Compiler Construction

Lecture 7: translation

Jeremy Yallop
jeremy.yallop@cl.cam.ac.uk
Lent 2025

The Slang language and compiler

Slang
Slang frontend
Interpreter 0
Downwards

Slang = **S**imple **Lang**uage.

Slang **language**: based on L3 (*Semantics of Programming Languages*, IB).

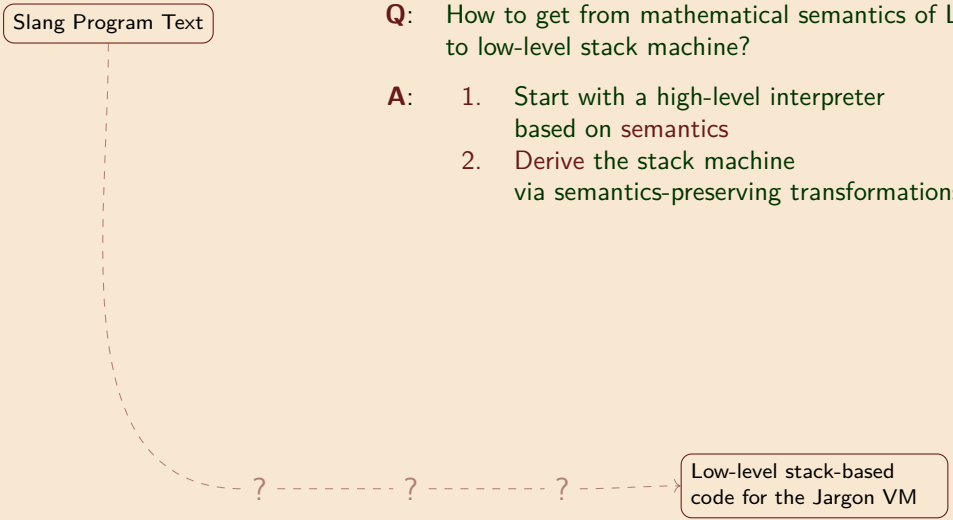
Slang **compiler**: written in OCaml, available from the course web site:

Implementation:	`https://github.com/yallop/cc_cl_cam_ac_uk`
Online explorer:	`https://yallop.github.io/cc_cl_cam_ac_uk`

A good way to learn about compilers is to modify one.

The course website suggests several improvements (some easy, some trickier). Contributed implementations of these suggestions are welcome!

The Gap: Slang to Jargon VM



Q: How to get from mathematical semantics of L3 to low-level stack machine?

A:
1. Start with a high-level interpreter based on semantics
2. Derive the stack machine via semantics-preserving transformations

e ::= () | (e) | n | x | ? |
(simple expressions; ? reads an integer from standard input)
fun (x : t) → e | e e | e bop e | uop e |
(functions, applications and operators)
true | **false** | **if** e **then** e **else** e |
(booleans)
let x : t = e **in** e | **let** f (x :t) :t = e **in** e |
(local definitions)
!e | **ref** e | e := e |
(references and assignments)
begin e; e;...e **end** | **while** e **do** e |
(sequencing and loops)
(e, e) | **fst** e | **snd** e |
(pairs)
inl t e | **inr** t e | **case** e **of** **inl** (x :t) → e | **inr** (x:t) → e
(sums; note type annotations)

