\text{ev}} \text{TimeOut}}$$

x times out

$$\frac{(x, env, \text{Some } (clock + 1)) \Downarrow_{\text{ev}} \text{TimeOut}}{(\text{App } f\ x, env, \text{Some } (clock + 1)) \Downarrow_{\text{ev}} \text{TimeOut}}$$

Big-step semantics with clock

Component TimeOuts passed through:

f times out

$$\frac{(f, env, \text{Some } (clock + 1)) \Downarrow_{\text{ev}} \text{TimeOut}}{(\text{App } f\ x, env, \text{Some } (clock + 1)) \Downarrow_{\text{ev}} \text{TimeOut}}$$

x times out

$$\frac{(x, env, \text{Some } (clock + 1)) \Downarrow_{\text{ev}} \text{TimeOut}}{(\text{App } f\ x, env, \text{Some } (clock + 1)) \Downarrow_{\text{ev}} \text{TimeOut}}$$

$$\frac{(x, env, \text{Some } clock) \Downarrow_{\text{ev}} \text{TimeOut}}{(\text{Add } x\ y, env, \text{Some } clock) \Downarrow_{\text{ev}} \text{TimeOut}}$$

$$\frac{(y, env, \text{Some } clock) \Downarrow_{\text{ev}} \text{TimeOut}}{(\text{Add } x\ y, env, \text{Some } clock) \Downarrow_{\text{ev}} \text{TimeOut}}$$

Big-step semantics with clock

Component TimeOuts passed through:

f times out

$$\frac{(f, env, \text{Some } (clock + 1)) \Downarrow_{\text{ev}} \text{TimeOut}}{(\text{App } f\ x, env, \text{Some } (clock + 1)) \Downarrow_{\text{ev}} \text{TimeOut}}$$

x times out

$$\frac{(x, env, \text{Some } (clock + 1)) \Downarrow_{\text{ev}} \text{TimeOut}}{(\text{App } f\ x, env, \text{Some } (clock + 1)) \Downarrow_{\text{ev}} \text{TimeOut}}$$

same for Add

$$\frac{(x, env, \text{Some } clock) \Downarrow_{\text{ev}} \text{TimeOut}}{(\text{Add } x\ y, env, \text{Some } clock) \Downarrow_{\text{ev}} \text{TimeOut}}$$

$$\frac{(y, env, \text{Some } clock) \Downarrow_{\text{ev}} \text{TimeOut}}{(\text{Add } x\ y, env, \text{Some } clock) \Downarrow_{\text{ev}} \text{TimeOut}}$$

Big-step semantics with clock

Normal results are handled as usual:

$$\frac{(x, env, cl) \Downarrow_{\mathrm{ev}} \mathsf{Result}\ (\mathsf{Int}\ i) \qquad (y, env, cl) \Downarrow_{\mathrm{ev}} \mathsf{Result}\ (\mathsf{Int}\ j)}{(\mathsf{Add}\ x\ y, env, cl) \Downarrow_{\mathrm{ev}} \mathsf{Result}\ (\mathsf{Int}\ (i + j))}$$

Big-step semantics with clock

Normal results are handled as usual:

$$\frac{(x, env, cl) \Downarrow_{\mathrm{ev}} \mathsf{Result}\ (\mathsf{Int}\ i) \qquad (y, env, cl) \Downarrow_{\mathrm{ev}} \mathsf{Result}\ (\mathsf{Int}\ j)}{(\mathsf{Add}\ x\ y, env, cl) \Downarrow_{\mathrm{ev}} \mathsf{Result}\ (\mathsf{Int}\ (i + j))}$$

Semantics clearly more verbose. However:

Big-step semantics with clock

Normal results are handled as usual:

$$\frac{(x, env, cl) \Downarrow_{\text{ev}} \text{Result (Int } i) \qquad (y, env, cl) \Downarrow_{\text{ev}} \text{Result (Int } j)}{(\text{Add } x\ y, env, cl) \Downarrow_{\text{ev}} \text{Result (Int } (i + j))}$$

Semantics clearly more verbose. However:

With some careful setup, we can prove a theorem:

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

Aside: types

With some careful setup, we can prove a theorem:

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

Aside: types

With some careful setup, we can prove a theorem:

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

Aside: types

What about:

$$\frac{(x, env, cl) \Downarrow_{\text{ev}} \text{Result (Int } i) \qquad (y, env, cl) \Downarrow_{\text{ev}} \text{Result (Closure } \ldots)}{(\text{Add } x\ y, env, cl) \Downarrow_{\text{ev}} \textbf{???}}$$

With some careful setup, we can prove a theorem:

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

Aside: types

What about:

wrong type

$$\frac{(x, env, cl) \Downarrow_{\text{ev}} \text{Result (Int } i) \qquad (y, env, cl) \Downarrow_{\text{ev}} \text{Result (Closure } \ldots)}{(\text{Add } x\ y, env, cl) \Downarrow_{\text{ev}} \textbf{???}}$$

With some careful setup, we can prove a theorem:

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

Aside: types

What about:

wrong type

$$\frac{(x, env, cl) \Downarrow_{\text{ev}} \text{Result (Int } i) \qquad (y, env, cl) \Downarrow_{\text{ev}} \text{Result (Closure } \ldots)}{(\text{Add } x\ y, env, cl) \Downarrow_{\text{ev}} \text{TypeError}}$$

With some careful setup, we can prove a theorem:

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

Aside: types

What about:

wrong type

$$\frac{(x, env, cl) \Downarrow_{\text{ev}} \text{Result (Int } i) \qquad (y, env, cl) \Downarrow_{\text{ev}} \text{Result (Closure } \ldots)}{(\text{Add } x\ y, env, cl) \Downarrow_{\text{ev}} \text{TypeError}}$$

TypeError is passed through just like TimeOut.

With some careful setup, we can prove a theorem:

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

Aside: types

What about:

wrong type

$$\frac{(x, env, cl) \Downarrow_{\text{ev}} \text{Result (Int } i) \qquad (y, env, cl) \Downarrow_{\text{ev}} \text{Result (Closure } \ldots)}{(\text{Add } x\ y, env, cl) \Downarrow_{\text{ev}} \text{TypeError}}$$

TypeError is passed through just like TimeOut.

With some careful setup, we can prove a theorem:

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

res is either Result, TimeOut or TypeError

Aside: types

What about:

wrong type

$$\frac{(x, env, cl) \Downarrow_{\text{ev}} \text{Result (Int } i) \qquad (y, env, cl) \Downarrow_{\text{ev}} \text{Result (Closure } \ldots)}{(\text{Add } x\ y, env, cl) \Downarrow_{\text{ev}} \text{TypeError}}$$

TypeError is passed through just like TimeOut.

With some careful setup, we can prove a theorem:

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

res is either Result, TimeOut or TypeError

TypeError never occurs if *exp* has passed type inference

Non-termination and termination

Evaluation diverges if

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

for all clock values

TimeOut happens

Non-termination and termination

Evaluation diverges if

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

for all clock values

TimeOut happens

Every unclocked evaluation has some clocked equivalent:

$$\forall exp\ env\ val.$$

$$(exp, env, \text{None}) \Downarrow_{\text{ev}} \text{Result } val$$

$$\Longleftrightarrow$$

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

termination: some clock value is sufficient

Reminder

From previous lecture:

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

We want to prove the same for the clocked semantics.

Bytecode with logical clock

Bytecode with logical clock

Making the bytecode clocked:

- we add clock to state
- clock is decremented by Tick instruction
- Tick instruction gets stuck if clock is zero
- clock is either on or off

Note:

- clock is logical device only
- not implemented in machine code

Prove: compilation respects clock

$(exp, env, \mathsf{Some}\ clock) \Downarrow_{\mathrm{ev}} res \implies$

"the code for exp is installed in bs etc." $\wedge$

"the clock in bs has value $clock$" $\implies$

if $res = \mathsf{TimeOut}$ then

$\exists bs'.\ \mathsf{terminates}\ bs\ bs' \wedge$

bs' tries to decrement zero clock

else

$\exists bs'\ val.$

$res = \mathsf{Result}\ val \wedge$

$\mathsf{terminates}\ bs\ bs' \wedge$

bs' not stuck due to clock, contains val

Prove: compilation respects clock

$(exp, env, \mathsf{Some}\ clock) \Downarrow_{\mathrm{ev}} res \implies$

"the code for exp is installed in bs etc." $\wedge$

"the clock in bs has value $clock$" $\implies$

if $res = \mathsf{TimeOut}$ then

$\exists bs'.\ \mathsf{terminates}\ bs\ bs' \wedge$

bs' tries to decrement zero clock

else

$\exists bs'\ val.$

$res = \mathsf{Result}\ val \wedge$

$\mathsf{terminates}\ bs\ bs' \wedge$

bs' not stuck due to clock, contains val

bytecode stays synchronised

Prove: compilation respects clock

$(exp, env, \mathsf{Some}\ clock) \Downarrow_{\mathrm{ev}} res \implies$

"the code for exp is installed in bs etc." $\wedge$

"the clock in bs has value $clock$" $\implies$

if $res = \mathsf{TimeOut}$ then

$\exists bs'.\ \mathsf{terminates}\ bs\ bs' \wedge$

bs' tries to decrement zero clock

else

$\exists bs'\ val.$

$res = \mathsf{Result}\ val \wedge$

$\mathsf{terminates}\ bs\ bs' \wedge$

bs' not stuck due to clock, contains val

bytecode stays synchronised

we assume TypeError impossible

If bytecode does not terminate…

Let's prove:

diverges bs $\wedge$
"the code for exp is installed in bs etc." $\Longrightarrow$
$\forall clock.\ (exp, env, \mathsf{Some}\ clock) \Downarrow_{\mathrm{ev}} \mathsf{TimeOut}$

If bytecode does not terminate…

Let's prove:

clock tured off

diverges bs $\wedge$
"the code for exp is installed in bs etc." $\Longrightarrow$
$\forall clock.\ (exp, env, \mathsf{Some}\ clock) \Downarrow_{\mathrm{ev}} \mathsf{TimeOut}$

If bytecode does not terminate…

Let's prove:

clock tured off

diverges bs $\wedge$
"the code for exp is installed in bs etc." $\Longrightarrow$
$\forall clock.\ (exp, env, \mathsf{Some}\ clock) \Downarrow_{\mathrm{ev}} \mathsf{TimeOut}$

Proof informally:

If bytecode does not terminate…

Let's prove:

clock tured off

diverges bs $\wedge$
"the code for exp is installed in bs etc." $\Longrightarrow$
$\forall clock.\ (exp, env, \text{Some } clock) \Downarrow_{\text{ev}} \text{TimeOut}$

Proof informally:

Proof by contradiction, suppose:

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

If bytecode does not terminate…

Let's prove:

clock tured off

diverges bs $\wedge$
"the code for exp is installed in bs etc." $\Longrightarrow$
$\forall clock.\ (exp, env, \text{Some } clock) \Downarrow_{\text{ev}} \text{TimeOut}$

Proof informally:

Proof by contradiction, suppose:

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

Then for some *clock* and *val*, we must have:

$$(exp, env, \text{Some } clock) \Downarrow_{\text{ev}} \text{Result } val$$

If bytecode does not terminate…

Let's prove:

clock tured off

diverges bs $\wedge$
"the code for exp is installed in bs etc." $\Longrightarrow$
$\forall clock.\ (exp, env, \mathsf{Some}\ clock) \Downarrow_{\mathrm{ev}} \mathsf{TimeOut}$

Proof informally:

Proof by contradiction, suppose:

$\exists clock.\ \neg((exp, env, \mathsf{Some}\ clock) \Downarrow_{\mathrm{ev}} \mathsf{TimeOut})$

Then for some *clock* and *val*, we must have:

$(exp, env, \mathsf{Some}\ clock) \Downarrow_{\mathrm{ev}} \mathsf{Result}\ val$

By compiler correctness theorem: terminates $bs\ bs'$

Contradiction.

