

# Compiler Construction

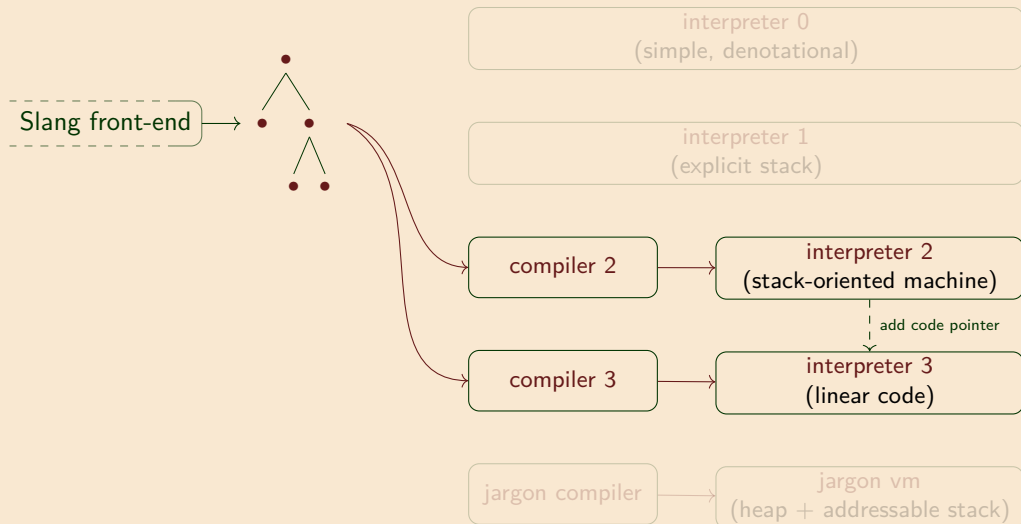
## Lecture 10: Interpreter 3

Jeremy Yallop

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

Lent 2026

# Reminder: the derivation



Non-linear code

# Nested code in interpreter 2: closures

Non-linear  
code



Linear code

Compilation

Execution

rev\_pair.slang

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

compiler 2

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
```

# Nested code in interpreter 2: conditionals

Non-linear  
code



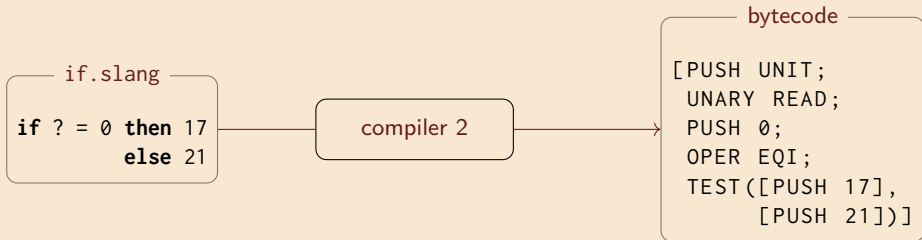
Linear code

Compilation

Execution

```
type instruction = ... | TEST of code * code | WHILE of code * code | ...  
and code = instruction list
```

```
let rec compile = function  
| If(e1, e2, e3) → compile e1 @ [TEST(compile e2, compile e3)]  
| ...
```



(WHILE also takes code arguments)

## Non-linear code



## Linear code

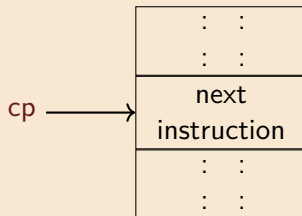
## Compilation

## Execution

Interpreter 2 copies code on the code stack.

We want to introduce a global instruction array indexed by a code pointer (**cp**).

At runtime the **cp** points at the next instruction to be executed.



## New instructions:

**LABEL L** Associate label **L** with this location in the code array

**GOTO L** Set the **cp** to the code address associated with **L**

Linear code

# Compile conditionals, loops

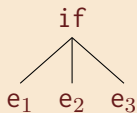
Non-linear  
code

Linear code



Compilation

Execution



compiler 3

*<code for  $e_1$ >*

TEST  $k$

*<code for  $e_2$ >*

GOTO  $m$

$k$ : *<code for  $e_3$ >*

$m$ :

# Compile loops

Non-linear  
code

Linear code



Compilation

Execution

while  
   $\wedge$   
e<sub>1</sub> e<sub>2</sub>

compiler 3

```
m:  <code for e1>  
    TEST k  
    <code for e2>  
    GOTO m  
k:
```

if ? = 0 then 17 else 21 end

Non-linear  
code

Interpreter 2

