Compiling a functional language

C. Paulin (courtesy of J.-C. Filliâtre)

M1 MPRI 2025–26

- Closures
- Tail calls
- Inline expansion
- Compiling pattern-matching

Introduction

Functional language

- Avoiding (or limiting) side-effects :
  - mathematical reasoning $f(3) = f(3)$
  - optimizing by code transformation easier
- Higher-order features (functions as first-class citizen) :
  - Functions as arguments
  - Functions as results of functions
  - Anonymous functions

The FUN language : concrete syntax

$$
\begin{aligned}
E ::= \;& n \mid \textbf{true} \mid \textbf{false} \\
& \mid x \mid E \circ\!\!\mathrm{p} E \\
& \mid \textbf{if } E \textbf{ then } E \textbf{ else } E \\
& \mid e\, e \mid \textbf{fun } x^{+} \rightarrow e \\
& \mid \textbf{let rec}?\ x\ x^{*}\ =\ E \textbf{ in } E
\end{aligned}
$$

- Syntactic sugar for function definition
  - function with only one argument : **fun** $x \rightarrow e$
  - local declaration **let** **rec**? $x =\ e$ **in** e

The FUN language : abstract syntax

```
type bop = Add | Sub | Mul | Lt | Eq (* / ... *)
type expr =
  | Int of int
  | Bop of bop * expr * expr
  | Var of string
  | Let of string * expr * expr
  | If  of expr * expr * expr
  | Fun of string * expr
  | App of expr * expr
  | Fix of string * expr
type prog = expr (* or (string * expr) list *)
```

Compiling a functional language

- Closures
- Tail calls
- Inline expansion
- Compiling pattern-matching

Representation of functions

- Formal parameters (arguments) + body
- When applied, we execute the body with actual values
- The body : code + a label to access it
- The body refers to variables : how to encode access to their values ?

What are the possible variables ?

- Global variables
- Arguments
- Local variables
- What else ?
  - depends on where functions can be defined/used
  - can/cannot appear inside another function ?
  - can/cannot be called outside the function which defines it

Stack allocation for imbricated functions

Example in Pascal

```
1  program main;
2  var f : integer;
3  procedure fib(n : integer);
4      procedure somme();
5      var tmp : integer;
6      begin
7      fib(n-2);
8      tmp := f;
9      fib(n-1);
10     f := f + tmp
11     end;
12 begin
13     if n <= 1 then f := n else somme()
14 end;
15 begin
16   fib(3);
17   writeln(f)
18 end.
```

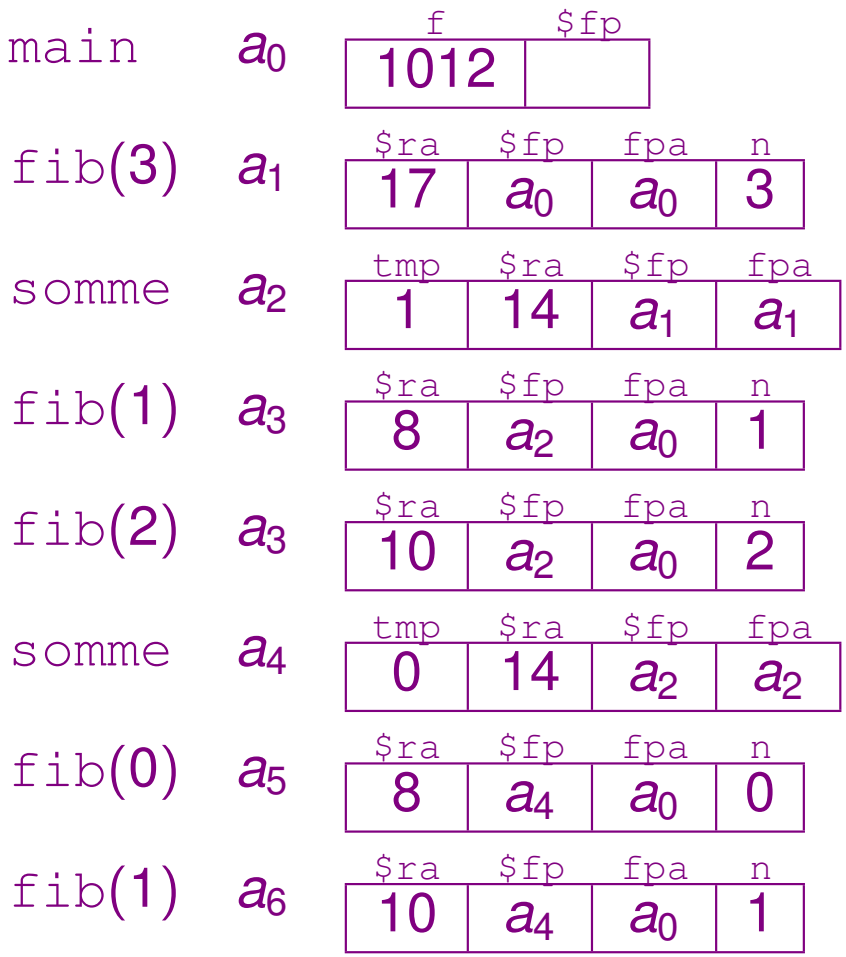
Stack frames

- we keep as before a pointer to the stack frame of the calling procedure
  - restored when the call is finished
  - *f* can be called from different functions
- an extra pointer to the last stack frame of the function *g* in which *f* was defined
  - *f* can access variables stored in the stack frame of *g* or one of its parents
- `fib` : argument *n* + saved values of `$fp` and `$ra` + stack frame of `main`