If bytecode does not terminate…

Let's prove:

clock tured off

diverges bs $\wedge$
"the code for exp is installed in bs etc." $\Longrightarrow$
$\forall clock.\ (exp, env, \text{Some } clock) \Downarrow_{\text{ev}} \text{TimeOut}$

Proof informally:

Proof by contradiction, suppose:

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

Then for some *clock* and *val*, we must have:

$(exp, env, \text{Some } clock) \Downarrow_{\text{ev}} \text{Result } val$

clock tured on

By compiler correctness theorem: terminates bs bs'

Contradiction.

If bytecode does not terminate…

Let's prove:

clock tured off

diverges bs $\wedge$
"the code for exp is installed in bs etc." $\Longrightarrow$
$\forall clock.\ (exp, env, \text{Some } clock) \Downarrow_{\text{ev}} \text{TimeOut}$

Proof informally:

Proof by contradiction, suppose:

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

Then for some *clock* and *val*, we must have:

$(exp, env, \text{Some } clock) \Downarrow_{\text{ev}} \text{Result } val$

clock tured on

By compiler correctness theorem: terminates bs bs'

must terminate also for clock off

Contradiction.

If bytecode terminates…

Let's prove:

terminates bs bs' $\wedge$
"the code for exp is installed in bs etc." $\wedge$
"bytecode has clock turned off" $\implies$
$\exists val.\ (exp, env, \mathsf{None}) \Downarrow_{\mathrm{ev}} \mathsf{Result}\ val \wedge$ "bs' contains val"

Proof informally:

If bytecode terminates…

Let's prove:

terminates bs bs' $\wedge$
"the code for exp is installed in bs etc." $\wedge$
"bytecode has clock turned off" $\implies$
$\exists val.\ (exp, env, \mathsf{None}) \Downarrow_{\mathrm{ev}} \mathsf{Result}\ val \wedge$ "bs' contains val"

Proof informally:

Since unclocked bytecode terminates, then there is some finite number of steps taken. Let *clock* be step count + 1.

If bytecode terminates…

Let's prove:

terminates bs bs' $\wedge$
"the code for exp is installed in bs etc." $\wedge$
"bytecode has clock turned off" $\Longrightarrow$
$\exists val.\ (exp, env, \mathsf{None}) \Downarrow_{\mathrm{ev}} \mathsf{Result}\ val \wedge$ "bs' contains val"

Proof informally:

Since unclocked bytecode terminates, then there is some finite number of steps taken. Let *clock* be step count + 1.

Clocked bytecode agrees with unlocked when started at *clock*.

If bytecode terminates…

Let's prove:

terminates bs bs' $\wedge$
"the code for exp is installed in bs etc." $\wedge$
"bytecode has clock turned off" $\Longrightarrow$
$\exists val.\ (exp, env, \mathsf{None}) \Downarrow_{\mathrm{ev}} \mathsf{Result}\ val \wedge$ "bs' contains val"

Proof informally:

Since unclocked bytecode terminates, then there is some finite number of steps taken. Let *clock* be step count + 1.

Clocked bytecode agrees with unlocked when started at *clock*.

The clocked big-step sem. always produces some result. Let *result* be the value produced for *exp* and *clock* from above.

If bytecode terminates…

Let's prove:

terminates bs bs' $\wedge$
"the code for exp is installed in bs etc." $\wedge$
"bytecode has clock turned off" $\Longrightarrow$
$\exists val.\ (exp, env, \mathsf{None}) \Downarrow_{\mathrm{ev}} \mathsf{Result}\ val \wedge$ "bs' contains val"

Proof informally:

Since unclocked bytecode terminates, then there is some finite number of steps taken. Let *clock* be step count + 1.

Clocked bytecode agrees with unlocked when started at *clock*.

The clocked big-step sem. always produces some result. Let *result* be the value produced for *exp* and *clock* from above.

Suppose *result* is Result *val*, then by compiler correctness *bs'* must contain *val* and not be stuck. (Bytecode sem. is deterministic.)

If bytecode terminates…

Let's prove:

terminates bs bs' $\wedge$
"the code for exp is installed in bs etc." $\wedge$
"bytecode has clock turned off" $\Longrightarrow$
$\exists val.\ (exp, env, \mathsf{None}) \Downarrow_{\mathrm{ev}} \mathsf{Result}\ val \wedge$ "bs' contains val"

Proof informally:

Since unclocked bytecode terminates, then there is some finite number of steps taken. Let *clock* be step count + 1.

