Verified compilation of a first-order Lisp language

Lecture 7

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

Magnus Myreen
Michaelmas term, 2013

Interpreter vs compiler

FP interpreter:

- program that implements the small-step semantics
- operates over syntax of the source FP program
- naive implementation wastes time (slow)
- time spent figuring out what operation to perform

Interpreter vs compiler

FP interpreter:

- program that implements the small-step semantics
- operates over syntax of the source FP program
- naive implementation wastes time (slow)
- time spent figuring out what operation to perform

Compilation:

- compiler translates source to native machine code
- evaluation is compile-then-execute-generated-code
- performance can be good
- compilation is dynamic or just-in-time (JIT) if it happens at runtime

Compilation in this lecture

Source: simple first-order FP language

↓

“bytecode”: stack-based little language

↓

machine code: as described in previous lectures

Compilation in this lecture

Ramana's guest lectures will compile higher-order FP

Source: simple first-order FP language

↓

"bytecode": stack-based little language

↓

machine code: as described in previous lectures

Little source language

```
val = SExp

exp ::= Var string
      | Const SExp
      | Add exp exp
      | Car exp
      | Cdr exp
      | Cons exp exp
      | Let string exp exp
      | If exp exp exp
      | ...
```

Little source language

```
val = SExp

exp ::= Var string
      | Const SExp
      | Add exp exp
      | Car exp
      | Cdr exp
      | Cons exp exp
      | Let string exp exp
      | If exp exp exp
      | ...
```

first-order Lisp, i.e. no closure values

Little source language

```
val = SExp
```

first-order Lisp, i.e. no closure values

```
exp ::= Var string
      | Const SExp
      | Add exp exp
      | Car exp
      | Cdr exp
      | Cons exp exp
      | Let string exp exp
      | If exp exp exp
      | ...
```

Semantics: big-step operational semantics similar to Lecture 2.

Bytecode

```
bc_inst ::= LOAD num
          | PUSH SExp
          | ADD
          | CAR
          | CDR
          | CONS
          | POP
          | POP1
          | JUMP num
          | JUMP_IF num
          | ...

bytecode_program = bc_inst list
```

Bytecode

```
bc_inst ::= LOAD num
          | PUSH SExp
          | ADD
          | CAR
          | CDR
          | CONS
          | POP
          | POP1
          | JUMP num
          | JUMP_IF num
          | ...

bytecode_program = bc_inst list
```

Semantics: small-step op. semantics sketched on next slide.

Semantics of bytecode

Operational semantics:

- small-step
- similar to machine code semantics, but more abstract
- stack based
- state tuple: (stack, pc, code)

Semantics of bytecode

Operational semantics:

assumes memory abstraction

- small-step
- similar to machine code semantics, but more abstract
- stack based
- state tuple: (stack, pc, code)

Semantics of bytecode

Operational semantics:

- small-step
- similar to machine code semantics, but more abstract
- stack based
- state tuple: (stack, pc, code)

assumes memory abstraction

instead of regs, memory, flags etc.

Semantics of bytecode

Operational semantics:

- small-step
- similar to machine code semantics, but more abstract
- stack based
- state tuple: (stack, pc, code)

assumes memory abstraction

instead of regs, memory, flags etc.

Examples:

$$\frac{\texttt{fetch pc code = POP}}{\texttt{(x::stack, pc, code)} \xrightarrow{\texttt{eval}} \texttt{(stack, pc + ilength [POP], code)}}$$

Semantics of bytecode

Operational semantics:

- small-step
- similar to machine code semantics, but more abstract
- stack based
- state tuple: (stack, pc, code)

assumes memory abstraction

instead of regs, memory, flags etc.

Examples:

POP instruction to be executed

$$\frac{\texttt{fetch pc code = POP}}{\texttt{(x::stack, pc, code)} \xrightarrow{\texttt{eval}} \texttt{(stack, pc + ilength [POP], code)}}$$

Semantics of bytecode

Operational semantics:

- small-step
- similar to machine code semantics, but more abstract
- stack based
- state tuple: (stack, pc, code)

assumes memory abstraction

instead of regs, memory, flags etc.