- `some` : saved values of `$fp` and `$ra` + local variable `tmp` + stack frame of `fib`

Dynamic behavior

```
main()
fib(3)
somme()
fib(1)
f:=1-ret fib(1)
tmp:=f
fib(2)
somme()
fib(0)
f:=0-ret fib(0)
tmp:=f
fib(1)
f:=1-ret fib(1)
f:=f+tmp
ret somme(),fib(2)
f:=f+tmp
ret somme(),fib(3)
```



Functions as output

- In Pascal, a function *f* defined in the procedure *g* will only be called in the body of *g*,
- There is at least one active frame-stack for *g*
- The function *f* can use variables introduced by *g*
- Does not work when functions are ordinary values

```
let g x =
  if x < 0 then fun y -> y - x
  else fun y -> y + x
let square f x = f (f x)
let main = square (g 5) 4
```

- How to design the code for the value of g x in a generic way ?
- This code mentions the argument x
- The value of x is in the stack frame of *g*
- This value disappears when g 5 returns !

Closure

- The solution is to use a structure called closure to represent a function
- The closure has 2 components
  - the address of the code to be executed
  - the environment : values of the variables used by the body of the function
- The closure is allocated in the heap :
  - if the closure is created by a function g it will still be accessible after g returns

Which variables in the environment ?

In the closure of the function **fun** x –> e we need all the values of the free variables

$$
\begin{aligned}
fv(c) &= \emptyset \\
fv(x) &= \{x\} \\
fv(e_1 \,\mathtt{op}\, e_2) &= fv(e_1) \cup fv(e_2) \\
fv(\mathtt{fun}\ x \to e) &= fv(e) \setminus \{x\} \\
fv(e_1\ e_2) &= fv(e_1) \cup fv(e_2) \\
fv(\mathtt{let}\ x = e_1\ \mathtt{in}\ e_2) &= fv(e_1) \cup (fv(e_2) \setminus \{x\}) \\
fv(\mathtt{let\ rec}\ x = e_1\ \mathtt{in}\ e_2) &= (fv(e_1) \cup fv(e_2)) \setminus \{x\} \\
fv(\mathtt{if}\ e_1\ \mathtt{then}\ e_2\ \mathtt{else}\ e_3) &= fv(e_1) \cup fv(e_2) \cup fv(e_3)
\end{aligned}
$$

Example

Program which approximates $\int_0^1 x^n dx$

```
let rec pow i x =
  if i = 0 then 1. else x *. pow (i-1) x

let integrate_xn n =
  let f = pow n in
  let eps = 0.001 in
  let rec sum x =
    if x >= 1. then 0. else f x +. sum (x +. eps) in
  sum 0. *. eps
```

Example

We explicit `fun` constructions

```
let rec pow =
  fun i ->
    fun x -> if i = 0 then 1. else x *. pow (i-1) x
```

- in the first closure, `fun i ->`, the envronment is $\{\texttt{pow}\}$
- in the second, `fun x ->`, it is $\{\texttt{i}, \texttt{pow}\}$

Example

```
let integrate_xn = fun n ->
  let f = pow n in
  let eps = 0.001 in
  let rec sum =
    fun x -> if x >= 1. then 0. else f x +. sum (x+.eps) in
  sum 0. *. eps
```

- for `fun n ->`, the environment is $\{\texttt{pow}\}$
- pour `fun x ->`, the environment is $\{\texttt{eps}, \texttt{f}, \texttt{sum}\}$