```
PUSH UNIT
UNARY READ
PUSH 0
OPER EQI
TEST(
    [PUSH 17],
    [PUSH 21]
)
```

Interpreter 3

```
PUSH UNIT
UNARY READ
PUSH 0
OPER EQI
TEST L0
PUSH 17
GOTO L1
LABEL L0
PUSH 21
LABEL L1
HALT
```

**Symbolic**  
code locations

Interpreter 3 (loaded)

```
0 PUSH UNIT
1 UNARY READ
2 PUSH 0
3 OPER EQI
4 TEST L0 = 7
5 PUSH 17
6 GOTO L1 = 9
7 LABEL L0
8 PUSH 21
9 LABEL L1
10 HALT
```

**Numeric**  
code locations

Linear code



Compilation

Execution

## Interpreter 3: compilation

# Data types: interpreter 2 vs interpreter 3

Non-linear  
code

interp\_2.mli

```
type value =  
  | ...  
  | CLOSURE of bool * closure  
and closure = instruction list * env  
and instruction =  
  | PUSH of value      | LOOKUP of var  
  | POP                | BIND of var  
  | FST                | SND  
  | APPLY  
  | MK_PAIR            | MK_INL  
  | MK_CLOSURE of instruction list  
  | ...
```

Linear code

interp\_3.mli

```
type label = string  
type location = label * address option  
  
type value =  
  | ...  
  | CLOSURE of location * env  
  
and instruction =  
  | PUSH of value      | LOOKUP of var  
  | POP                | BIND of var  
  | FST                | SND  
  | APPLY              | RETURN  
  | MK_PAIR            | MK_INL  
  | MK_CLOSURE of location  
  
  | GOTO of location   | LABEL of label
```

Compilation



Execution

Code locations: ("L", None): not yet loaded (assigned numeric address)  
("L", Some i): label "L" has been assigned numeric address i

# Compilation of if: interpreter 2 vs interpreter 3

Non-linear  
code

interp\_2.ml

```
let rec compile = function
| If (e1, e2, e3) → compile e1 @ [TEST(compile e2, compile e3)]
...

```

Linear code

interp\_3.ml

```
let rec comp = function
| If(e1, e2, e3) → let else_label = new_label () in
                   let after_else_label = new_label () in
                   let defs1, c1 = comp e1 in
                   let defs2, c2 = comp e2 in
                   let defs3, c3 = comp e3 in
                   (defs1 @ defs2 @ defs3,
                    (c1
                     @ [TEST(else_label, None)]
                     @ c2
                     @ [GOTO (after_else_label, None);
                        LABEL else_label]
                     @ c3
                     @ [LABEL after_else_label])))

```

Compilation



Execution

# Compilation of lambda: interpreter 2 vs interpreter 3

Non-linear  
code

interp\_2.ml

```
let rec compile = function
| Lambda(x, e) → [MK_CLOSURE(BIND x :: compile e @ [ SWAP; POP; RETURN ])]
...

```

Linear code

interp\_3.ml

```
let rec comp = function
| Lambda(x, e) → let defs, c = comp e in
                  let f = new_label () in
                  let def = [LABEL f; BIND x]
                           @ c @ [SWAP; POP; RETURN]
                  in (def @ defs, [MK_CLOSURE(f, None)])
...
let compile e =
  let defs, c = comp e in
  c      (* body of program *)
@ [HALT] (* stop the interpreter *)
@ defs  (* the function definitions *)

```

Compilation



Execution

(NB: defs are definitions to add after HALT)

# Example: compiled code for rev\_pair.slang

Non-linear  
code

Linear code

Compilation



Execution

rev\_pair.slang

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

compiler 3

bytecode

```
MK_CLOSURE(rev_pair)
BIND rev_pair
PUSH 21
PUSH 17
MK_PAIR
LOOKUP rev_pair
APPLY
SWAP
POP
HALT
LABEL rev_pair:
  BIND p
  LOOKUP p
  SND
  LOOKUP p
  FST
  MK_PAIR
  SWAP
  POP
  RETURN
```

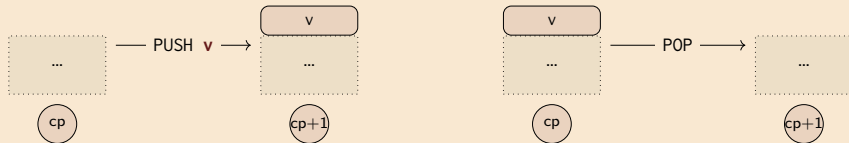
## Interpreter 3: execution

# Interpreter 3: stack manipulation

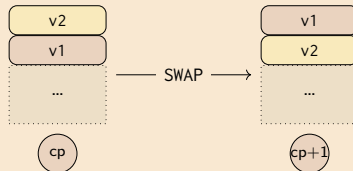
Non-linear  
code