Clocked bytecode agrees with unlocked when started at *clock*.

The clocked big-step sem. always produces some result. Let *result* be the value produced for *exp* and *clock* from above.

Suppose *result* is Result *val*, then by compiler correctness *bs'* must contain *val* and not be stuck. (Bytecode sem. is deterministic.)

If *result* is TimeOut, then by compiler correctness clocked bytecode must have got stuck. Contradiction, i.e. not TimeOut.

Bytecode correct

From previous slides:

If bytecode terminates, then ML semantics ($\Downarrow_{\mathrm{ev}}$) terminates

If bytecode diverges, then ML semantics diverges (TimeOut)

Bytecode correct

From previous slides:

If bytecode terminates, then ML semantics ($\Downarrow_{\mathrm{ev}}$) terminates

If bytecode diverges, then ML semantics diverges (TimeOut)

What about the machine code?

Machine code divergence?

What about the machine code?

Machine code divergence?

What about the machine code?

Correctness of bytecode-to-machine-code compilation:

$$bs \rightarrow^* bs' \implies$$

```
{ PC (base + bs.pc) * BYTECODE (bs.stack,err) }
  base: to_mc bc.code
{ PC (base + bs'.pc) * BYTECODE (bs'.stack,err) ∨ PC err * true }
```

Machine code divergence?

What about the machine code?

Correctness of bytecode-to-machine-code compilation:

$bs \rightarrow^* bs' \implies$

```
{ PC (base + bs.pc) * BYTECODE (bs.stack,err) }
  base: to_mc bc.code
{ PC (base + bs'.pc) * BYTECODE (bs'.stack,err) ∨ PC err * true }
```

Definition of machine-code Hoare triple:

```
{ P } C { Q } = ∀frame. mcTotal (P * C * frame)
                                (Q * C * frame)
```

mcTotal $P\ Q\ =\ \forall s.\ P\ s\implies \exists t.\ s \longrightarrow^* t \wedge Q\ t$

Machine code divergence?

What about the machine code?

Correctness of bytecode-to-machine-code compilation:

bytecode semantics' step relation

$bs \rightarrow^* bs' \implies$

```
{ PC (base + bs.pc) * BYTECODE (bs.stack,err) }
  base: to_mc bc.code
{ PC (base + bs'.pc) * BYTECODE (bs'.stack,err) ∨ PC err * true }
```

Definition of machine-code Hoare triple:

```
{ P } C { Q } = ∀frame. mcTotal (P * C * frame)
                                (Q * C * frame)
```

mcTotal $P\ Q\ =\ \forall s.\ P\ s \implies \exists t.\ s \longrightarrow^* t \wedge Q\ t$

machine-code semantics' step relation

Machine code divergence?

What about the machine code?

Correctness of bytecode-to-machine-code compilation:

bytecode semantics' step relation

$bs \rightarrow^* bs' \implies$

```
{ PC (base + bs.pc) * BYTECODE (bs.stack,err) }
  base: to_mc bc.code
{ PC (base + bs'.pc) * BYTECODE (bs'.stack,err) ∨ PC err * true }
```

unable to specify non-termination

Definition of machine-code Hoare triple:

```
{ P } C { Q } = ∀frame. mcTotal (P * C * frame)
                                (Q * C * frame)
```

mcTotal $P\ Q\ =\ \forall s.\ P\ s \implies \exists t.\ s \longrightarrow^* t \wedge Q\ t$

machine-code semantics' step relation

Machine code divergence?

$$\mathsf{mcTotal}\ P\ Q\ =\ \forall s.\ P\ s\ \Longrightarrow\ \exists t.\ s \longrightarrow^{*} t \wedge Q\ t$$

Machine code divergence?

for every state *s*

execution reaches state *t*

mcTotal $P\ Q\ =\ \forall s.\ P\ s\ \Longrightarrow\ \exists t.\ s \longrightarrow^{*} t \wedge Q\ t$

such that *Q* is true for *t*

Machine code divergence?