Representing a closure

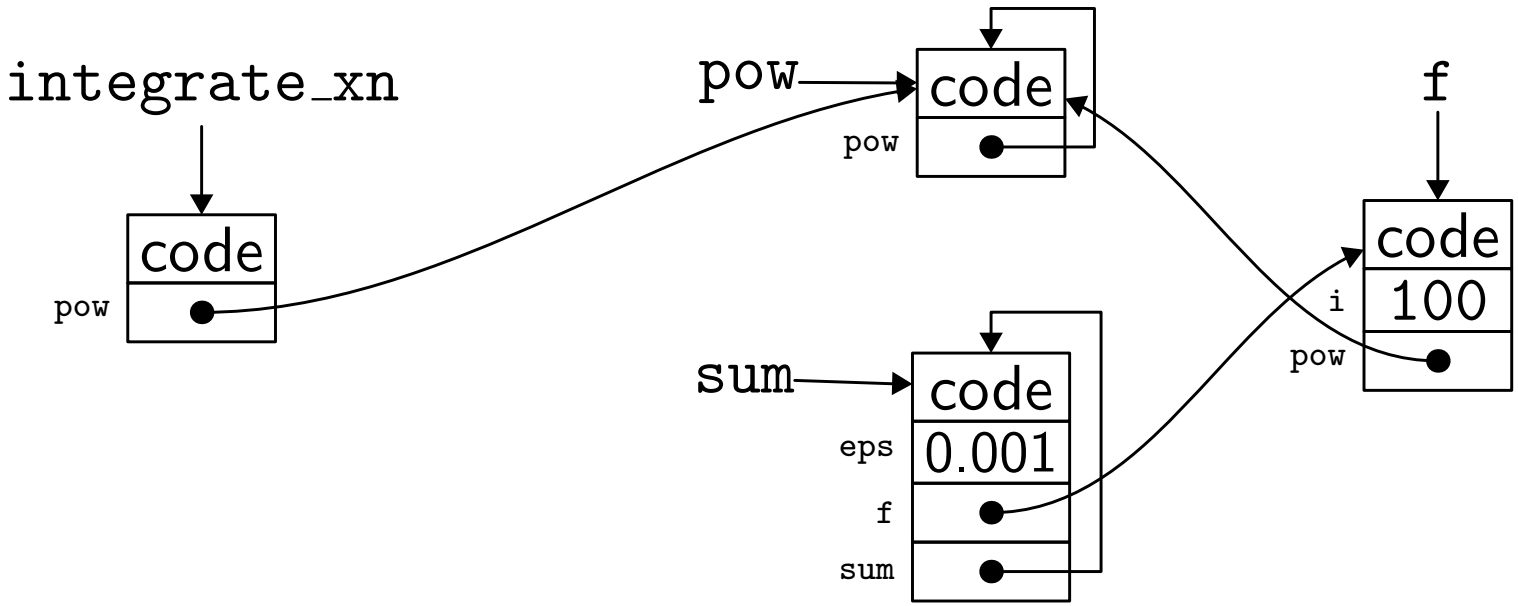
- a unique block in the heap
- the first field is the address of the code
- the remaining fields contain values of the free variables

(other solutions are possible : environment in a second bloc, linked closures, etc.)

Example

```
let rec pow i x = if i = 0 then 1. else x *. pow (i-1) x
let integrate_xn n =
  let f = pow n in
  let eps = 0.001 in
  let rec sum x = if x >= 1. then 0. else f x +. sum (x+.eps) in
  sum 0. *. eps
```

when executing `integrate_xn 100`, we have 4 closures :



Compiling

A relatively easy way to compile closures is to proceed with 2 phases

1. we look in the code for constructions $\texttt{fun}\ x \to e$ and we replace them with an explicit closure construction

$$\texttt{clos}\ f\ [y_1, \ldots, y_n]$$

with y_i the free variables of $\texttt{fun}\ x \to e$ and f a new name associated with a global declaration of function

$$\texttt{letfun}\ f\ [y_1, \ldots, y_n]\ x = e'$$

with e' obtained from e by recursively removing the constructions $\texttt{fun}$ (*closure conversion*)

2. we compile the obtained code which has only $\texttt{letfun}$ function declarations

Example

```
letfun fun2 [i,pow] x =
  if i = 0 then 1. else x *. pow (i-1) x
letfun fun1 [pow] i =
  clos fun2 [i,pow]
let rec pow =
  clos fun1 [pow]
letfun fun3 [eps,f,sum] x =
  if x >= 1. then 0. else f x +. sum (x +. eps)
letfun fun4 [pow] n =
  let f = pow n in
  let eps = 0.001 in
  let rec sum = clos fun3 [eps,f,sum] in
  sum 0. *. eps
let integrate_xn =
  clos fun4 [pow]
```

Change of representation

before

```
type var = string

type expr =
  | Evar of var
  | Efun of var * expr
  | Eapp of expr * expr
  | Elet of var * expr * expr
  | Eif
of expr * expr * expr
  | ...
type decl = var * expr

type prog = decl list
```

after

```
type var =
  | Vglobal of string
  | Vlocal  of int
  | Vclos   of int
  | Varg
type expr =
  | Evar  of var
  | Eclos of string * var list
  | Eapp  of expr * expr
  | Elet  of int * expr * expr
  | Eif
of expr * expr * expr
  | ...
type decl =
  | Let    of string * expr
  | Letfun of string * expr
type prog = decl list
```

Implementing

An ident in the body of the function can represent

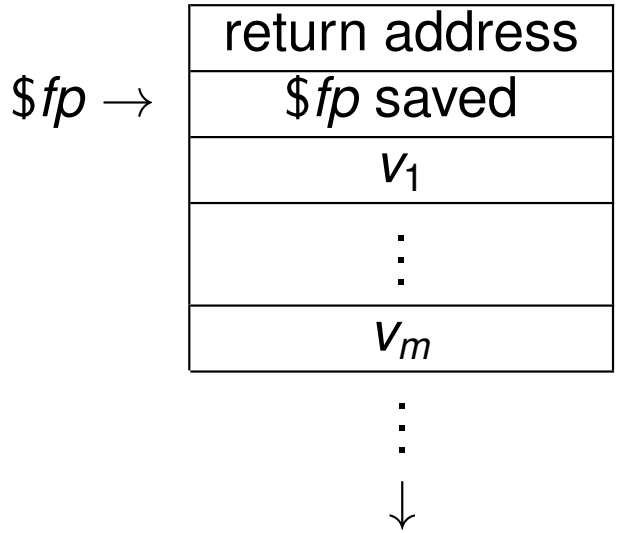
- `Vglobal` *s* : a global variable (introduced by `let`) with name *s*
- `Vlocal` *n* : a local variable (introduced by `let in`),
  at position *n* in the stack-frame
- `Vclos` *n* : a free variable in the closure, at position *n*
- `Varg` : the unique argument of the fonction (`x` in `fun x -> e`)

thi scope analysis of variable can be done simultaneously with closure conversion

