

Compiler Construction

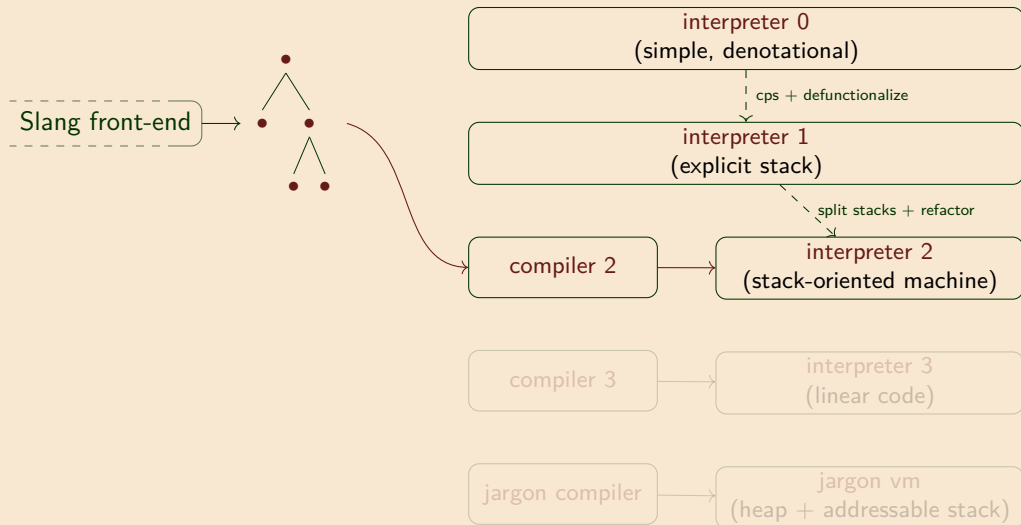
Lecture 9: Deriving interpreter 2

Jeremy Yallop

`jeremy.yallop@cl.cam.ac.uk`

Lent 2026

Reminder: the derivation



Reprise

fib transformation recap

Reprise

```
let rec fib m =  
  if m = 0 then 1  
  else if m = 1 then 1  
  else fib (m-1) + fib (m-2)
```

CPS
convert

```
let rec fib_cps m k =  
  if m = 0 then k 1  
  else if m = 1 then k 1  
  else fib_cps (m-1) (fun a →  
    fib_cps (m-2) (fun b →  
      k (a+b)))
```

defunct
-ionalize

```
let rec apply_tag_list_cont k v = match k, v with  
| [], a → a  
| SUB2 m::k, a → fib_cps_defun_tags (m-2) (ADD a::k)  
| ADD a::k, b → apply_tag_list_cont k (a+b)  
and fib_cps_defun_tags m k =  
  if m = 0 then apply_tag_list_cont k 1  
  else if m = 1 then apply_tag_list_cont k 1  
  else fib_cps_defun_tags (m-1) (SUB2 m::k)
```

list
continuations

```
let rec apply_cont k v = match k, v with  
| ID, a → a  
| K1 (m, k), a → fib_cps_defun (m-2) (K2 (a, k))  
| K2 (a, k), b → apply_cont k (a+b)  
and fib_cps_defun m k =  
  if m = 0 then apply_cont k 1  
  else if m = 1 then apply_cont k 1  
  else fib_cps_defun (m-1) (K1 (m, k))
```

single
recursion

```
let rec eval = function  
| FIB, 0, k → eval (APP, 1, k)  
| FIB, 1, k → eval (APP, 1, k)  
| FIB, m, k → eval (FIB, m-1, SUB2 m::k)  
| APP, a, SUB2 m::k → eval (FIB, m-2, ADD a::k)  
| APP, b, ADD a::k → eval (APP, a+b, k)  
| APP, a, [] → a
```

small
steps

```
let step = function  
| FIB, 0, k → (APP, 1, k)  
| FIB, 1, k → (APP, 1, k)  
| FIB, m, k → (FIB, m-1, SUB2 m::k)  
| APP, a, SUB2 m::k → (FIB, m-2, ADD a::k)  
| APP, b, ADD a::k → (APP, a+b, k)
```

```
let step = function  
| FIB, 0, k → (APP, 1, k)  
| FIB, 1, k → (APP, 1, k)  
| FIB, m, k → (FIB, m-1, SUB2 m::k)  
| APP, a, SUB2 m::k → (FIB, m-2, ADD a::k)  
| APP, b, ADD a::k → (APP, a+b, k)
```

