

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 **L**anguage.

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

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

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

Slang

Slang Program Text



Slang
frontend

Interpreter
0

Downwards

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

Low-level stack-based
code for the Jargon VM

Slang Syntax (informal)

Slang

$e ::= () \mid (e) \mid n \mid x \mid ? \mid$

(simple expressions; ? reads an integer from standard input)

$\text{fun } (x : t) \rightarrow e \mid e e \mid e \text{ bop } e \mid \text{uop } e \mid$

(functions, applications and operators)

$\text{true} \mid \text{false} \mid \text{if } e \text{ then } e \text{ else } e \mid$

(booleans)

$\text{let } x : t = e \text{ in } e \mid \text{let } f (x : t) : t = e \text{ in } e \mid$

(local definitions)

$!e \mid \text{ref } e \mid e := e \mid$

(references and assignments)

$\text{begin } e; e; \dots e \text{ end} \mid \text{while } e \text{ do } e \mid$

(sequencing and loops)

$(e, e) \mid \text{snd } e \mid \text{fst } e \mid$

(pairs)

$\text{inl } t e \mid \text{inr } t e \mid \text{case } e \text{ of inl } (x : t) \rightarrow e \mid \text{inr } (x : t) \rightarrow e$

(sums; note type annotations)

Slang frontend

Interpreter 0

Downwards

Slang



Slang
frontend

Interpreter
0

Downwards

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

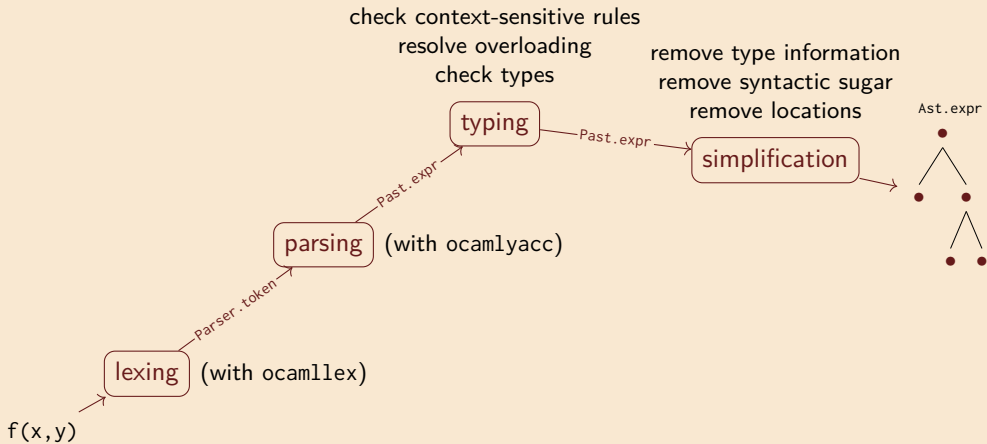
Slang

Slang
frontend



Interpreter
0

Downwards



Slang

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
```

Slang
frontend



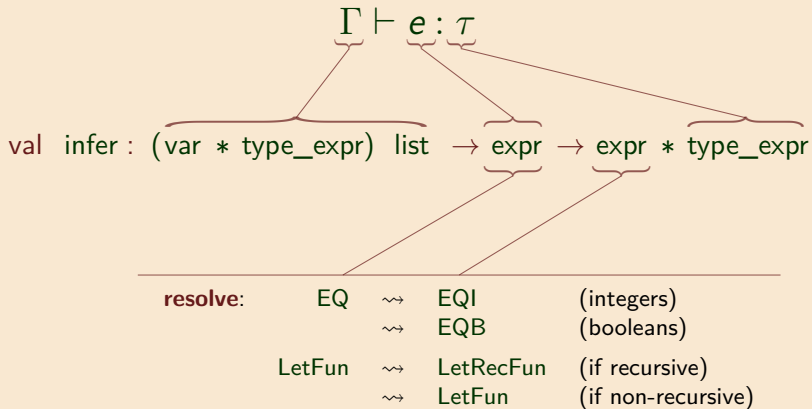
Interpreter
0

Downwards

Type checking & resolution

Slang

In `slang/static.mli`:



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

Slang
frontend



Interpreter
0

Downwards

Slang

In `slang/past_to_ast.ml`:

```
val translate_expr : Past.expr → Ast.expr
```

Slang
frontend



Interpreter
0

`translate_expr` **simplifies expressions** to remove “syntactic sugar”:

$$\text{let } x : t = e1 \text{ in } e2 \rightsquigarrow (\text{fun } (x : t) \rightarrow e2) e1$$

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

Downwards

The Slang internal AST

Slang

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)

Slang
frontend

Interpreter
0

Downwards

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

Slang

Slang
frontend

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)

Interpreter
0

Downwards

Approaches to Mathematical Semantics

Slang

Slang
frontend

Interpreter
0



Downwards

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.

$$\llbracket e \rrbracket \eta = v$$

Denotational Semantics
(Part II)

A rough denotational semantics for L3

Slang

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

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

$\mathbf{V}$ = set of values

$\approx \mathbf{A}$

+ $\mathbb{N}$

+ $\mathbb{B}$

+ $\{()\}$

+ $\mathbf{V} \times \mathbf{V}$

+ $(\mathbf{V} + \mathbf{V})$

+ $(\mathbf{V} \times \mathbf{S}) \rightarrow (\mathbf{V} \times \mathbf{S})$

Set of values $\mathbf{V}$ solves this
"domain equation"

(here + means disjoint union)

Solving such equations is not trivial

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

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

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

Slang
frontend

Interpreter
0

Downwards

Interpreter 0: An OCaml approximation

Slang

$\mathbf{A}$ = set of addresses

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

$\mathbf{V}$ = set of values

$\approx \mathbf{A}$
+ $\mathbb{N}$
+ $\mathbb{B}$
+ $\{()\}$
+ $\mathbf{V} \times \mathbf{V}$
+ $(\mathbf{V} + \mathbf{V})$
+ $(\mathbf{V} \times \mathbf{S}) \rightarrow (\mathbf{V} \times \mathbf{S})$

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

$\mathbf{M}$ = the meaning function

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

Slang
frontend

Interpreter
0

Downwards

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

Slang

From `slang/interp_0.ml`:

Slang
frontend

Interpreter
0

Downwards

```
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')
  ...
```

Slang functions $\mapsto$ OCaml functions

Slang

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)
```

Slang
frontend

Interpreter
0

Downwards

Downwards

From Interpreter 0 to the Jargon VM

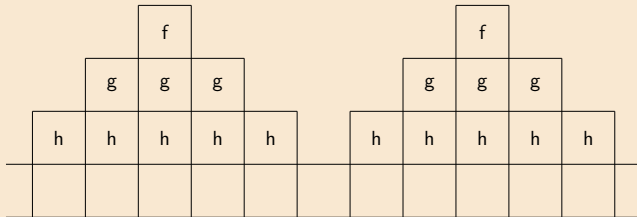
Slang

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

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

Slang frontend

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



———— execution —————→

Interpreter

Downwards



Recall tail recursion: sum vs sum_tr

Slang

```
let rec sum l =  
  match l with  
  | [] → 0  
  | x :: xs → x + sum xs
```

Slang
frontend

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

Interpreter
0

```
sum [1;2;3]  
↪ 1 + sum [2;3]  
↪ 1 + (2 + sum [3])  
↪ 1 + (2 + (3 + sum []))  
↪ 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
```

Downwards



Convert tail-recursion to iteration

Slang

Tail-recursive sum

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

use a loop
in place of recursion

Iterative sum

```
let sum_iter l a =  
  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 l 0
```

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

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

Slang
frontend

Interpreter
0

Downwards



Transforming recursion to tail-recursion

Slang

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

Two steps:

Slang
frontend

1. CPS transformation

Add an extra argument to each function

`let f x = ...` $\rightsquigarrow$ `let f x k = ...`

`let z = f v in e` $\rightsquigarrow$ `f v (fun z → e)`

These *continuation* arguments represent
“the rest of the computation”

2. Defunctionalization

Turn function arguments into data

`(fun x → e)` $\rightsquigarrow$ `Conti(v1, ..., vk)`

`f v` $\rightsquigarrow$ `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

Interpreter
0

Downwards



Next time: CPS & defunctionalization