Compiling scheme

each fonction has a unique argument (`Varg`), passed in register $\$a_0$
the cloture address is passed in the register $\$a_1$

the stack frame looks like :
with $v_1, \ldots, v_m$ places for local variables :
it is built by the callee



Compiling

We need to explain how to compile

- the construction of the closure `Eclos`(f, l)
- a function call `Eapp`(e_1, e_2)
- access to a variable `Evar` x
- function declaration `Letfun`(f, e)

Building a closure

Compiling

$$\texttt{Eclos}(f,\ [y_1, \ldots, y_n])$$

1. allocate a block of size $n + 1$ on the heap (using `malloc`)
2. put the f address in the field 0
   (f is a label refering to the address)
3. store the value of variables $y_1, \ldots, y_n$ in the fields 1 to n
4. returns the address of the block

note : we wait for the GC to free the block when possible

Function call

Compiling an application

$$\texttt{Eapp}(e_1, e_2)$$

1. compile e_1 and put its value (the address of a closure) in register $\$a1$
2. compile e_2 and put its value in register $\$a0$
3. call the function using the adress stored in the first field of the closure
   `jar ($a1)`

Accessing a variable

For compiling the accss to a variable

`Evar` *x*

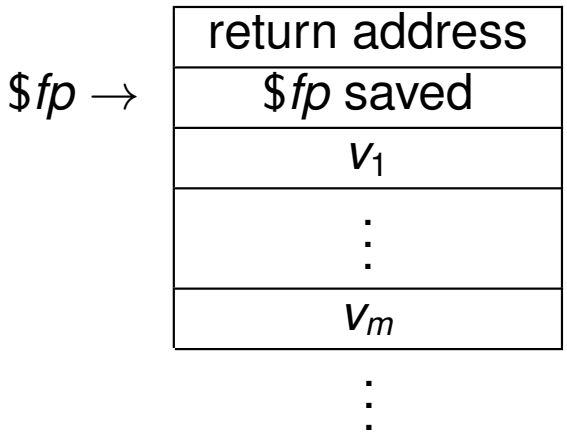
we have 5 possible cases

- `Vglobal` *s* : the value is at the address with label *s*
- `Vlocal` *n* : the value is at the address *n*($*fp*)
- `Vclos` *n* : the value is at the address *n*($*a*1)
- `Varg` : the value is in $*a*0

Function declaration

Compiling declaration

$$\texttt{Letfun}(f, e)$$



we do as usual for functions

1. we save and replace $\$fp$
2. we allocate the stack-frame (enough room for local variables) in e
3. we produce code to evaluate e with a result in $\$v0$
4. we free the stack-frame and restore $\$fp$
5. we return

Optimisations

If a function f is defined by

$$\texttt{let}\ f\ x_1 \ldots x_n = e$$

and called with n arguments

$$f\ e_1\ \ldots\ e_n$$

building the intermediate closures will be costly
We can implement a « traditional » with all arguments passed

However a partial application will generate a closure

OCaml does this optimisation ; on first-order code, the efficiency is the same as in a traditional (non functional) language

Optimisation

- If the closure does not survive the end of the function where it is defined, then the closure can be allocated on the stack
- A static analysis should be performed to make sure it is safe (*escape analysis*)

Example

```
let integrate_xn n =
  let f = ... in
  let eps = 0.001 in
  let rec sum x = if x >= 1. then 0. else f x +. sum (x+.eps) in
  sum 0. *. eps
```

Compiling a functional language

- Closures
- Tail calls
- Inline expansion
- Compiling pattern-matching

Tail call

Definition

A call to $(f\ e_1\ \ldots\ e_n)$ in the body of a function g is a tail call if it is the last instruction of g before returning its result.

- a *tail recursive function* is a function where all recursive calls are tail calls
- a tail call can be a non-recursive call
- in a recursive function, only some of the calls can be tail calls
- when f performs a tail call to g,
  - the stack-frame of the caller f contains the right return address for g
  - the local values stored in the stack frame of f will not be used after calling g
  - the stack frame of f can be reused a a stack frame for g
  - we can do a simple jump to the code of g
- tail-recursive functions can be compiled as efficiently as loops

Using higher-order functions

- How to get tail-recursive functions ?
- Example :

```
type 'a tree = Empty | Node of 'a tree * 'a * 'a tree
let rec height = function
| Empty            -> 0
| Node (l, _, r) -> 1 + max (height l) (height r)
```

Solution using continuations

- Instead of computing the height h, we compute $k(h)$ for an arbitrary function k (the continuation)

```
val heightc: 'a tree -> (int -> 'b) -> 'b
let height t = heightc t (fun h -> h)
```