Two stacks

Two stages

Interpreter
0

Interpreter
2

The derivation again, with an evaluator

Reprise

Plan:

1. start with a simple interpreter
2. derive stack machine & “compiler” that translates expressions to instructions

The simple interpreter:

```
type expr =
| INT of int
| ADD of expr * expr
| MUL of expr * expr

(* simple recursive expression evaluator *)
let rec eval = function
| INT a → a
| ADD (e1, e2) → eval e1 + eval e2
| MUL (e1, e2) → eval e1 * eval e2
```

Example: `eval (ADD (MUL (INT 3, INT 4), INT 5))` $\rightsquigarrow$ 17

Reprise

CPS-converting eval produces eval_cps (and eval_2):

```
type cont_2 = int → int
```

```
let rec eval_cps e k = match e with  
| INT a → k a  
| ADD (e1, e2) → eval_cps e1 (fun v1 →  
                    eval_cps e2 (fun v2 →  
                        k (v1 + v2)))  
| MUL (e1, e2) → eval_cps e1 (fun v1 →  
                    eval_cps e2 (fun v2 →  
                        k (v1 * v2)))
```

```
let eval_2 e = eval_cps e (fun x → x)
```

Two stacks

Two stages

Interpreter
0Interpreter
2

Reprise

Defunctionalizing eval_cps produces eval_cps_defun (and eval_3):

```

type cont_3 =
| ID
| OUTER_ADD of expr * cont_3
| OUTER_MUL of expr * cont_3
| INNER_ADD of int * cont_3
| INNER_MUL of int * cont_3

let rec apply f x = match f, x with
| ID, v → v
| OUTER_ADD (e2, k), v1 → eval_cps_defun e2 (INNER_ADD (v1, k))
| OUTER_MUL (e2, k), v1 → eval_cps_defun e2 (INNER_MUL (v1, k))
| INNER_ADD (v1, k), v2 → apply k (v1 + v2)
| INNER_MUL (v1, k), v2 → apply k (v1 * v2)
and eval_cps_defun (e, k) = match e with
| INT a → apply k a
| ADD (e1, e2) → eval_cps_defun e1 (OUTER_ADD (e2, k))
| MUL (e1, e2) → eval_cps_defun e1 (OUTER_MUL (e2, k))

let eval_3 e = eval_cps_defun e ID

```

Two stacks

Two stages

Interpreter
0Interpreter
2

Convert continuations to lists

Reprise

Converting continuations to lists gives eval_cps_defun_tags (& eval_4):

```
type tag = O_ADD of expr
         | I_ADD of int
         | O_MUL of expr
         | I_MUL of int
```

```
type cont_4 = tag list
```

```
let rec apply f x = match f, x with
| [], v → v
| O_ADD e2 :: k, v1 → eval_cps_defun_tags e2 (I_ADD v1 :: k)
| O_MUL e2 :: k, v1 → eval_cps_defun_tags e2 (I_MUL v1 :: k)
| I_ADD v1 :: k, v2 → apply k (v1 + v2)
| I_MUL v1 :: k, v2 → apply k (v1 * v2)
```

```
and eval_cps_defun_tags e k = match e with
| INT a → apply k a
| ADD (e1, e2) → eval_cps_defun_tags e1 (O_ADD e2 :: k)
| MUL (e1, e2) → eval_cps_defun_tags e1 (O_MUL e2 :: k)
```

```
let eval_4 e = eval_cps_defun_tags e []
```

