Compilation and Program Analysis (#11) : Functional languages

Laure Gonnord
http://laure.gonnord.org/pro/teaching/capM1.html
Laure.Gonnord@ens-lyon.fr

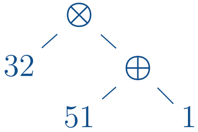
Master 1, ENS de Lyon

Janv 2017

Inspiration D. Hirschkoff for CAP 2015-16

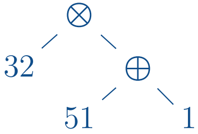
Arithmetic expressions and functions

32*(51+1)



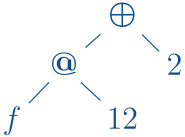
Arithmetic expressions and functions

32*(51+1)



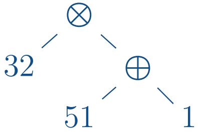

```
let f x = (3*x)
```

f(12)+2



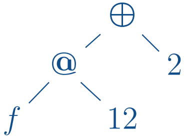
Arithmetic expressions and functions

32*(51+1)



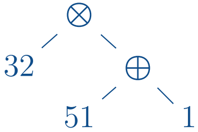

```
let f x = (3*x)
let f = fun x -> (3*x)
```

f(12)+2



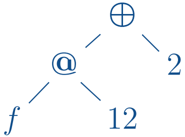
Arithmetic expressions and functions

32*(51+1)



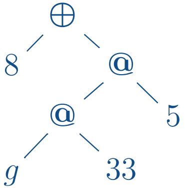

```
let f x = (3*x)
let f = fun x -> (3*x)
```

f(12)+2



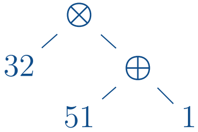

```
let g x y = 3*x+y
```

8 + (g 33 5)



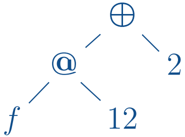
Arithmetic expressions and functions

32*(51+1)



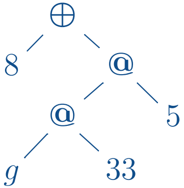

```
let f x = (3*x)
let f = fun x -> (3*x)
```

f(12)+2




```
let g x y = 3*x+y
let g = fun x -> (fun y -> 3*x+y)
```

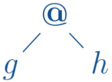
8 + (g 33 5)



Functions on the right (functions as arguments)

```
let g = fun f -> f 3
let h = fun x -> x+5
```

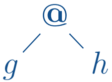
g h

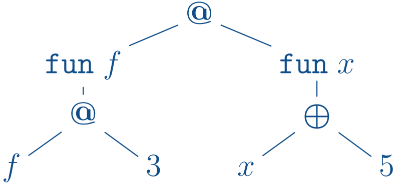


Functions on the right (functions as arguments)

```
let g = fun f -> f 3
let h = fun x -> x+5
```

g h

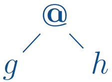


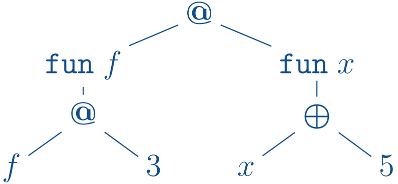


Functions on the right (functions as arguments)

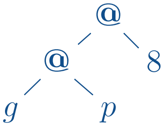
```
let g = fun f -> f 3
let h = fun x -> x+5
```

g h



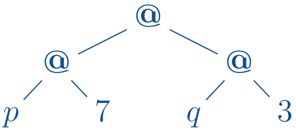



```
let p = fun x y -> x+y
let q = fun z -> z+2
```



g p 8

p 7 (q 3)



Syntax 1/2

- notation for applications : `g 3`
  - in maths : g(3)
  - sometimes `g@3` to stress that application is a binary operator
- using the `let` construct
  - a program is a sequence of `let`s, possibly followed by an expression (the “main”)
  - `let x = 3 in let y = 4 in let z = 5 in (x+y*z)`

    will also be written

```
let x = 3
let y = 4
let z = 5
(x+y*z)
```

Syntax 2/2

- a **nested** `let..in`

```
let f = fun x →
  let y = g (x*x) in
if y>0 then y else x
```

⟵ here is f's **return** (y or x)

μML, a small functional programming language

- syntax

$$
\begin{array}{lll}
e ::= & \texttt{fun}\ x \rightarrow e \mid e_1\ e_2 \mid x & \text{core functional language} \\
& \mid \texttt{let}\ x = e_1\ \texttt{in}\ e_2 & \\
& \mid e_1 + e_2 \mid 1, 2, 3, \ldots & \text{if you insist}
\end{array}
$$

$x, y, z, \ldots \in Vars$ variable identifiers

- operational semantics (see in course 03)
  - first version : $e \rightarrow^v v$ *no environment*
  - second version : $\sigma, e \rightarrow^v v$

"reduction semantics"

$$\frac{}{c \xrightarrow{v} c} \qquad \frac{}{op \xrightarrow{v} op} \qquad \frac{}{(\texttt{fun}\ x \rightarrow e) \xrightarrow{v} (\texttt{fun}\ x \rightarrow e)}$$

$$\frac{e_1 \xrightarrow{v} v_1 \quad e_2 \xrightarrow{v} v_2}{(e_1, e_2) \xrightarrow{v} (v_1, v_2)} \qquad \frac{e_1 \xrightarrow{v} v_1 \quad e_2[x \leftarrow v_1] \xrightarrow{v} v}{\texttt{let}\ x = e_1\ \texttt{in}\ e_2 \xrightarrow{v} v}$$

$$\frac{e_1 \xrightarrow{v} (\texttt{fun}\ x \rightarrow e) \quad e_2 \xrightarrow{v} v_2 \quad e[x \leftarrow v_2] \xrightarrow{v} v}{e_1\ e_2 \xrightarrow{v} v}$$

- But substitution costs.