for every state *s*

execution reaches state *t*

$$\text{mcTotal}\ P\ Q\ =\ \forall s.\ P\ s\ \Longrightarrow\ \exists t.\ s \longrightarrow^{*} t \wedge Q\ t$$

such that *Q* is true for *t*

Divergence = machine code gets stuck, i.e. never finishes

Machine code divergence?

for every state *s*

execution reaches state *t*

$$\text{mcTotal}\ P\ Q\ =\ \forall s.\ P\ s\ \Longrightarrow\ \exists t.\ s \longrightarrow^{*} t \wedge Q\ t$$

such that *Q* is true for *t*

Divergence = machine code gets stuck, i.e. never finishes

In our case:

Divergence = machine code is forever afterwards executing bytecode

Temporal logic

Divergence = machine code is forever afterwards executing bytecode

Temporal logic

Divergence = machine code is forever afterwards executing bytecode

Traces of machine code states:

$$\begin{aligned} \text{trace } seq \quad = \quad & \forall i.\ \text{if } (\exists s.\ seq(i) \longrightarrow s) \\ & \quad \text{then } seq(i) \longrightarrow seq(i+1) \\ & \quad \text{else } seq(i+1) = \text{error} \end{aligned}$$

Temporal logic

Divergence = machine code is forever afterwards executing bytecode

Traces of machine code states:

$$\text{trace } seq \quad = \quad \forall i.\ \text{if } (\exists s.\ seq(i) \longrightarrow s)$$
$$\text{then } seq(i) \longrightarrow seq(i+1)$$
$$\text{else } seq(i+1) = \text{error}$$

possibly infinite trace

Temporal logic

Divergence = machine code is forever afterwards executing bytecode

Traces of machine code states:

$$\text{trace } seq \quad = \quad \forall i.\ \text{if } (\exists s.\ seq(i) \longrightarrow s)$$
$$\text{then } seq(i) \longrightarrow seq(i+1)$$
$$\text{else } seq(i+1) = \text{error}$$

possibly infinite trace

error is a stuck state

Temporal logic

Divergence = machine code is forever afterwards executing bytecode

Traces of machine code states:

$$\text{trace } seq \quad = \quad \forall i.\ \text{if } (\exists s.\ seq(i) \longrightarrow s)$$
$$\text{then } seq(i) \longrightarrow seq(i+1)$$
$$\text{else } seq(i+1) = \text{error}$$

possibly infinite trace

error is a stuck state

Trace:

$$seq(0) \longrightarrow seq(1) \longrightarrow seq(2) \longrightarrow seq(3) \longrightarrow \ldots \longrightarrow seq(n) \longrightarrow seq(n+1) \longrightarrow \ldots$$

Temporal logic

Divergence = machine code is forever afterwards executing bytecode

Traces of machine code states:

$$\text{trace } seq \quad = \quad \forall i.\ \text{if } (\exists s.\ seq(i) \longrightarrow s)$$
$$\text{then } seq(i) \longrightarrow seq(i+1)$$
$$\text{else } seq(i+1) = \text{error}$$

possibly infinite trace

error is a stuck state

Trace:

$$seq(0) \longrightarrow seq(1) \longrightarrow seq(2) \longrightarrow seq(3) \longrightarrow \ldots \longrightarrow seq(n) \longrightarrow seq(n+1) \longrightarrow \ldots$$

machine-code semantics' step relation

mcTemporal

Standard temporal logic operators:

$$
\begin{aligned}
\text{now } P\ seq &= P\ (seq(0)) \\
\text{next } \psi\ seq &= \psi\ (\lambda i.\ seq(i+1)) \\
\text{always } \psi\ seq &= \forall k.\ \psi\ (\lambda i.\ seq(i+k)) \\
\text{eventually } \psi\ seq &= \exists k.\ \psi\ (\lambda i.\ seq(i+k)) \\
(\psi \Rightarrow \phi)\ seq &= (\psi\ seq \implies \phi\ seq)
\end{aligned}
$$

mcTemporal

Standard temporal logic operators:

holds for initial state

$$
\begin{aligned}
\text{now } P\ seq &= P\ (seq(0)) \\
\text{next } \psi\ seq &= \psi\ (\lambda i.\ seq(i+1)) \\
\text{always } \psi\ seq &= \forall k.\ \psi\ (\lambda i.\ seq(i+k)) \\
\text{eventually } \psi\ seq &= \exists k.\ \psi\ (\lambda i.\ seq(i+k)) \\
(\psi \Rightarrow \phi)\ seq &= (\psi\ seq \implies \phi\ seq)
\end{aligned}
$$

mcTemporal

Standard temporal logic operators:

$$
\begin{array}{rcl}
\text{now } P\ seq & = & P\ (seq(0)) \\
\text{next } \psi\ seq & = & \psi\ (\lambda i.\ seq(i+1)) \\
\text{always } \psi\ seq & = & \forall k.\ \psi\ (\lambda i.\ seq(i+k)) \\
\text{eventually } \psi\ seq & = & \exists k.\ \psi\ (\lambda i.\ seq(i+k)) \\
(\psi \Rightarrow \phi)\ seq & = & (\psi\ seq \implies \phi\ seq)
\end{array}
$$

holds for initial state

delete first state

mcTemporal

Standard temporal logic operators:

$$
\begin{array}{rcl}
\text{now } P\ seq & = & P\ (seq(0)) \\
\text{next } \psi\ seq & = & \psi\ (\lambda i.\ seq(i+1)) \\
\text{always } \psi\ seq & = & \forall k.\ \psi\ (\lambda i.\ seq(i+k)) \\
\text{eventually } \psi\ seq & = & \exists k.\ \psi\ (\lambda i.\ seq(i+k)) \\
(\psi \Rightarrow \phi)\ seq & = & (\psi\ seq \implies \phi\ seq)
\end{array}
$$

holds for initial state

delete first state

for all tails

mcTemporal

Standard temporal logic operators:

$$
\begin{aligned}
\text{now } P\ seq &= P\ (seq(0)) \\
\text{next } \psi\ seq &= \psi\ (\lambda i.\ seq(i+1)) \\
\text{always } \psi\ seq &= \forall k.\ \psi\ (\lambda i.\ seq(i+k)) \\
\text{eventually } \psi\ seq &= \exists k.\ \psi\ (\lambda i.\ seq(i+k)) \\
(\psi \Rightarrow \phi)\ seq &= (\psi\ seq \implies \phi\ seq)
\end{aligned}
$$

holds for initial state

delete first state

for all tails

for some tail

mcTemporal

Standard temporal logic operators:

$$
\begin{aligned}
\text{now } P\ seq &= P\ (seq(0)) \\
\text{next } \psi\ seq &= \psi\ (\lambda i.\ seq(i+1)) \\
\text{always } \psi\ seq &= \forall k.\ \psi\ (\lambda i.\ seq(i+k)) \\
\text{eventually } \psi\ seq &= \exists k.\ \psi\ (\lambda i.\ seq(i+k)) \\
(\psi \Rightarrow \phi)\ seq &= (\psi\ seq \implies \phi\ seq)
\end{aligned}
$$

holds for initial state

delete first state

for all tails

for some tail

implies lifted

mcTemporal

Standard temporal logic operators:

$$\begin{aligned}
\text{now } P\ seq &= P\ (seq(0)) \\
\text{next } \psi\ seq &= \psi\ (\lambda i.\ seq(i+1)) \\
\text{always } \psi\ seq &= \forall k.\ \psi\ (\lambda i.\ seq(i+k)) \\
\text{eventually } \psi\ seq &= \exists k.\ \psi\ (\lambda i.\ seq(i+k)) \\
(\psi \Rightarrow \phi)\ seq &= (\psi\ seq \implies \phi\ seq)
\end{aligned}$$

holds for initial state

delete first state

for all tails

for some tail

implies lifted

Making statements about machine code:

$$\text{mcTemporal } \psi = \forall seq.\ \text{trace } seq \implies \psi\ seq$$

mcTemporal

Standard temporal logic operators:

$$\begin{aligned}
\text{now } P\ seq &= P\ (seq(0)) \\
\text{next } \psi\ seq &= \psi\ (\lambda i.\ seq(i+1)) \\
\text{always } \psi\ seq &= \forall k.\ \psi\ (\lambda i.\ seq(i+k)) \\
\text{eventually } \psi\ seq &= \exists k.\ \psi\ (\lambda i.\ seq(i+k)) \\
(\psi \Rightarrow \phi)\ seq &= (\psi\ seq \implies \phi\ seq)
\end{aligned}$$

holds for initial state

delete first state

for all tails

for some tail

implies lifted

Making statements about machine code:

$$\text{mcTemporal } \psi = \forall seq.\ \text{trace } seq \implies \psi\ seq$$

mcTotal (from before) is an instance:

$$\text{mcTotal } P\ Q \iff \text{mcTemporal } ((\text{now } P) \Rightarrow (\text{eventually } (\text{now } Q)))$$