```
let step (cp, evs) = match get_instruction cp, evs with
| PUSH v,          evs → (cp+1, V v :: evs)
| POP,             s :: evs → (cp+1, evs)
| SWAP,            s1 :: s2 :: evs → (cp+1, s2 :: s1 :: evs)
| ...
```

Linear code



Compilation



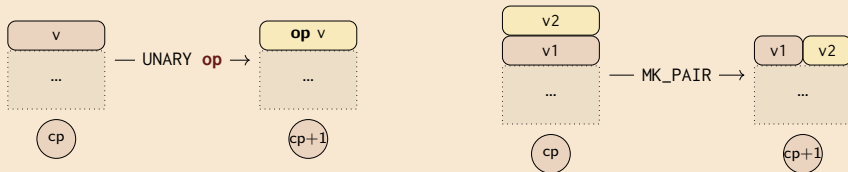
Execution

# Interpreter 3: pairs and primitives

Non-linear  
code

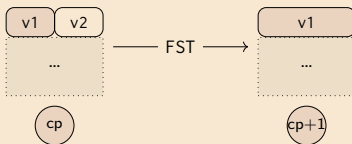
```
let step (cp, evs) = match get_instruction cp, evs with
| UNARY op,          V v::evs → (cp+1, V(do_unary(op, v))::evs)
| MK_PAIR,          V v2::V v1::evs → (cp+1, V(PAIR(v1, v2))::evs)
| FST,              V(PAIR (v, _))::evs → (cp+1, V v::evs)
| ...
```

Linear code



Compilation

Execution



# Interpreter 3: environments

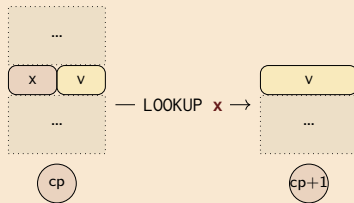
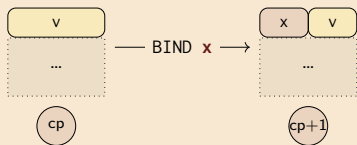
Non-linear  
code

Linear code

Compilation

Execution

```
let step (cp, evs) = match get_instruction cp, evs with  
| BIND x,      V v::evs → (cp+1, EV [(x, v)]::evs)  
| LOOKUP x,    evs      → (cp+1, V(search(evs, x))::evs)  
| ...
```



# Interpreter 3: closures and conditionals

Non-linear  
code

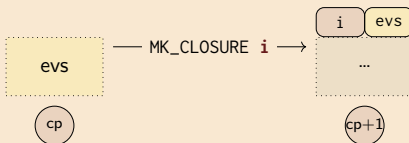
```
let step (cp, evs) = match get_instruction cp, evs with
| TEST (_, Some _), V(BOOL true)::evs → (cp+1, evs)
| TEST (_, Some i), V(BOOL false)::evs → (i, evs)
| MK_CLOSURE i, evs → (cp+1, V(CLOSURE
                           (i, evs_to_env evs))::evs)
| ...
```

Linear code



Compilation

Execution



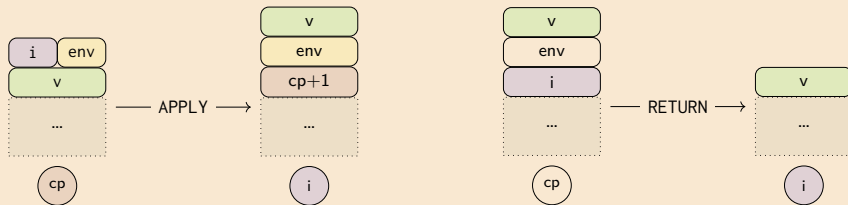
# Interpreter 3: applications and control flow

Non-linear  
code

```
let step (cp, evs) = match get_instruction cp, evs with
| APPLY, V(CLOSURE ((_, Some i), e))
    :: V v :: evs → (i, V v :: EV e :: RA(cp+1) :: evs)
| RETURN, V v :: _ :: RA i :: evs → (i, V v :: evs)
| LABEL i, evs → (cp+1, evs)
| GOTO (_, Some i), evs → (i, evs)
```

Linear code

Compilation



Execution



Non-linear  
code

Linear code

Compilation

Execution

The machine is **becoming simpler** (no OCaml stack; no nested code)

The treatment of **environments** is still very **inefficient**

It still pushes **complex values on the stack**, unlike most virtual machines

It still uses **OCaml's memory management** to manipulate complex values

Next time: **Jargon VM**