Examples:

POP instruction to be executed

$$\frac{\texttt{fetch pc code = POP}}{\texttt{(x::stack, pc, code)} \xrightarrow{\texttt{eval}} \texttt{(stack, pc + ilength [POP], code)}}$$

element at top of stack ...

Semantics of bytecode

Operational semantics:

assumes memory abstraction

- small-step
- similar to machine code semantics, but more abstract
- stack based
- state tuple: (stack, pc, code)

instead of regs, memory, flags etc.

Examples:

POP instruction to be executed

```
fetch pc code = POP
──────────────────────────────────────────────────────────────
(x::stack, pc, code) --eval--> (stack, pc + ilength [POP], code)
```

element at top of stack …

… is removed by POP

Semantics of bytecode

Operational semantics:

- small-step
- similar to machine code semantics, but more abstract
- stack based
- state tuple: (stack, pc, code)

assumes memory abstraction

instead of regs, memory, flags etc.

Examples:

POP instruction to be executed

$$\frac{\texttt{fetch pc code = POP}}{\texttt{(x::stack, pc, code)} \xrightarrow{\texttt{eval}} \texttt{(stack, pc + ilength [POP], code)}}$$

$$\frac{\texttt{fetch pc code = PUSH x}}{\texttt{(stack, pc, code)} \xrightarrow{\texttt{eval}} \texttt{(x::stack, pc + ilength [PUSH x], code)}}$$

More examples

$$\frac{\texttt{fetch pc code = POP1}}{\texttt{(y::x::stack, pc, code)} \xrightarrow{\texttt{eval}} \texttt{(y::stack, pc + ilength [POP1], code)}}$$

More examples

removed element below top

$$\frac{\texttt{fetch pc code = POP1}}{\texttt{(y::x::stack, pc, code)} \xrightarrow{\texttt{eval}} \texttt{(y::stack, pc + ilength [POP1], code)}}$$

More examples

$$\frac{\texttt{fetch pc code = POP1}}{\texttt{(y::x::stack, pc, code)} \xrightarrow{\texttt{eval}} \texttt{(y::stack, pc + ilength [POP1], code)}}$$

removed element below top

$$\frac{\texttt{fetch pc code = CONS}}{\texttt{(y::x::stack, pc, code)} \xrightarrow{\texttt{eval}} \texttt{(Dot x y::stack, pc + ilength [CONS], code)}}$$

introduced Dot pair

More examples

$$\frac{\texttt{fetch pc code = POP1}}{\texttt{(y::x::stack, pc, code)} \xrightarrow{\texttt{eval}} \texttt{(y::stack, pc + ilength [POP1], code)}}$$

removed element below top

$$\frac{\texttt{fetch pc code = CONS}}{\texttt{(y::x::stack, pc, code)} \xrightarrow{\texttt{eval}} \texttt{(Dot x y::stack, pc + ilength [CONS], code)}}$$

introduced Dot pair

$$\frac{\texttt{fetch pc code = JUMP\_IF n} \qquad \texttt{isTrue x}}{\texttt{(x::stack, pc, code)} \xrightarrow{\texttt{eval}} \texttt{(stack, pc + n, code)}}$$

jumped by offset n

More examples

$$\frac{\texttt{fetch pc code = POP1}}{\texttt{(y::x::stack, pc, code)} \xrightarrow{\texttt{eval}} \texttt{(y::stack, pc + ilength [POP1], code)}}$$

removed element below top

$$\frac{\texttt{fetch pc code = CONS}}{\texttt{(y::x::stack, pc, code)} \xrightarrow{\texttt{eval}} \texttt{(Dot x y::stack, pc + ilength [CONS], code)}}$$

introduced Dot pair

$$\frac{\texttt{fetch pc code = JUMP\_IF n} \qquad \texttt{isTrue x}}{\texttt{(x::stack, pc, code)} \xrightarrow{\texttt{eval}} \texttt{(stack, pc + n, code)}}$$

jumped by offset n

$$\frac{\texttt{fetch pc code = JUMP\_IF n} \qquad \neg\texttt{(isTrue x)}}{\texttt{(x::stack, pc, code)} \xrightarrow{\texttt{eval}} \texttt{(stack, pc + ilength [JUMP\_IF n], code)}}$$