What for ?

```
let rec heightc t k = match t with
  | Empty ->
      k 0
  | Node (l, _, r) ->
      heightc l (fun hl ->
      heightc r (fun hr ->
      k (1 + max hl hr)))
```

- Calls to `heightc` and `k` are now tail-calls
- The stack with remain constant
- The closure will be allocated on the heap

Compiling a functional language

- Closures
- Tail calls
- Inline expansion
- Compiling pattern-matching

Optimizing functions calls

- Introducing many functions can be good for code readability
- A function call is costly
  - building the stack-frame
  - jumping to another part of the code
- Optimization (register allocation) are done inside the body of a function

Inline expansion of function bodies

- If we have a function definition **let** f x1 ... xn = e
- We replace a function call f a1 ... an by
  let x1 = a1 **and** ... xn = an **in** e
- if ai is atomic we replace xi by ai in e
- We need to avoid variable capture (rename in order to have different variable names)

```
let x = 5 in
let g y = y + x in
let f x = g 1 + x
in f 2 + x
```

```
let x = 5 in
let g y = y + x in
let f x = 1+x+x
in f 2 + x
```

```
let x = 5 in
let g y = y + x in
let f a = 1+x+a
in f 2 + x
```

- If all occurrences of the function have been inlined, remove the function definition

Loop invariant hoisting

- Some recursive functions have “invariant” arguments

```
let rec f x1 .. xn = e[f,x1,..,xn]
```

- in e all recursive calls to f e1 .. en satisfies ek=xk for some k
- change the definition of f to avoid recomputing a constant parameter

```
let f x1 .. xn =
let rec f' (xi)_{i<>k}
      = e[(f e1 .. en) <- f' (ei)_{i<>k}]
in f' (xi)_{i<>k}
```

Example

```
let rec dolist f l c = match l with
  [] -> c ()
 |a::m -> let dorest () = dolist f m c
          in f a dorest
let double j = j + j
let print_double i c =
   let again () = putInt (double i) c
   in putInt i again

let print_table l c =
    dolist print_double l c
```

```
let dolist f l c =
  let rec dlrec l = match l with
    [] -> c ()
  |a::m -> let dorest () = dlrec m
    in f a dorest
    in dlrec l
...
let print_table l c =
   let rec dlrec l = match l with
    [] -> c ()
  |a::m -> let dorest () = dlrec m
    in print_double a dorest
   in dlrec l
```

Heuristics for program inlining

- Inlining makes code bigger, and has to stop at some point
- Inline some specific function calls that are frequently executed
- Inline functions that are only called once ! then delete the function definition
- Inline functions with very small bodies (compensates the extra instructions for function calls)

Compiling a functional language

- Closures
- Tail calls
- Inline expansion
- Compiling pattern-matching

Pattern-matching in ML

- *pattern matching* generalises conditional constructions
- used in combination with definition of algebraic data-types
- function definition

$$\texttt{function}\ p_1 \to e_1 \mid \ldots \mid p_n \to e_n$$

- genealized conditionals

$$\texttt{match}\ e\ \texttt{with}\ p_1 \to e_1 \mid \ldots \mid p_n \to e_n$$

- exceptions handling

$$\texttt{try}\ e\ \texttt{with}\ p_1 \to e_1 \mid \ldots \mid p_n \to e_n$$

Pattern-matching The compiler will translate these high-level constructions to elementary tests (testing constructors, comparing constants) and access to fields of structured values.

we only consider the construction

$$\texttt{match}\ x\ \texttt{with}\ p_1 \to e_1 \mid \ldots \mid p_n \to e_n$$

Pattern

A *pattern* is defined using the abstract syntax

$$p ::= x \mid C(p, \ldots, p)$$

with C a constructor, which can be

- a constant like `false`, `true`, 0, 1, `"hello"`, etc.
- a constant constructor of an algebraich types , like `[]` or for example `Empty` déclared by `type t = Empty | ...`
- a constructor with arity $n \geq 1$ like `::` or for instance `Node` déclared by `type t = Node of t * t | ...`
- a constructor for n-tuple, with $n \geq 2$

Linear patterns

Definition (linear patterns)

A pattern p is linear if a variable appears at most once in p.

example : the pattern (x, y) is linear, but (x, x) is not

note : OCaml accepts non linear patterns only in OR patterns

```
# let (x,x) = (1,2);;
```

Error : Variable x is bound several times in this matching

```
# let x,0 | 0,x = ...;;
```

We only consider linear patterns and no disjunctive patterns (can be rewritten as several disjoint cases)

Pattern and values

The values are

$$v ::= C(v, \ldots, v)$$

with C the set of constants and constructors defined like for patterns

Definition (pattern-matching)

A value v matches a pattern p if there exists a substitution σ (which replaces variables by values) such that $v = \sigma(p)$.

note : we may assume that the domain of σ
is exactly the set of variables of p

Result

Any value matches a pattern $p = x$ which is just a variable ; PROP A value v matches $p = C(p_1, \ldots, p_n)$ iff v is equal to $v = C(v_1, \ldots, v_n)$ and v_i matches p_i for all $i = 1, \ldots, n$. proof :
It works because we have linear patterns, counter-example with the pattern $C(x, x)$ and the value $C(0, 1)$

Pattern-matching with several cases

Definition

In the pattern-matching

$$\texttt{match}\ x\ \texttt{with}\ p_1 \to e_1 \mid \ldots \mid p_n \to e_n$$

if v is the value of x, we say that v matches the case p_i if v matches p_i but v does not match p_j for $j < i$.