Two stacks

Two stages

Interpreter
0

Interpreter
2

Eliminate mutual recursion & split: step + driver

Reprise

Eliminating mutual recursion & splitting eval into step + driver gives:

```
type acc =          (* "Accumulator" containing either *)
| A_INT of int      (* an int, or *)
| A_EXP of expr     (* an expression *)

let step : cont * acc → cont * acc = function
| k          , A_EXP (INT a)          → (k          , A_INT a)
| k          , A_EXP (ADD (e1, e2))    → (O_ADD e2 :: k, A_EXP e1)
| k          , A_EXP (MUL (e1, e2))    → (O_MUL e2 :: k, A_EXP e1)
| O_ADD e2 :: k, A_INT v1              → (I_ADD v1 :: k, A_EXP e2)
| O_MUL e2 :: k, A_INT v1              → (I_MUL v1 :: k, A_EXP e2)
| I_ADD v1 :: k, A_INT v2              → (k          , A_INT (v1+v2))
| I_MUL v1 :: k, A_INT v2              → (k          , A_INT (v1*v2))
| []         , A_INT v                 → ([]        , A_INT v)

let rec driver : cont * acc → int = function
| [] , A_INT v → v
| state → driver (step state) (* tail recursive *)

let eval_5 e = driver ([], A_EXP e)
```

Two stacks

Two stages

Interpreter
0

Interpreter
2

Two stacks, for values and expressions

An expression stack and a value stack

Reprise

There are really **two independent stacks** here: one for “expressions”
one for values

Two stacks



Two stages

Interpreter

0

Interpreter

2

```
type directive = E of expr (* 'expressions' *)
                  | DO_ADD
                  | DO_MUL
type directive_stack = directive list

type value_stack = int list

(* The state is two stacks *)
type state_6 = directive_stack * value_stack

val step_6 : state_6 → state_6
val driver_6 : state_6 → int

val eval_6 : expr → int
```

Refactor step to use two stacks

Reprise

The refactored implementation of step manipulates the two stacks separately:

Two stacks



Two stages

Interpreter
0

Interpreter
2

```
let step_6 : state_6 → state_6 = function
| E (INT v)      :: ds,      vs → (                                ds,      v :: vs)
| E (ADD (e1,e2)):: ds,      vs → (E e1::E e2::DO_ADD::ds,      vs)
| E (MUL (e1,e2)):: ds,      vs → (E e1::E e2::DO_MUL::ds,      vs)
| DO_ADD         :: ds, v2::v1::vs → (                                ds, v1 + v2::vs)
| DO_MUL         :: ds, v2::v1::vs → (                                ds, v1 * v2::vs)
| _ → failwith "eval : runtime error!"

let rec driver_6 : state_6 → int = function
| ([], [v]) → v
| state → driver_6 (step_6 state)

let eval_6 (e : expr) : int = driver_6 ([E e], [])
```

Reprise

Two stacks



Two stages

Interpreter
0Interpreter
2

	ds	vs
inspect {	[E(ADD(MUL(INT 89,INT 2),MUL(INT 10,INT 4)))]	[]
	[DO_ADD;E(MUL(INT 10, INT 4));E(MUL(INT 89, INT 2))]	[]
compute {	[DO_ADD;E(MUL(INT 10, INT 4));DO_MUL;E(INT 2);E(INT 89)]	[]
	[DO_ADD;E(MUL(INT 10, INT 4));DO_MUL;E(INT 2)]	[89]
	[DO_ADD;E(MUL(INT 10, INT 4));DO_MUL]	[89;2]
inspect {	[DO_ADD;E(MUL(INT 10, INT 4))]	[178]
	[DO_ADD;DO_MUL;E(INT 4);E(INT 10)]	[178]
compute {	[DO_ADD;DO_MUL;E(INT 4)]	[178;10]
	[DO_ADD;DO_MUL]	[178;10;4]
	[DO_ADD]	[178;40]
	[]	[218]

(The top of each stack is on the right)

Interleaving *inspect* + *compute*

Reprise

Two stacks



Two stages

Interpreter
0

Interpreter
2

The evaluator is **interleaving** two quite distinct computations:

1. **inspect**: decomposition of an expression e into sub-expressions $e_1, e_2, \dots$
2. **compute**: the computation of $+$ and $\times$

Idea Since e is known from the start, complete inspect before starting compute

In general: refactor interpreter as translator + lower-level interpreter.

```
interpret_higher (e) = interpret_lower(compile(e))
```

(Examples: interpret Python by compiling to bytecode;
interpret machine code by compiling to micro-code)