didn't jump

More examples

```
fetch pc code = POP1
----------------------------------------------------------------
(y::x::stack, pc, code) --eval--> (y::stack, pc + ilength [POP1], code)
```

removed element below top

```
fetch pc code = CONS
----------------------------------------------------------------
(y::x::stack, pc, code) --eval--> (Dot x y::stack, pc + ilength [CONS], code)
```

introduced Dot pair

```
fetch pc code = JUMP_IF n        isTrue x
----------------------------------------------------------------
(x::stack, pc, code) --eval--> (stack, pc + n, code)
```

jumped by offset n

```
fetch pc code = JUMP_IF n        ¬(isTrue x)
----------------------------------------------------------------
(x::stack, pc, code) --eval--> (stack, pc + ilength [JUMP_IF n], code)
```

didn’t jump

```
fetch pc code = LOAD (length xs)
----------------------------------------------------------------
(xs ++ x::stack, pc, code) --eval--> (x::xs ++ x::stack, pc + ilength
                                        [LOAD (length xs) n], code)
```

copied x and pushed to top

More examples

removed element below top

```
fetch pc code = POP1
────────────────────────────────────────────
(y::x::stack, pc, code) --eval--> (y::stack, pc + ilength [POP1], code)
```

introduced Dot pair

```
fetch pc code = CONS
────────────────────────────────────────────
(y::x::stack, pc, code) --eval--> (Dot x y::stack, pc + ilength [CONS], code)
```

jumped by offset n

```
fetch pc code = JUMP_IF n        isTrue x
────────────────────────────────────────────
(x::stack, pc, code) --eval--> (stack, pc + n, code)
```

didn't jump

```
fetch pc code = JUMP_IF n        ¬(isTrue x)
────────────────────────────────────────────
(x::stack, pc, code) --eval--> (stack, pc + ilength [JUMP_IF n], code)
```

copied x and pushed to top

```
fetch pc code = LOAD (length xs)
────────────────────────────────────────────
(xs ++ x::stack, pc, code) --eval--> (x::xs ++ x::stack, pc + ilength
                                         [LOAD (length xs) n], code)
```

Bytecode → machine code

Define a function (to_mc) from bc_inst to machine code:

```
to_mc (POP::rest) = “ load r0,[r0+4] ” ++ to_mc rest
```

Bytecode → machine code

Define a function (to_mc) from bc_inst to machine code:

```
to_mc (POP::rest) = " load r0,[r0+4] " ++ to_mc rest
```

We prove the correctness of this machine code implementation w.r.t. state assertion `BYTECODE`:

$$\texttt{(stack, pc, code)} \xrightarrow{\texttt{eval}} \texttt{(stack', pc', code')} \Longrightarrow$$

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

Bytecode → machine code

Define a function (to_mc) from bc_inst to machine code:

```
to_mc (POP::rest) = “ load r0,[r0+4] ” ++ to_mc rest
```

We prove the correctness of this machine code implementation w.r.t. state assertion BYTECODE:

if bytecode executes step …

$(\texttt{stack},\ \texttt{pc},\ \texttt{code}) \xrightarrow{\texttt{eval}} (\texttt{stack'},\ \texttt{pc'},\ \texttt{code'}) \Longrightarrow$

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

Bytecode → machine code

Define a function (to_mc) from bc_inst to machine code:

```
to_mc (POP::rest) = “ load r0,[r0+4] ” ++ to_mc rest
```

We prove the correctness of this machine code implementation w.r.t. state assertion BYTECODE:

if bytecode executes step …

$$\texttt{(stack, pc, code)} \xrightarrow{\texttt{eval}} \texttt{(stack', pc', code')} \Longrightarrow$$

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

… then m.c. performs the same w.r.t. the BYTECODE assertion

Bytecode → machine code

Define a function (to_mc) from bc_inst to machine code:

```
to_mc (POP::rest) = “ load r0,[r0+4] ” ++ to_mc rest
```

We prove the correctness of this machine code implementation w.r.t. state assertion BYTECODE:

if bytecode executes step …

```
(stack, pc, code) --eval--> (stack’, pc’, code’) ⟹

{ PC (base + pc) * BYTECODE (stack,err) }
  base: to_mc code
{ PC (base + pc’) * BYTECODE (stack’,err) v PC err * true }
```

… then m.c. performs the same w.r.t. the BYTECODE assertion

… or jumps to err (due to lack of heap space)

BYTECODE assertion

Proper definition: uses a real stack in memory (array). That's fast.