The result of the matching is $\sigma(e_i)$, with σ a substitution such that $\sigma(p_i) = v$.

if v does not match any p_i, then the matching raises an error (exception `Match_failure` in OCaml)

Compiling pattern-matching

First algorithm :

We suppose given

- $constr(e)$, returns the constructor for the value of e,
- $\#_i(e)$, returns the i-ième component of the value

If $e = C(v_1, \ldots, v_n)$ then $constr(e) = C$ and $\#_i(e) = v_i$

Compiling pattern-matching

We start by compiling d'one line

$$code(\texttt{match}\ e\ \texttt{with}\ p \to action) = F(p, e, action)$$

with the compilation function F defined by :

$$\begin{array}{l}
F(x, e, action) = \\
\quad \texttt{let}\ x = e\ \texttt{in}\ action \\
F(C, e, action) = \\
\quad \texttt{if}\ constr(e) = C\ \texttt{then}\ action\ \texttt{else}\ error \\
F(C(p), e, action) = \\
\quad \texttt{if}\ constr(e) = C\ \texttt{then}\ F(p, \#_1(e), action)\ \texttt{else}\ error \\
F(C(p_1, \ldots, p_n), e, action) = \\
\quad \texttt{if}\ constr(e) = C\ \texttt{then} \\
\qquad F(p_1, \#_1(e), F(p_2, \#_2(e), \ldots F(p_n, \#_n(e), action)\ldots) \\
\quad \texttt{else}\ error
\end{array}$$

Example

```
match x with 1 :: y :: z -> y + length z
```

its compilation gives the (pseudo-)code :

```
if constr(x) = :: then
  if constr(#1(x)) = 1 then
    if constr(#2(x)) = :: then
      let y = #1(#2(x)) in
      let z = #2(#2(x)) in
      y + length(z)
    else error
  else error
else error
```

note : $\#_2$`(x)` is computed several times, we could use `let` expressions in the definition of F to avoid that.

Correctness

if $e \xrightarrow{*} v$ alors

$$F(p, e, action) \xrightarrow{*} \sigma(action) \qquad \text{if there exists } \sigma \text{ s.t. } v = \sigma(p)$$
$$F(p, e, action) \xrightarrow{*} error \qquad \text{otherwise}$$

proof : by induction on p

Dealing with several lines

We replace *error* by interpreting next line

$$\mathit{code}(\texttt{match}\ x\ \texttt{with}\ p_1 \to e_1 \mid \ldots \mid p_n \to e_n) = \\ F(p_1, x, e_1, F(p_2, x, e_2, \ldots F(p_n, x, e_n, \mathit{error}) \ldots))$$

F now defined by :

$$F(x, e, \mathit{success}, \mathit{error}) = \\ \quad \texttt{let}\ x = e\ \texttt{in}\ \mathit{success}$$

$$F(C, e, \mathit{success}, \mathit{error}) = \\ \quad \texttt{if}\ \mathit{constr}(e) = C\ \texttt{then}\ \mathit{success}\ \texttt{else}\ \mathit{error}$$

$$F(C(p_1, \ldots, p_n), e, \mathit{success}, \mathit{error}) = \\ \quad \texttt{if}\ \mathit{constr}(e) = C\ \texttt{then} \\ \quad\quad F(p_1, \#_1(e), F(p_2, \#_2(e), \ldots F(p_n, \#_n(e), \mathit{success}, \mathit{error}) \ldots, \mathit{error}) \\ \quad \texttt{else}\ \mathit{error}$$

Example

```
match x with [] -> 1 | 1 :: y -> 2 | z :: y -> z
```

```
if constr(x) = [] then
  1
 else
  if constr(x) = :: then
    if constr(#1(x)) = 1 then
      let y = #2(x) in 2
    else
      if constr(x) = :: then
        let z = #1(x) in let y = #2(x) in z
      else error
  else
    if constr(x) = :: then
      let z = #1(x) in let y = #2(x) in z
    else error
```

Problems

Its not very efficient because

- we do the same tests on different lines
- some tests are useless : if $constr(e) \neq [\,]$ then we can deduce $constr(e) = ::$)

Another algorithm

We consider another algorithm which handle all the lines simultaneously

the problem is represented as a matrix

$$\left|\begin{array}{cccccc} e_1 & e_2 & \dots & e_m & & \\ p_{1,1} & p_{1,2} & \dots & p_{1,m} & \to & action_1 \\ \vdots & \vdots & \ddots & \vdots & \vdots & \vdots \\ p_{n,1} & p_{n,2} & \dots & p_{n,m} & \to & action_n \end{array}\right|$$

correspondin to the pattern-matching

$$\begin{array}{l} \texttt{match}\ (e_1, e_2, \dots, e_m)\ \texttt{with} \\ \mid (p_{1,1}, p_{1,2}, \dots, p_{1,m}) \to action_1 \\ \mid \dots \\ \mid (p_{n,1}, p_{n,2}, \dots, p_{n,m}) \to action_n \end{array}$$