Two stages, for compilation and evaluation

Refactoring: *compilation* + evaluation

Reprise

Never put off till run-time what you can do at compile-time.
– David Gries

First stage: **inspect** (before starting **compute**):

```
type instr = (* low-level instructions *)
| lpush of int
| lplus
| lmult
```

```
type code = instr list
```

```
type state_7 = code * value_stack
```

```
let rec compile : expr → code = function
| INT a          → [lpush a]
| ADD (e1, e2) → compile e1 @ compile e2 @ [lplus]
| MUL (e1, e2) → compile e1 @ compile e2 @ [lmult]
```

Two stacks

Two stages



Interpreter
0

Interpreter
2

Refactoring: compilation + *evaluation*

Reprise

Second stage: **compute** (without interleaving **inspect**):

```
let step_7 : state_7 → state_7 = function
| lpush v :: is,      vs → (is,      v      :: vs)
| lplus  :: is, v2::v1::vs → (is, v1 + v2 :: vs)
| lmult   :: is, v2::v1::vs → (is, v1 * v2 :: vs)
| _      → failwith "eval : runtime error!"
```

```
let rec driver_7 = function
| ([], [v]) → v
| _ → driver_7 (step_7 state)
```

```
let eval_7 e = driver_7 (compile e, []) !
```

Two stacks

Two stages



Interpreter
0

Interpreter
2

An eval_7 trace

Reprise

inspect { compile (ADD(MUL(INT 89, INT 2), MUL(INT 10, INT 4)))
~>
[add; mul; push 4; push 10; mul; push 2; push 89]

Two stacks

Two stages



Interpreter
0

compute {
[add; mul; push 4; push 10; mul; push 2; push 89] []
[add; mul; push 4; push 10; mul; push 2] [89]
[add; mul; push 4; push 10; mul] [89; 2]
[add; mul; push 4; push 10] [178]
[add; mul; push 4] [178; 10]
[add; mul] [178; 10; 4]
[add] [178; 40]
[] [218]

instruction
stack

value
stack

Interpreter
2

(The top of each stack is on the right)

Application to interpreter 0

interpret is implicitly using OCaml's runtime stack

Reprise

interp_0.ml

```
let rec interpret (e, env, store) =  
  match e with  
  | Integer n → (INT n, store)  
  | Op(e1, op, e2) →  
    let (v1, store1) = interpret(e1, env, store) in  
    let (v2, store2) = interpret(e2, env, store1) in  
    (do_oper(op, v1, v2), store2)  
  :
```

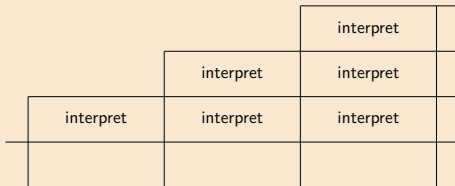
Two stacks

Two stages

Interpreter
0



Interpreter
2



Every call to `interpret` builds an activation record on OCaml's runtime stack.

Interpreter 2 will make this stack explicit

———— execution —————→

Reprise

Use derivation: `eval` $\rightsquigarrow$ `eval_7` as a guide to derive interpreter 0 $\rightsquigarrow$ interpreter 2.

Two stacks

Interpreter 0 is analagous to `eval`, a naive recursive evaluator.

1. Convert to **continuation-passing style**
2. **Defunctionalize**

Two stages

Interpreter 1 is analogous to `eval_6`.

It has one continuation stack for expressions, values and environments

1. **Split the stack** into instruction stack + and a value/environment stack
2. **Stage** as compiler + lower-level interpreter

Interpreter 2 is analogous to `eval_7`

Interpreter
0



Interpreter
2

Properties of Interpreter 2

Reprise

Two stacks

Two stages

Interpreter
0



Interpreter
2

Interpreter 2 is a **high-level stack-oriented machine** with these properties:

- It makes the OCaml runtime stack explicit
- Complex values are pushed onto stacks
- It has one stack for values and environments
- It has one stack for instructions
- The heap is used only for references
- Its instructions have tree-like structure

(we will not look at the details of interpreter 1)