BYTECODE assertion

Proper definition: uses a real stack in memory (array). That's fast.

But to keep it simple, let's just use `HEAP` from previous lecture:

BYTECODE assertion

Proper definition: uses a real stack in memory (array). That's fast.

But to keep it simple, let's just use `HEAP` from previous lecture:

➡ stack can be packaged up in an s-expression:

```
    to_sexp [] = Sym “NIL”
to_sexp (x::xs) = Dot x (to_sexp xs)
```

BYTECODE assertion

Proper definition: uses a real stack in memory (array). That's fast.

But to keep it simple, let's just use `HEAP` from previous lecture:

➡ stack can be packaged up in an s-expression:

```
      to_sexp [] = Sym "NIL"
to_sexp (x::xs) = Dot x (to_sexp xs)
```

➡ we can now define:

```
BYTECODE (stack,err) = HEAP (to_sexp stack,_,_,_,err)
```

BYTECODE assertion

Proper definition: uses a real stack in memory (array). That's fast.

But to keep it simple, let's just use `HEAP` from previous lecture:

- ➡ stack can be packaged up in an s-expression:

```
      to_sexp [] = Sym "NIL"
to_sexp (x::xs) = Dot x (to_sexp xs)
```

- ➡ we can now define:

```
BYTECODE (stack,err) = HEAP (to_sexp stack,_,_,_,err)
```

Exercise: prove POP case correct for to_mc function (prev. slide).

Source → Bytecode

Initial example:

```
(cons '1 '2)   i.e.  Cons (Const (Val 1)) (Const (Val 2))
```

Source → Bytecode

Initial example:

```
(cons '1 '2)   i.e.  Cons (Const (Val 1)) (Const (Val 2))
```

is to compile to

```
[PUSH (Val 1), PUSH (Val 2), CONS]
```

Source → Bytecode

Initial example:

```
(cons '1 '2)   i.e.  Cons (Const (Val 1)) (Const (Val 2))
```

is to compile to

```
[PUSH (Val 1), PUSH (Val 2), CONS]
```

Execution of this pushes

```
Dot (Val 1) (Val 2)
```

onto the stack, leaves rest of stack untouched.

Source → Bytecode

Initial example:

```
(cons '1 '2)   i.e.  Cons (Const (Val 1)) (Const (Val 2))
```

is to compile to

```
[PUSH (Val 1), PUSH (Val 2), CONS]
```

Execution of this pushes

```
Dot (Val 1) (Val 2)
```

onto the stack, leaves rest of stack untouched.

Draft implementation:

```
to_bc (Const x) = [PUSH x]
to_bc (Cons e1 e2) = to_bc e1 ++ to_bc e2 ++ [CONS]
```

Correctness of compilation

$\forall$exp env result code stack pc.

(exp, env) $\Downarrow$ result $\wedge$ to_bc exp = code

$\Longrightarrow$

(stack, 0, code) $\xrightarrow{\texttt{eval}}$ (result :: stack, ilength code, code)

Correctness of compilation

If big-step sem. terminates with result

$$\forall \texttt{exp env result code stack pc.}$$

$$(\texttt{exp, env}) \Downarrow \texttt{result} \;\wedge\; \texttt{to\_bc exp = code}$$

$$\Longrightarrow$$

$$(\texttt{stack, 0, code}) \xrightarrow{\texttt{eval}} (\texttt{result :: stack, ilength code, code})$$

Correctness of compilation

If big-step sem. terminates with result

```
∀exp env result code stack pc.

  (exp, env) ⇓ result  ∧  to_bc exp = code

  ⟹

  (stack, 0, code) --eval--> (result :: stack, ilength code, code)
```

… then execution of generated bytecode pushes result onto stack.