Another algorithm

The algorithm F proceeds recursively on the matrix

- $n = 0$

$$F \left| \begin{array}{ccc} e_1 & \cdots & e_m \end{array} \right| = error$$

- $m = 0$

$$F \left| \begin{array}{cc} \rightarrow & action_1 \\ & \vdots \\ \rightarrow & action_n \end{array} \right| = action_1$$

Another algorithm

If the left column has only variables

$$M = \left| \begin{array}{cccc} e_1 & e_2 & \dots & e_m & \\ x_{1,1} & p_{1,2} & \dots & p_{1,m} & \rightarrow \quad action_1 \\ \vdots & & & & \\ x_{n,1} & p_{n,2} & \dots & p_{n,m} & \rightarrow \quad action_n \end{array} \right|$$

on élimine cette colonne en introduisant des `let`

$$F(M) = F \left| \begin{array}{ccccc} e_2 & \dots & e_m & & \\ p_{1,2} & \dots & p_{1,m} & \rightarrow & \texttt{let}\ x_{1,1} = e_1\ \texttt{in}\ action_1 \\ \vdots & & & & \\ p_{n,2} & \dots & p_{n,m} & \rightarrow & \texttt{let}\ x_{n,1} = e_1\ \texttt{in}\ action_n \end{array} \right|$$

Another algorithm

The left colums has at least one pattern starting with a constructor (or constant)

For instance we have three different constructors : C arity 1, D arity 0 and E arity 2

$$M = \left| \begin{array}{llllll} e_1 & e_2 & \dots & e_m & & \\ C(q) & p_{1,2} & \dots & p_{1,m} & \to & action_1 \\ D & p_{2,2} & & p_{2,m} & \to & action_2 \\ x & p_{3,2} & & p_{3,m} & \to & action_3 \\ E(r,s) & p_{4,2} & & p_{4,m} & \to & action_4 \\ y & p_{5,2} & & p_{5,m} & \to & action_5 \\ C(t) & p_{6,2} & & p_{6,m} & \to & action_6 \\ E(u,v) & p_{7,2} & \dots & p_{7,m} & \to & action_7 \end{array} \right|$$

for each constructor C, D et E, we built a sub-matrix with all lines that could match a value starting with this constructor correspondant au filtrage d'une valeur pour ce constructeur

Another algorithm

$$M = \left| \begin{array}{llllll} e_1 & e_2 & \dots & e_m & & \\ \underline{C(q)} & p_{1,2} & \dots & p_{1,m} & \to & action_1 \\ D & p_{2,2} & & p_{2,m} & \to & action_2 \\ \underline{x} & p_{3,2} & & p_{3,m} & \to & action_3 \\ E(r,s) & p_{4,2} & & p_{4,m} & \to & action_4 \\ \underline{y} & p_{5,2} & & p_{5,m} & \to & action_5 \\ \underline{C(t)} & p_{6,2} & & p_{6,m} & \to & action_6 \\ E(u,v) & p_{7,2} & \dots & p_{7,m} & \to & action_7 \end{array} \right|$$

so

$$M_C = \left| \begin{array}{llllll} \#_1(e_1) & e_2 & \dots & e_m & & \\ q & p_{1,2} & \dots & p_{1,m} & \to & action_1 \\ \_ & p_{3,2} & & p_{3,m} & \to & \texttt{let } x = e_1 \texttt{ in } action_3 \\ \_ & p_{5,2} & & p_{5,m} & \to & \texttt{let } y = e_1 \texttt{ in } action_5 \\ t & p_{6,2} & \dots & p_{6,m} & \to & action_6 \end{array} \right|$$

Another algorithm

$$M = \left| \begin{array}{lllll} e_1 & e_2 & \dots & e_m & \\ C(q) & p_{1,2} & \dots & p_{1,m} & \rightarrow \ action_1 \\ \underline{D} & p_{2,2} & & p_{2,m} & \rightarrow \ action_2 \\ \underline{x} & p_{3,2} & & p_{3,m} & \rightarrow \ action_3 \\ E(r,s) & p_{4,2} & & p_{4,m} & \rightarrow \ action_4 \\ \underline{y} & p_{5,2} & & p_{5,m} & \rightarrow \ action_5 \\ C(t) & p_{6,2} & & p_{6,m} & \rightarrow \ action_6 \\ E(u,v) & p_{7,2} & \dots & p_{7,m} & \rightarrow \ action_7 \end{array} \right|$$

so

$$M_D = \left| \begin{array}{lll} e_2 & \dots & e_m \\ p_{2,2} & & p_{2,m} \rightarrow action_2 \\ p_{3,2} & & p_{3,m} \rightarrow \texttt{let}\ x = e_1\ \texttt{in}\ action_3 \\ p_{5,2} & \dots & p_{5,m} \rightarrow \texttt{let}\ y = e_1\ \texttt{in}\ action_5 \end{array} \right|$$