Data types: interpreter 0 vs interpreter 2

Reprise

interp_0.mli

```
type address

type store = address → value

and value =
| REF of address
| INT of int
| BOOL of bool
| UNIT
| PAIR of value * value
| INL of value
| INR of value
| FUN of (value * store
          → value * store)

type env = Ast.var → value
```

interp_2.mli

```
type address = int

type value =
| REF of address
| INT of int
| BOOL of bool
| UNIT
| PAIR of value * value
| INL of value
| INR of value
| CLOSURE of bool * closure
and closure = code * env

and instruction =
| PUSH of value           | LOOKUP of var
| POP                     | BIND of var
| FST                     | SND
| Deref                   | APPLY
| MK_PAIR                 | MK_INL
| MK_CLOSURE of code      | MK_REC of var * code
| ...
```

Two stacks

Two stages

Interpreter
0



Interpreter
2

Interpreter 2

Reprise

Two stacks

Correctness criterion (informal):

If e passes the front-end and `Interp_0.interpret e = v`
then `driver (compile e, []) = <v>` (where `<v>` represents v)

Two stages

We therefore need `compile e` to leave the code for e on top of the stack

Interpreter
0

`val compile : expr → code`

Interpreter
2



From interpreter 0 to interpreter 2: a few cases

Reprise

interp_0.ml

```
let rec interpret (e, env, store) = match e with
| Fst e → (match interpret (e, env, store) with
| PAIR (v1, _), store' → v1, store'
| _ → complain "Runtime error: expecting a pair")
| If(e1, e2, e3) → let v, store' = interpret (e1, env, store) in
  (match v with
  | BOOL true → interpret (e2, env, store')
  | BOOL false → interpret (e3, env, store')
  | v → complain "Runtime error: expecting a boolean!")
```

Two stacks

Two stages

interp_2.ml

```
let rec compile = function
| Fst e → compile e @ [FST]
| If (e1, e2, e3) → compile e1 @ [TEST (compile e2, compile e3)]
:
let step = function
| FST :: ds, V(PAIR (v, _)) :: evs → ds, V v :: evs
| TEST (c1, c2) :: ds, V (BOOL true) :: evs → c1 @ ds, evs
| TEST (c1, c2) :: ds, V (BOOL false) :: evs → c2 @ ds, evs
| POP :: ds, s :: evs → ds, evs
:
```

Interpreter
0

Interpreter
2



From interpreter 0 to interpreter 2: cases for functions

Reprise

— interp_0.ml —

```
let rec interpret (e, env, store) = match e with
| Lambda (x, e) → FUN (fun (v, s) →
                        interpret(e, update (env, (x, v)), s)), store
| App (e1, e2) →
  let v2, store1 = interpret (e2, env, store) in
  let v1, store2 = interpret (e1, env, store1) in
  (match v1 with
   | FUN f → f (v2, store2)
   | v → complain "Runtime error: expecting a function")
```

Two stacks

Two stages

— interp_2.ml —

```
let rec compile = function
| Lambda (x, e) → [MK_CLOSURE(BIND x :: compile e @ [SWAP; POP])]
| App (e1, e2) → compile e2 @ compile e1 @ [APPLY; SWAP; POP]
:

let step = function
| BIND x::ds, V v::evs → ds, EV [(x, v)]::evs
| MK_CLOSURE c::ds, evs → ds, V (mk_fun (c, evs_to_env evs))::evs
| APPLY::ds, V (CLOSURE (_, (c, env)))::V v::evs →
  c @ ds, V v::EV env::evs
:
```

Interpreter
0

Interpreter
2



Example: Compiled code for rev_pair.slang

Reprise

Two stacks

Two stages

Interpreter
0

Interpreter
2

rev_pair.slang

```
let rev_pair (p : int * int)
    : int * int =
    (snd p, fst p)
in
    rev_pair (21, 17)
```

compile

bytecode

```
MK_CLOSURE
  ([BIND p; LOOKUP p;
   SND; LOOKUP p; FST;
   MK_PAIR; SWAP; POP]);
BIND rev_pair;
PUSH 21;
PUSH 17;
MK_PAIR;
LOOKUP rev_pair;
APPLY;
SWAP;
POP;
SWAP;
POP
```

Next time: flattening the code