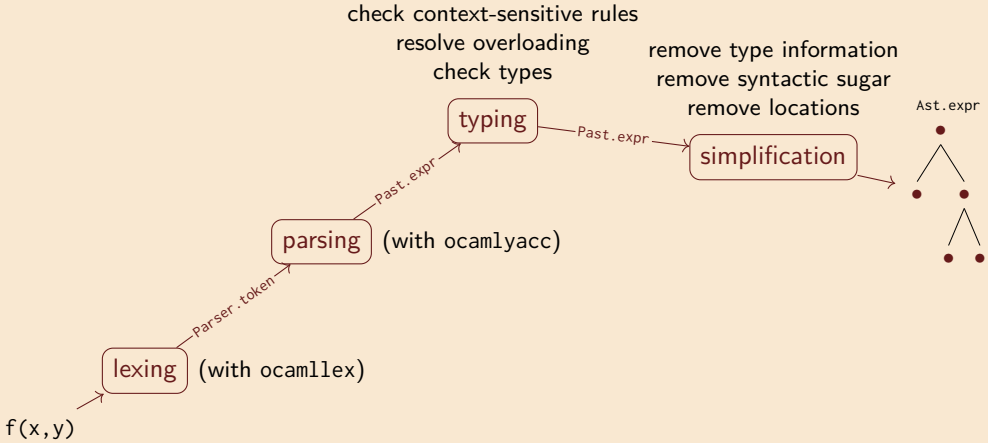
From slang/examples/fib.slang:

```
let fib (m : int) : int =
   if m = 0 then 1
   else if m = 1 then 1
   else fib (m - 1) + fib (m -2)
in
   fib(?)
```

From slang/examples/gcd.slang:

```
let gcd (p : int * int) : int =
  let m : int = fst p in
  let n : int = snd p in
    if m = n then m
    else if m < n then gcd (m, n - m)
    else gcd(m - n,  n)
in gcd (?, ?)
```

Slang front end


check context-sensitive rules
resolve overloading
check types
remove type information
remove syntactic sugar
remove locations
Ast.expr
typing
Past.expr
simplification
Past.expr
parsing
(with ocamlyacc)
Parser.token
lexing
(with ocamllex)
f(x,y)

The Slang parsed AST

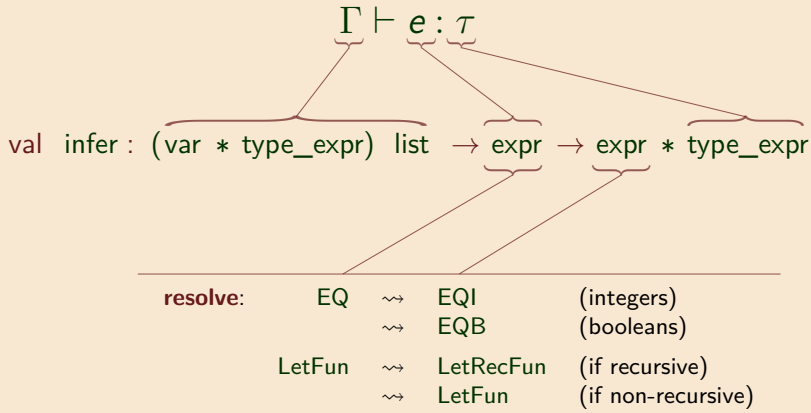
In slang/past.ml:

Locations (loc) are reported in error messages

```
type var = string
type loc = Lexing.position

type expr =
| Var of loc * var
| Integer of loc * int
| Op of loc * expr * oper * expr
| If of loc * expr * expr * expr
| Pair of loc * expr * expr
| Case of loc * expr * lambda * lambda
| Lambda of loc * lambda
| App of loc * expr * expr
| Let of loc * var * type_expr * expr * expr
| LetFun of loc * var * lambda * type_expr * expr
| LetRecFun of loc * var * lambda * type_expr * expr
| ... (* many cases omitted *)
and lambda = var * type_expr * expr
```

In slang/static.mli:



Infer types, apply (context-sensitive) rules that cannot be resolved in context-free grammars

Expression simplification

In slang/past_to_ast.ml:

$$\texttt{val translate\_expr : Past.expr} \to \texttt{Ast.expr}$$

translate_expr **simplifies expressions** to remove "syntactic sugar":

$$\texttt{let x : t = e1 in e2} \quad \rightsquigarrow \quad \texttt{(fun (x : t)} \to \texttt{e2) e1}$$

The output type (Ast.expr) does not contain Let nodes.