Another algorithm

$$M = \left| \begin{array}{lllll} e_1 & e_2 & \ldots & e_m & \\ C(q) & p_{1,2} & \ldots & p_{1,m} & \rightarrow \ action_1 \\ D & p_{2,2} & & p_{2,m} & \rightarrow \ action_2 \\ \underline{x} & p_{3,2} & & p_{3,m} & \rightarrow \ action_3 \\ \underline{E(r,s)} & p_{4,2} & & p_{4,m} & \rightarrow \ action_4 \\ \underline{y} & p_{5,2} & & p_{5,m} & \rightarrow \ action_5 \\ C(t) & p_{6,2} & & p_{6,m} & \rightarrow \ action_6 \\ \underline{E(u,v)} & p_{7,2} & \ldots & p_{7,m} & \rightarrow \ action_7 \end{array} \right|$$

so

$$M_E = \left| \begin{array}{lllll} \#_1(e_1) & \#_2(e_1) & e_2 \quad \ldots & e_m & \\ \_ & \_ & p_{3,2} & p_{3,m} & \rightarrow \ \texttt{let}\ x = e_1\ \texttt{in}\ action_3 \\ r & s & p_{4,2} & p_{4,m} & \rightarrow \ action_4 \\ \_ & \_ & p_{5,2} & p_{5,m} & \rightarrow \ \texttt{let}\ y = e_1\ \texttt{in}\ action_5 \\ u & v & p_{7,2} \quad \ldots & p_{7,m} & \rightarrow \ action_7 \end{array} \right|$$

Another algorithm

we define a submatrix for all remaining values (with constructors different from C, D and E), ie variables

$$M_R = \left| \begin{array}{lll} e_2 & \dots & e_m \\ p_{3,2} & & p_{3,m} \quad \to \quad \texttt{let } x = e_1 \texttt{ in } action_3 \\ p_{5,2} & \dots & p_{5,m} \quad \to \quad \texttt{let } y = e_1 \texttt{ in } action_5 \end{array} \right|$$

Another algorithm

we define

$$
\begin{aligned}
F(M) = \quad & \texttt{case}\ \mathit{constr}(e_1)\ \texttt{in} \\
& C \Rightarrow F(M_C) \\
& D \Rightarrow F(M_D) \\
& E \Rightarrow F(M_E) \\
& \texttt{otherwise} \Rightarrow F(M_R)
\end{aligned}
$$

Termination

The algorithm terminates

The value

$$\sum_{i,j} size(p_{i,j})$$

decreases strictly with each recursive call to F, with

$$\begin{array}{rcl} size(x) & = & 1 \\ size(C(p_1, \ldots, p_n)) & = & 1 + \sum_{i=1}^{n} size(p_i) \end{array}$$

Correctness

indication : use the interprétation of the matrix as

$$
\begin{array}{l}
\texttt{match}\ (e_1, e_2, \ldots, e_m)\ \texttt{with} \\
\mid (p_{1,1}, p_{1,2}, \ldots, p_{1,m}) \rightarrow \mathit{action}_1 \\
\mid \ldots \\
\mid (p_{n,1}, p_{n,2}, \ldots, p_{n,m}) \rightarrow \mathit{action}_n
\end{array}
$$

Efficiency

The type of the expression e_1 leads to optimizations :

$$\begin{array}{l}\texttt{case } \mathit{constr}(e_1) \texttt{ in} \\ \quad C \Rightarrow F(M_C) \\ \quad D \Rightarrow F(M_D) \\ \quad E \Rightarrow F(M_E) \\ \quad \texttt{otherwise} \Rightarrow F(M_R)\end{array}$$

in many cases :

- no test if only one constructor (for instance a n-tuple) : $F(M) = F(M_C)$
- no `otherwise` case when C, D and E are the only constructors
- a simple `if then else` if only 2 constructeurs
- a table for jumping directly to the appropriate case when finitely many constructors
- a binary tree or hash-table when infinitely many constructors (integers or strings)

Example

considérons

```
match x with [] -> 1 | 1 :: y -> 2 | z :: y -> z
```

matrix

$$M = \left| \begin{array}{lll} x & & \\ [\,] & \to & 1 \\ 1::y & \to & 2 \\ z::y & \to & z \end{array} \right|$$

We obtain

```
case constr(x) in
  [] -> 1
  :: -> case constr(#1(x)) in
          1 -> let y = #2(x) in 2
          otherwise ->
            let z = #1(x) in let y = #2(x) in z
```

Bonus

- We detect redondant cases
  when an action does not appear in the code produced

  example

```
match x with false -> 1 | true -> 2 | false -> 3
```

  gives

```
case constr(x) in false -> 1 | true -> 2
```

- We detect non exhaustive patterns
  error appears in the code

  example

```
match x with 0 -> 0 | 1 -> 1
```

  gives

```
case constr(x) in 0 -> 0 | 1 -> 1
                  | otherwise -> error
```

Summary

- Functions as first-class citizen : code + environment (values for free variables)
- Variables correspond to different locations in the memory
  - Static addresses for global variables
  - Stack for values with a known limited lifetime
  - Registers for actual parameters, some local variables. . .
  - Heap for other values
- Generating access code :
  - Convention for a static place (register) where to find the (dynamic) address of the block (+ a static shift value)
- Code transformation
  - Understanding the semantics of high-level features by translation into “simpler code” (closer to the machine-level architecture)
  - Sometimes transformation is done in the programming language itself
  - Many possible optimizations (preserving the semantics)