Bytecode simulation

Correctness of bytecode-to-machine-code step:

$bs \rightarrow bs' \implies$

```
mcTemporal
  ((now (BYTECODE (bs.stack,err) * …))
   ⇒ (next (eventually (now (BYTECODE (bs'.stack,err) * …)))))
    v (eventually (now (PC err * …)))
```

Bytecode simulation

Correctness of bytecode-to-machine-code step:

$bs \rightarrow bs' \implies$

at least one machine-code step

```
mcTemporal
  ((now (BYTECODE (bs.stack,err) * …))
   ⇒ (next (eventually (now (BYTECODE (bs'.stack,err) * …)))))
     v (eventually (now (PC err * …)))
```

Bytecode simulation

Correctness of bytecode-to-machine-code step:

$bs \rightarrow bs' \implies$

at least one machine-code step

```
mcTemporal
  ((now (BYTECODE (bs.stack,err) * …))
   ⇒ (next (eventually (now (BYTECODE (bs'.stack,err) * …)))))
      v (eventually (now (PC err * …)))
```

Preservation of divergence:

diverges $bs \implies$

```
mcTemporal
  ((now (BYTECODE (bs.stack,err) * …))
   ⇒ (always (eventually (now (∃bs'. BYTECODE (bs'.stack,err) * …)))))
      v (eventually (now (PC err * …)))
```

Bytecode simulation

Correctness of bytecode-to-machine-code step:

$bs \rightarrow bs' \implies$

at least one machine-code step

```
mcTemporal
  ((now (BYTECODE (bs.stack,err) * …))
   ⇒ (next (eventually (now (BYTECODE (bs'.stack,err) * …)))))
      v (eventually (now (PC err * …)))
```

Preservation of divergence:

diverges $bs \implies$

forever executing bytecode

```
mcTemporal
  ((now (BYTECODE (bs.stack,err) * …))
   ⇒ (always (eventually (now (∃bs'. BYTECODE (bs'.stack,err) * …)))))
      v (eventually (now (PC err * …)))
```

Complete FP implementation

Read-eval-print loop (REPL)

- sets up initial heap abstraction
- main loop:
  - read: parses input and runs type inference
  - eval: compiles expression to machine code and jumps to generated machine code
- print results and then repeat main loop

Complete FP implementation

Read-eval-print loop (REPL)

- sets up initial heap abstraction
- main loop:
  - read: parses input and runs type inference
  - eval: compiles expression to machine code and jumps to generated machine code
- print results and then repeat main loop

empty input causes loop to stop

Complete FP implementation

Read-eval-print loop (REPL)

- sets up initial heap abstraction
- main loop:
  - read: parses input and runs type inference
  - eval: compiles expression to machine code and jumps to generated machine code
- print results and then repeat main loop

empty input causes loop to stop

generated code is the only code allowed to diverge

Summary

Compiler correctness

- conveniently proved w.r.t. big-step op. sem.
- standard theorem says nothing about diverging programs
- logical clock captures divergence

Divergence in bytecode-to-machine-code compilation

- temporal logic specification and proof

Top-level theorem:

- either: machine code and source semantics terminates
- or: machine code and source semantics diverges

Summary

Compiler correctness

- conveniently proved w.r.t. big-step op. sem.
- standard theorem says nothing about diverging programs
- logical clock captures divergence

Divergence in bytecode-to-machine-code compilation

- temporal logic specification and proof

Top-level theorem:

- either: machine code and source semantics terminates
- or: machine code and source semantics diverges

machine code allowed to terminate with out-of-memory error