The Slang internal AST

In slang/ast.ml:

```
type var = string
type oper = ADD | ... | EQB | EQI
type expr =
| Var of var
| Integer of int
| Op of expr * oper * expr
| If of expr * expr * expr
| Pair of expr * expr
| Case of expr * lambda * lambda
| Lambda of lambda
| App of expr * expr
| LetFun of var * lambda * expr
| LetRecFun of var * lambda * expr
| ... (* many cases omitted *)
and lambda = var * expr
```

Differences from slang/past.ml

No locations
(error reporting is finished)

No types
(not used for compilation)

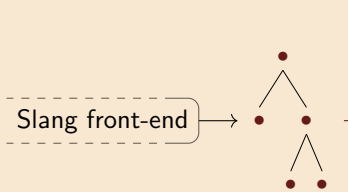
No `EQ`
(resolved to `EQI` or `EQB`)

No `Let`
(removed during simplification)

Some compilers (e.g. OCaml) drop types here; others (e.g. GHC) use them in the middle end

Interpreter 0

Lectures 7–11: the derivation



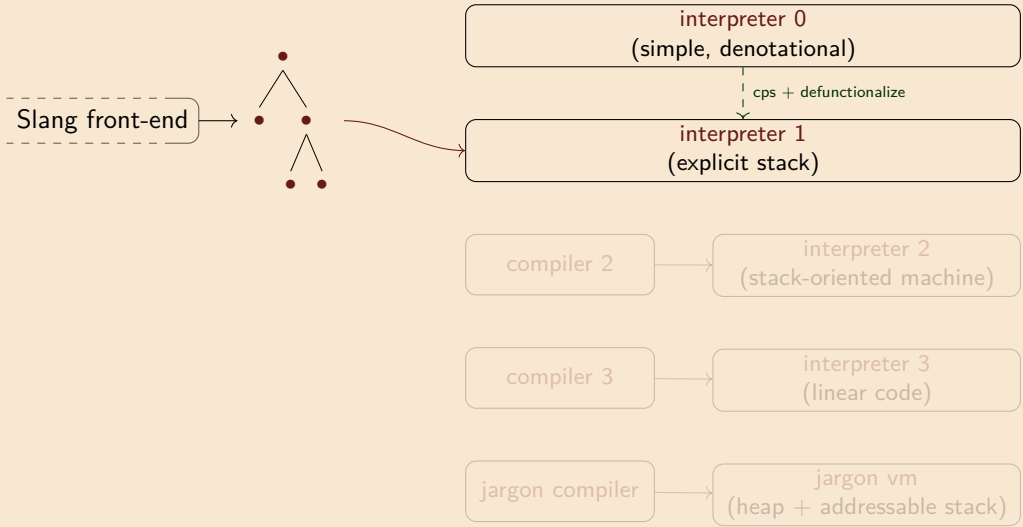
interpreter 0
(simple, denotational)

interpreter 1
(explicit stack)

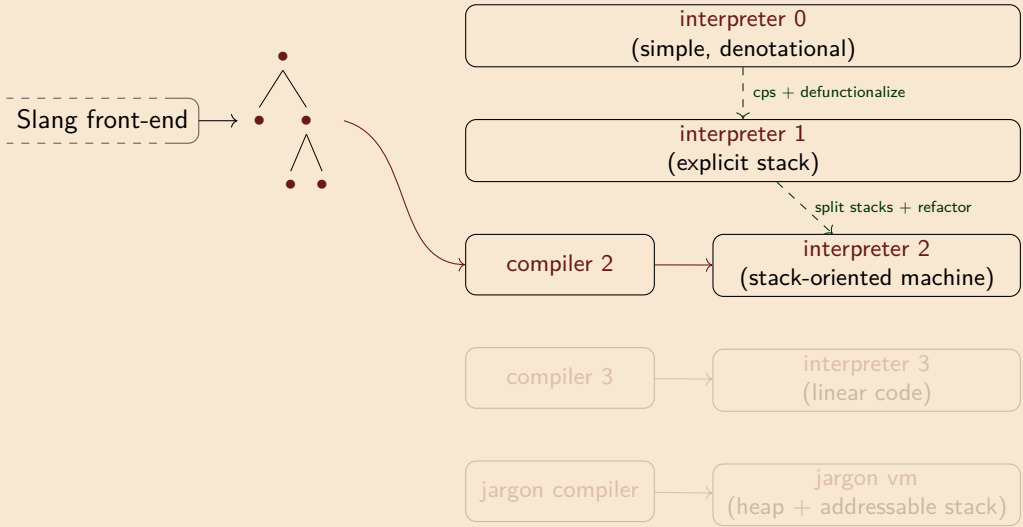
compiler 2 → interpreter 2
(stack-oriented machine)

compiler 3 → interpreter 3
(linear code)