Correctness of compilation

If big-step sem. terminates with result

```
∀exp env result code stack pc.

  (exp, env) ⇓ result  ∧  to_bc exp = code

  ⟹

  (stack, 0, code) --eval--> (result :: stack, ilength code, code)
```

… then execution of generated bytecode pushes result onto stack.

Proof: by induction on ⇓

Correctness of compilation

If big-step sem. terminates with result

```
∀exp env result code stack pc.

  (exp, env) ⇓ result  ∧  to_bc exp = code

  ⟹

  (stack, 0, code) --eval--> (result :: stack, ilength code, code)
```

needs to be generalised for proof by induction

… then execution of generated bytecode pushes result onto stack.

Proof: by induction on ⇓

Correctness of compilation

If big-step sem. terminates with result

```
∀exp env result code stack pc.

  (exp, env) ⇓ result  ∧  to_bc exp = code

  ⟹

  (stack, 0, code) --eval--> (result :: stack, ilength code, code)
```

needs to be generalised for proof by induction

… then execution of generated bytecode pushes result onto stack.

Proof: by induction on ⇓

What about compilation of Var?

Correctness of compilation

If big-step sem. terminates with result

```
∀exp env result code stack pc.

  (exp, env) ⇓ result  ∧  to_bc exp = code

  ⟹

  (stack, 0, code) --eval--> (result :: stack, ilength code, code)
```

needs to be generalised for proof by induction

… then execution of generated bytecode pushes result onto stack.

Proof: by induction on ⇓

What about compilation of Var?

… need to modify compiler and theorem.

Compilation of Var and Let

Implementation:

```
to_bc st (Const x)      = [PUSH x]
to_bc st (Cons e1 e2)   = to_bc st e1 ++
                          to_bc (NONE::st) e2 ++ [CONS]
to_bc st (Var v)        = [LOAD (index_of v st)]
to_bc st (Let v e1 e2) = to_bc st e1 ++
                          to_bc (SOME v::st) e2 ++ [POP1]
```

```
index_of v (x::st) = if x = SOME v then 0
                     else index_of v st + 1
```

Compilation of Var and Let

Implementation:

keeps track of where variables are

```
to_bc st (Const x)      = [PUSH x]
to_bc st (Cons e1 e2)   = to_bc st e1 ++
                          to_bc (NONE::st) e2 ++ [CONS]
to_bc st (Var v)        = [LOAD (index_of v st)]
to_bc st (Let v e1 e2) = to_bc st e1 ++
                          to_bc (SOME v::st) e2 ++ [POP1]
```

```
index_of v (x::st) = if x = SOME v then 0
                     else index_of v st + 1
```

Compilation of Var and Let

Implementation:

keeps track of where variables are

shape of stack

```
to_bc st (Const x)      = [PUSH x]
to_bc st (Cons e1 e2)   = to_bc st e1 ++
                          to_bc (NONE::st) e2 ++ [CONS]
to_bc st (Var v)        = [LOAD (index_of v st)]
to_bc st (Let v e1 e2) = to_bc st e1 ++
                          to_bc (SOME v::st) e2 ++ [POP1]
```

```
index_of v (x::st) = if x = SOME v then 0
                     else index_of v st + 1
```

Compilation of Var and Let

Implementation:

keeps track of where variables are

shape of stack

```
to_bc st (Const x)      = [PUSH x]
to_bc st (Cons e1 e2)   = to_bc st e1 ++
                          to_bc (NONE::st) e2 ++ [CONS]
to_bc st (Var v)        = [LOAD (index_of v st)]
to_bc st (Let v e1 e2)  = to_bc st e1 ++
                          to_bc (SOME v::st) e2 ++ [POP1]
```

Var compiles to LOAD

```
index_of v (x::st) = if x = SOME v then 0
                     else index_of v st + 1
```

Compilation of `Var` and `Let`

Implementation:

keeps track of where variables are

shape of stack

```
to_bc st (Const x)      = [PUSH x]
to_bc st (Cons e1 e2)   = to_bc st e1 ++
                          to_bc (NONE::st) e2 ++ [CONS]
to_bc st (Var v)        = [LOAD (index_of v st)]
to_bc st (Let v e1 e2)  = to_bc st e1 ++
                          to_bc (SOME v::st) e2 ++ [POP1]
```