A new version of semantics, with environments

Need: bind a variable x to a value v in a term a.

Inefficient approach: the textual substitution $a[x \leftarrow v]$.

Alternative: remember the binding $x \mapsto v$ in an auxiliary data structure called an environment. When we need the value of x during evaluation, just look it up in the environment.

The evaluation relation becomes $e \vdash a \Rightarrow v$
e is a partial mapping from names to values (CBV).

Additional evaluation rule for variables:

$$\frac{e(x) = v}{e \vdash x \Rightarrow v}$$

The following 5 slides are from X. Leroy

Lexical scoping : static binding

What to we want for x's value ?

```
let x = 1 in
let f = λy.x in
let x = "foo" in
f 0
```

- x should be 1 (value at the definition of `f`)

The notion of closure - Landin 1964

To implement lexical scoping, function abstractions $\lambda x.a$ must not evaluate to themselves, but to a function closure: a pair

$$(\lambda x.a)[e]$$

of the function text and an environment e associating values to the free variables of the function.

`let x = 1 in`	$\mathtt{x} \mapsto 1$
`let f = λy.x in`	$\mathtt{x} \mapsto 1;\ \mathtt{f} \mapsto (\lambda \mathtt{y}.\mathtt{x})[\mathtt{x} \mapsto 1]$
`let x = "foo" in`	$\mathtt{x} \mapsto$ "foo"; $\mathtt{f} \mapsto (\lambda \mathtt{y}.\mathtt{x})[\mathtt{x} \mapsto 1]$
`f 0`	evaluate `x` in environment $\mathtt{x} \mapsto 1;\ \mathtt{y} \mapsto 0$

Natural semantics with env and closures

Values: $v ::= N \mid (\lambda x.a)[e]$

Environments: $e ::= x_1 \mapsto v_1; \ldots; x_n \mapsto v_n$

$$\frac{e(x) = v}{e \vdash x \Rightarrow v} \qquad e \vdash N \Rightarrow N \qquad e \vdash \lambda x.a \Rightarrow (\lambda x.a)[e]$$

$$\frac{e \vdash a \Rightarrow (\lambda x.c)[e'] \quad e \vdash b \Rightarrow v' \quad e' + (x \mapsto v') \vdash c \Rightarrow v}{e \vdash a\ b \Rightarrow v}$$

- How to implement ?

What for ?

An implementation (among others) of the natural semantics with closures, in two steps :

- Compilation : Fun expression $\rightarrow$ list of instructions of an abstract machine
- Execution : eval of the abstract machine (implem of big step semantics of the AM).

Abstract machine instructions : (ocaml type)

```
type instr =
  (* Arithmetic fragment *)
  | Cst of int
  | Add
  (* Let fragment *)
  | Access of variable
  | Let of variable
  | EndLet
  (* Functional fragment *)
  | Closure of variable * code
  | App
  | Ret
```

A bit on compilation

The machine has a stack where to push everything to remember. To add two expressions, we have to push the two operands in the stack, then the `Add` instruction.

Fun expression $\rightarrow$ list of instructions :

- 42 ?
- $e_1 + e_2$?
- $\texttt{let}\ x = 1$?
- x ?
- $\texttt{fun}\ x \rightarrow 2$?
- $(\texttt{fun}\ x \rightarrow 2)7$?

Execution (semantics) of the abstract machine : expressions, local definition

Code	Env.	Stack	Code	Env.	Stack	Comment
`Cst k`;c	σ	s	c	σ	$k.s$	push the value in the stack
`Add`;c	σ	$k_1.k_2.s$	c	σ	$(k_1+k_2).s$	get operands, then add
`Access` x;c	σ	s	c	σ	$\sigma(x).s$	push the current value of x
`Let` x;c	σ	$z.s$	c	$(x,z).\sigma$	s	the value to bind is on top of stack
`EndLet`;c	$z.\sigma$	s	c	σ	s	unbind the last value.

Example to evaluate !

```
>c (let y = 40 in 2+y)
Cst 40; Let "y"; Cst 2 ; Access "y" ; Add; EndLet
```

Execution of the abstract machine : functions

Code	Env.	Stack	Code	Env.	Stack	Comment
`Clos(x,c');c`	σ	s	c	σ	$(x, c')[\sigma].s$	a new closure (x, c') on top of stack
`App`;c	σ	$(x, c')[\sigma'].v.s$	c'	$(x, v).\sigma'$	$c.\sigma.s$	push the code and initial env
`Ret`;c	σ	$v.c'.\sigma'.s'$	c'	σ'	$v.s$	stack top = returned value

Example to evaluate !

```
c ((fun x -> x+1) 42)
Cst 42; Closure("x", Access "x"; Cst 1; Add; Ret); App
```

Implementation of this abstract machine

See the lab :

- values are $VInt, VClosure \ldots$.
- environments as lists of $(var, value)$.
- stacks as lists of values.
- evaluation : $(code, env, stack) \rightarrow (code', env', stack')$ with empty env and empty stack at initial state.