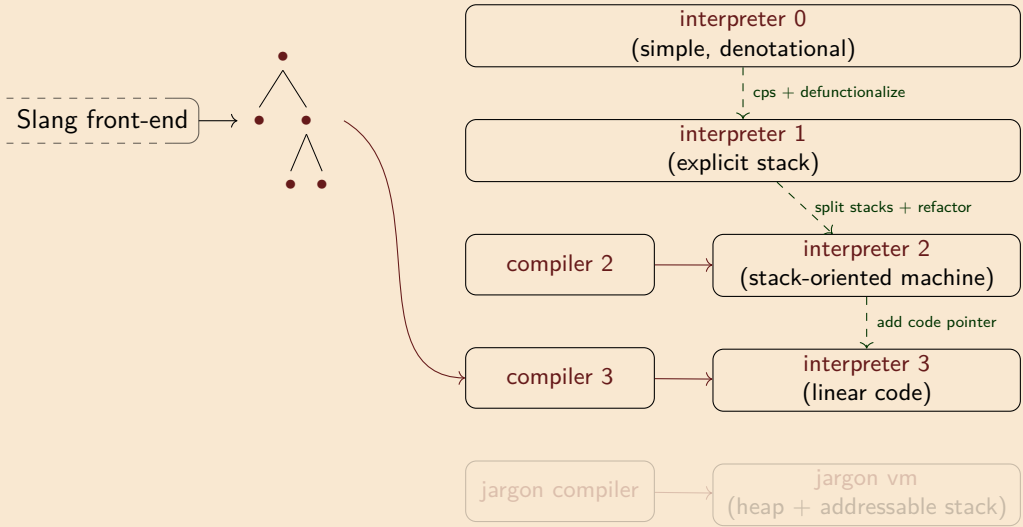
jargon compiler → jargon vm
(heap + addressable stack)

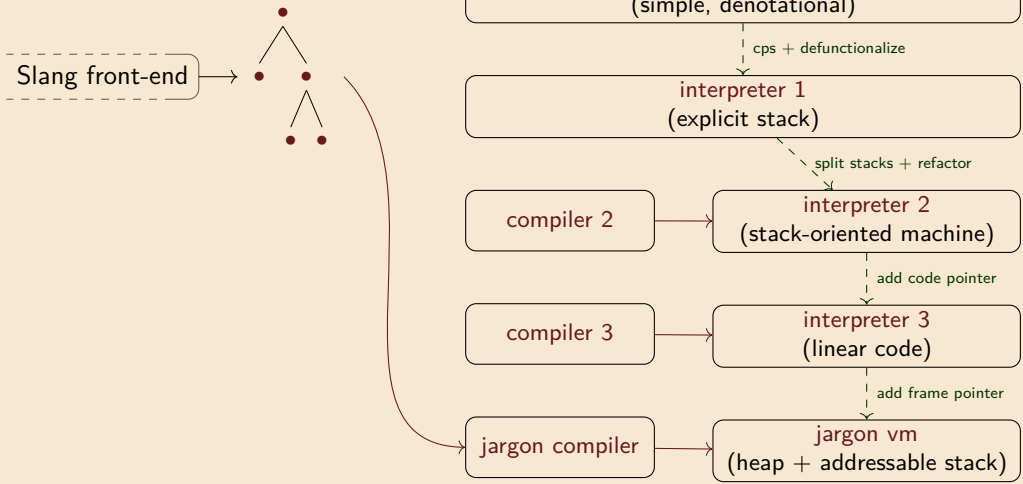

Slang front-end
interpreter 0
(simple, denotational)
cps + defunctionalize
interpreter 1
(explicit stack)
compiler 2
interpreter 2
(stack-oriented machine)
compiler 3
interpreter 3
(linear code)
jargon compiler
jargon vm
(heap + addressable stack)