Var compiles to LOAD

Let expression can refer to bound var

```
index_of v (x::st) = if x = SOME v then 0
                     else index_of v st + 1
```

Correctness (revisited)

$$\forall \texttt{exp env result code stack pc st}.$$

$$(\texttt{exp, env}) \Downarrow \texttt{result} \;\wedge\; \texttt{to\_bc st exp = code} \;\wedge\; \texttt{stack\_inv stack st env}$$

$$\Longrightarrow$$

$$(\texttt{stack, 0, code}) \xrightarrow{\texttt{eval}} (\texttt{result :: stack, ilength code, code})$$

Correctness (revisited)

```
∀exp env result code stack pc st.

  (exp, env) ⇓ result  ∧  to_bc st exp = code  ∧  stack_inv stack st env
  ⟹
  (stack, 0, code) --eval--> (result :: stack, ilength code, code)
```

where stack_inv ensures that stack holds values of env according to st.

```
             stack_inv xs [] env = true
  stack_inv (x::xs) (NONE::st) env = stack_inv xs st env
stack_inv (x::xs) (SOME v::st) env = stack_inv xs (del v st) env ∧ env(v) = x

         del v [] = []
  del v (NONE::st) = NONE :: del v st
del v (SOME w::st) = if v = w then NONE :: del v st else SOME w :: del v st
```

Function calls

$\forall$exp env result code stack pc st fns ctxt.

(exp, fns, env) $\Downarrow$ result $\wedge$ stack_inv stack st env $\wedge$

to_bc (st, ctxt, ilength c1) exp = c2 $\wedge$

“ to_bc-compiled code for all function in ctxt exists in c1++c2 ”

$\Longrightarrow$

(stack, ilength c1, c1++c2) $\xrightarrow{\text{eval}}$ (result :: stack, ilength (c1++c2), c1++c2)

Function calls

function definitions (first-order Lisp)

$\forall$exp env result code stack pc st fns ctxt.

(exp, fns, env) $\Downarrow$ result ∧ stack_inv stack st env ∧
to_bc (st, ctxt, ilength c1) exp = c2 ∧
" to_bc-compiled code for all function in ctxt exists in c1++c2 "
$\Longrightarrow$
(stack, ilength c1, c1++c2) $\xrightarrow{\texttt{eval}}$ (result :: stack, ilength (c1++c2), c1++c2)

Function calls

function definitions (first-order Lisp)

mapping from func. names to compiler info

```
∀exp env result code stack pc st fns ctxt.

  (exp, fns, env) ⇓ result  ∧  stack_inv stack st env  ∧
  to_bc (st, ctxt, ilength c1) exp = c2  ∧
```

" to_bc-compiled code for all function in ctxt exists in c1++c2 "

$\Longrightarrow$

(stack, ilength c1, c1++c2) $\xrightarrow{\text{eval}}$ (result :: stack, ilength (c1++c2), c1++c2)

Function calls

function definitions (first-order Lisp)

mapping from func. names to compiler info

∀exp env result code stack pc st fns ctxt.

  (exp, fns, env) $\Downarrow$ result ∧ stack_inv stack st env ∧
  to_bc (st, ctxt, ilength c1) exp = c2 ∧
  " to_bc-compiled code for all function in ctxt exists in c1++c2 "
  $\Longrightarrow$
  (stack, ilength c1, c1++c2) $\xrightarrow{\texttt{eval}}$ (result :: stack, ilength (c1++c2), c1++c2)

we assume all functions in ctxt have been compiled using to_bc

Function calls

function definitions (first-order Lisp)

mapping from func. names to compiler info

```
∀exp env result code stack pc st fns ctxt.

   (exp, fns, env) ⇓ result  ∧  stack_inv stack st env  ∧
   to_bc (st, ctxt, ilength c1) exp = c2  ∧
   “ to_bc-compiled code for all function in ctxt exists in c1++c2 ”
   ⟹

   (stack, ilength c1, c1++c2) --eval--> (result :: stack, ilength (c1++c2), c1++c2)
```

c2 included to allow recur

we assume all functions in ctxt have been compiled using to_bc

Tail calls

Efficient loops in FP are written as tail-recursion (using tail calls).