Lectures 7–11: the derivation



Lectures 7–11: the derivation




Slang front-end
interpreter 0
(simple, denotational)
cps + defunctionalize
interpreter 1
(explicit stack)
split stacks + refactor
compiler 2
interpreter 2
(stack-oriented machine)
add code pointer
compiler 3
interpreter 3
(linear code)
add frame pointer
jargon compiler
jargon vm
(heap + addressable stack)

Approaches to Mathematical Semantics

Operational

Meaning defined via transition relations on abstract machine states

$$\langle e, \sigma \rangle \longrightarrow \langle e', \sigma' \rangle$$

Semantics (Part 1B)

Axiomatic

Meaning defined via logical specifications of behaviour

$$\{P\}\, C\, \{Q\}$$

Hoare Logic (Part II)
Separation Logic

Denotational

Meaning defined via mathematical objects such as functions.

$$[\![e]\!]\eta = v$$

Denotational Semantics (Part II)

A rough denotational semantics for L3

$\mathbb{N}$ = integers $\quad \mathbb{B}$ = booleans $\quad \mathbf{A}$ = addresses $\quad \mathbf{I}$ = identifiers

$\mathbf{E}$ = environments = $\mathbf{I} \to \mathbf{V}$ $\quad$ $\mathbf{S}$ = stores = $\mathbf{A} \to \mathbf{V}$

$$
\begin{aligned}
\mathbf{V} &= \text{set of values} \\
&\approx \mathbf{A} \\
&\quad + \mathbb{N} \\
&\quad + \mathbb{B} \\
&\quad + \{()\} \\
&\quad + \mathbf{V} \times \mathbf{V} \\
&\quad + (\mathbf{V} + \mathbf{V}) \\
&\quad + (\mathbf{V} \times \mathbf{S}) \to (\mathbf{V} \times \mathbf{S})
\end{aligned}
$$

Set of values $\mathbf{V}$ solves this "domain equation" (here $+$ means disjoint union)

Solving such equations is not trivial

$\mathbf{M}$ = the meaning function $\quad$ Expr = L3 expressions

$$\mathbf{M} : (\text{Expr} \times \mathbf{E} \times \mathbf{S}) \to (\mathbf{V} \times \mathbf{S})$$

(**Aside**: What is the meaning of a non-terminating expression?)

Interpreter 0: An OCaml approximation

$\mathbf{A}$ = set of addresses

$\mathbf{S}$ = set of stores = $\mathbf{A} \to \mathbf{V}$

$$
\begin{array}{rcl}
\mathbf{V} & = & \text{set of values} \\
& \approx & \mathbf{A} \\
& & + \mathbb{N} \\
& & + \mathbb{B} \\
& & + \{()\} \\
& & + \mathbf{V} \times \mathbf{V} \\
& & + (\mathbf{V} + \mathbf{V}) \\
& & + (\mathbf{V} \times \mathbf{S}) \to (\mathbf{V} \times \mathbf{S})
\end{array}
$$

$\mathbf{E}$ = set of environments = $\mathbf{I} \to \mathbf{V}$

$\mathbf{M}$ = the meaning function

$\mathbf{M} : (\text{Expr} \times \mathbf{E} \times \mathbf{S}) \to (\mathbf{V} \times \mathbf{S})$

From slang/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

val interpret :
   Ast.expr * env * store → value * store
```

interpret: many cases are straightforward

From slang/interp_0.ml:

```
let rec interpret (e, env, store) =
  match e with
  | 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!")
  | Pair(e1, e2) →
     let (v1, store1) = interpret (e1, env, store) in
     let (v2, store2) = interpret (e2, env, store1) in
        (PAIR(v1, v2), store2)
  | Fst e →
     (match interpret (e, env, store) with
      | (PAIR (v1, _), store') → (v1, store')
      | (v, _) → complain "Runtime error: expecting a pair!")
  | Inl e → let (v, store') = interpret (e, env, store) in
                (INL v, store')
  ...
```

From slang/interp_0.ml:

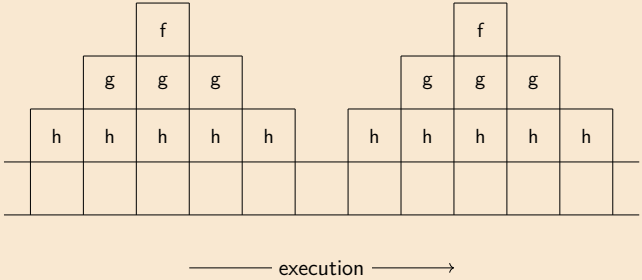
```
let rec interpret (e, env, store) =
match e with
  :
| Lambda(x, e) → FUN (fun (v, s) →
                        interpret(e, update(env, (x, v)), s)), store
| App(e1, e2) →
   let (v2, store1) = interpret(e2, env, store) in
   let (v1, store2) = interpret(e1, env, store1) in
   (match v1 with
    | FUN f → f (v2, store2)
    | v → complain "Runtime error: function expected!")
| LetRecFun(f, (x, body), e) →
   let rec new_env g = (* a recursive environment! *)
     if g = f then FUN (fun (v, s) →
                          interpret(body, update(new_env, (x, v)), s))
     else env g
   in interpret(e, new_env, store)
```

Downwards

Interpreter 0 uses OCaml's stack. How can we move toward the Jargon VM?

```
let f x = x + 1 in
let g y = f (y+2) + 2 in
let h w = g (w+1) + 3 in
  h (h 17)
```

At run-time the call stack contains an activation record for each invocation



Recall tail recursion: sum *vs* sum_tr

Slang

Slang frontend

Interpreter 0

Downwards

```
let rec sum = function
  | [] → 0
  | x :: xs → x + sum xs
```

```
let rec sum_tr acc = function
  | [] → acc
  | x :: xs → sum_tr (x + acc) xs
let sum' xs = sum_tr 0 xs
```

```
    sum [1;2;3]
⇝   1 + sum [2;3]
    '_____'
      stack
⇝   1 + (2 + sum [3])
    '_____________'
          stack
⇝   1 + (2 + (3 + sum []))
    '_________________'
            stack
⇝   1 + (2 + (3 + 0))
⇝   1 + (2 + 3)
⇝   1 + 5
⇝   6
```

```
    sum' [1;2;3]
⇝   sum_tr 0 [1;2;3]
⇝   sum_tr (1+0) [2;3]
⇝   sum_tr 1 [2;3]
⇝   sum_tr (2+1) [3]
⇝   sum_tr 3 [3]
⇝   sum_tr (3+3) []
⇝   sum_tr 6 []
⇝   6
```

Convert tail-recursion to iteration

Tail-recursive sum

```
let rec sum_tr a = function
  | [] → a
  | x::xs → sum_tr (x+a) xs
let sum' xs = sum_tr 0 xs
```

Iterative sum

```
let sum_iter a l =
  let ra = ref a in
  let rl = ref l in
  let result = ref 0 in
  let not_done = ref true in
  let _ = while !not_done
  do
    match !rl with
    | [] → result := !ra;
           not_done := false
    | x::xs → ra := x + !ra;
              rl := xs;
  done;
  in !result
let sum l = sum_iter 0 l
```

one ref per argument
one ref for the result
one ref for the loop

use a loop
in place of recursion

use refs in body
in place of variables
(e.g. !rl for l)

We illustrate tail-recursion elimination as a source-to-source transformation.

In practice, compilers compile low-level representations of tail-recursive code to loops.

We will consider all tail-recursive functions as representing **iterative** programs

Transforming recursion to tail-recursion

Can transform *all* recursive functions into first-order tail-recursive functions.

Two steps:

1. CPS transformation

Add an extra argument to each function

```
let f x = ...          ⇝   let f x k = ...
let z = f v in e       ⇝   f v (fun z → e)
```

These *continuation* arguments represent "the rest of the computation"

2. Defunctionalization

Turn function arguments into data

```
(fun x → e)    ⇝   Cont_i(v_1, ..., v_k)
f v            ⇝   apply f v
```

The *defunctionalized continuations* form a stack

Result: tail-recursive functions that carry their own stacks as extra arguments

Next step: CPS-transform & defunctionalize interpreter 0

Next time: CPS & defunctionalization