Tail calls

Efficient loops in FP are written as tail-recursion (using tail calls).

Two versions of fac:

```
fac n = if n = 0 then 1 else n × fac (n-1)

fac n = f (n,1)
        where f (n,k) = if n = 0 then k else f (n-1,k × n)
```

Tail calls

Efficient loops in FP are written as tail-recursion (using tail calls).

Two versions of fac:

not a tail call, leaves ‘n ×’ to be done after call

```
fac n = if n = 0 then 1 else n × fac (n-1)

fac n = f (n,1)
        where f (n,k) = if n = 0 then k else f (n-1,k × n)
```

Tail calls

Efficient loops in FP are written as tail-recursion (using tail calls).

Two versions of fac:

not a tail call, leaves '`n ×`' to be done after call

```
fac n = if n = 0 then 1 else n × fac (n-1)

fac n = f (n,1)
        where f (n,k) = if n = 0 then k else f (n-1,k × n)
```

tail call leaves no work in this function left to be done

Tail calls

Efficient loops in FP are written as tail-recursion (using tail calls).

Two versions of fac:

not a tail call, leaves ‘n ×’ to be done after call

```
fac n = if n = 0 then 1 else n × fac (n-1)

fac n = f (n,1)
        where f (n,k) = if n = 0 then k else f (n-1,k × n)
```

tail call leaves no work in this function left to be done

FP implementations must ensure that tail calls do not waste space.

Tail calls

Efficient loops in FP are written as tail-recursion (using tail calls).

Two versions of fac:

not a tail call, leaves ‘`n ×`’ to be done after call

```
fac n = if n = 0 then 1 else n × fac (n-1)

fac n = f (n,1)
        where f (n,k) = if n = 0 then k else f (n-1,k × n)
```

tail call leaves no work in this function left to be done

FP implementations must ensure that tail calls do not waste space.

- in our example: `to_bc` must rewind stack before tail-call
- otherwise, the subroutine cannot immediately return to caller

Closures

What if we source sem. has closure values?

Closures

What if we source sem. has closure values?

Short answer: brings new complexity

Closures

What if we source sem. has closure values?

Short answer: brings new complexity

Example: correctness theorem requires relation R

```
∀exp env result … .
  (exp, env) ⇓ result  ∧  …
  ⟹
  ∃x. (stack, …) —eval→ (x :: stack, …)  ∧  R x result
```

Closures

What if we source sem. has closure values?

Short answer: brings new complexity

Example: correctness theorem requires relation R

$$\forall \texttt{exp env result} \ldots .$$
$$(\texttt{exp, env}) \Downarrow \texttt{result} \;\wedge\; \ldots$$
$$\Longrightarrow$$
$$\exists \texttt{x}.\ (\texttt{stack}, \ldots) \xrightarrow{\texttt{eval}} (\texttt{x :: stack}, \ldots) \;\wedge\; \texttt{R x result}$$

non-trivial relation between semantics values (e.g. closures) and bytecode values (e.g. code pointers)

Summary

Compilation

- faster than interpreter-based implementation
- best staged, e.g. via 'bytecode' language that operates on top of memory abstraction (from previous lecture)
- easy to compile to stack-based language
- tail-calls must be efficient

Summary

Compilation

- faster than interpreter-based implementation
- best staged, e.g. via ‘bytecode’ language that operates on top of memory abstraction (from previous lecture)
- easy to compile to stack-based language
- tail-calls must be efficient

Correctness

- correctness statement non-trivial for non-trivial language
- proof by induction on big-step op. sem.
- closure values introduce complexity (topic of next lecture)

Summary

Compilation

- faster than interpreter-based implementation
- best staged, e.g. via 'bytecode' language that operates on top of memory abstraction (from previous lecture)
- easy to compile to stack-based language
- tail-calls must be efficient

Correctness

- correctness statement non-trivial for non-trivial language
- proof by induction on big-step op. sem.
- closure values introduce complexity (topic of next lecture)

guest lectures by Ramana Kumar