Interpretation and Compilation of Languages

Master Programme in Computer Science

Mário Pereira mjp.pereira@fct.unl.pt

Nova School of Science and Technology, Portugal

May 12, 2025

Lecture 10

based on lectures by Jean-Christophe Filliâtre and Léon Gondelman
previous editions by João Costa Seco, Luís Caires, and Bernardo Toninho

Today: Compilation of Functional Languages

1. First-class functions
2. Tail call optimization
3. Pattern-matching

First-class Functions

On key aspect of functional programming is first-class functions, which means that a function is a value like any other.

In particular, we can

- receive a function as a parameter
- return a function as a result
- store a function in a data structure
- build new functions dynamically

The ability to pass functions as parameters already exists in languages such as ALGOL, PASCAL, ADA, etc.

Similarly, the notion of function pointers already exists (FORTRAN, C, C++, etc.)

But the notion of first-class functions is strictly more powerful.

Let us illustrate it with OCAML.

Let us consider this fragment of OCaml.

$$
\begin{array}{rcl}
e & ::= & c \\
& | & x \\
& | & \texttt{fun}\ x \rightarrow e \\
& | & e\ e \\
& | & \texttt{let}\ [\texttt{rec}]\ x = e\ \texttt{in}\ e \\
& | & \texttt{if}\ e\ \texttt{then}\ e\ \texttt{else}\ e \\
\\
d & ::= & \texttt{let}\ [\texttt{rec}]\ x = e \\
\\
p & ::= & d\ \dots\ d
\end{array}
$$

Functions can be nested:

```
let sum n =
  let f x = x * x in
  let rec loop i =
    if i = n then 0 else f i + loop (i+1)
  in
  loop 0
```

Scoping is as usual.

(`let f x = x * x` is sugar for `let f = fun x -> x * x`)

Nested functions also exist in languages such as Algol, Pascal, Ada, etc.

The compilation schema uses the fact that every referenced variable is allocated somewhere in a function frame somewhere in the stack.

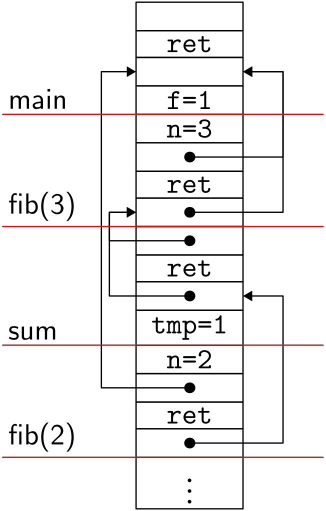
The compiler *concatenates* the function frames, so that it can retrieve the value of some variable.

```
program fib;

var f : integer;

procedure fib(n : integer);
   procedure sum();
   var tmp : integer;
   begin
     fib(n-2); tmp := f;
     fib(n-1); f := f + tmp
   end;
begin
   if n <= 1 then f := n else sum()
end;

begin fib(3); writeln(f) end.
```



We can pass functions as parameters

```
let square f x =
  f (f x)
```

and return functions

```
let f x =
  if x < 0 then fun y -> y - x else fun y -> y + x
```

Here, the function returned by `f` uses `x` but the stack frame for `f` just disappeared!

So we cannot compile functions in the usual way.

The solution is to use a closure (in Portuguese, fecho).

This is a heap-allocated data structure (to survive function calls) containing

- a pointer to the code (of the function body)
- the values of the variables that may be needed by this code; this is called the environment

P. J. Landin, *The Mechanical Evaluation of Expressions*,
The Computer Journal, 1964

What are the variables to be stored in the environment of the closure representing $\texttt{fun}\ x \rightarrow e$?

These are the free variables of $\texttt{fun}\ x \rightarrow e$.

The set $fv(e)$ of the free variables of the expression e is computed as follows:

$$
\begin{array}{rcl}
fv(c) & = & \emptyset \\
fv(x) & = & \{x\} \\
fv(\texttt{fun}\ x \rightarrow e) & = & fv(e) \setminus \{x\} \\
fv(e_1\ e_2) & = & fv(e_1) \cup fv(e_2) \\
fv(\texttt{let}\ x = e_1\ \texttt{in}\ e_2) & = & fv(e_1) \cup (fv(e_2) \setminus \{x\}) \\
fv(\texttt{let rec}\ x = e_1\ \texttt{in}\ e_2) & = & (fv(e_1) \cup fv(e_2)) \setminus \{x\} \\
fv(\texttt{if}\ e_1\ \texttt{then}\ e_2\ \texttt{else}\ e_3) & = & fv(e_1) \cup fv(e_2) \cup fv(e_3)
\end{array}
$$

Let us consider the following program approximating $\int_0^1 x^n dx$

```
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
```

Let us make constructions `fun` explicit and let us consider the closures

```
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 environment is {`pow`}
- in the second closure, `fun x ->`, it is {`i`,`pow`}

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

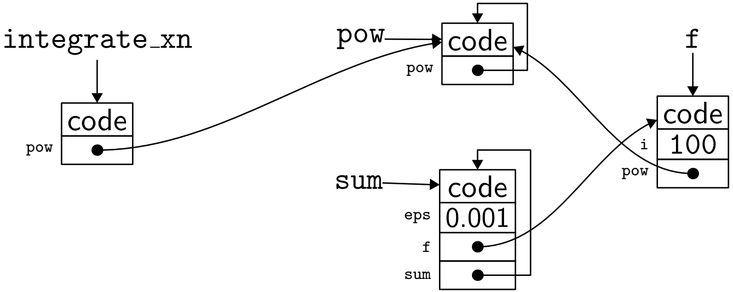
The closure is a single heap-allocated block, whose

- first field contains the code pointer
- next fields contains the values of the free variables (the environment)

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
```

During the execution of `integrate_xn 100`, we have four closures:



A good way to compile closures is to proceed in two steps:

1. first, we replace all expressions `fun` $x \to e$ by explicit closure constructions

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

   where the y_i are the free variables of `fun` $x \to e$ and f is the name of a global function

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

   where e' is derived from e by replacing constructions `fun` recursively (this is called closure conversion)

2. we compile the obtained code, which only contains `letfun` function declarations

On the example, we get

```
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]
```

Before

$$
\begin{array}{rcl}
e & ::= & c \\
 & | & x \\
 & | & \texttt{fun}\ x \rightarrow e \\
 & | & e\ e \\
 & | & \texttt{let}\ [\texttt{rec}]\ x = e\ \texttt{in}\ e \\
 & | & \texttt{if}\ e\ \texttt{then}\ e\ \texttt{else}\ e \\
\\
d & ::= & \texttt{let}\ [\texttt{rec}]\ x = e \\
\\
p & ::= & d\ \dots\ d
\end{array}
$$

After

$$
\begin{array}{rcl}
e & ::= & c \\
 & | & x \\
 & | & \texttt{clos}\ f\ [x, \dots, x] \\
 & | & e\ e \\
 & | & \texttt{let}\ [\texttt{rec}]\ x = e\ \texttt{in}\ e \\
 & | & \texttt{if}\ e\ \texttt{then}\ e\ \texttt{else}\ e \\
\\
d & ::= & \texttt{let}\ [\texttt{rec}]\ x = e \\
 & | & \texttt{letfun}\ f\ [x, \dots, x]\ x = e \\
\\
p & ::= & d\ \dots\ d
\end{array}
$$

In the new syntax trees, an identifier x can be

- a global variable introduced by `let`
  (allocated in the data segment)
- a local variable introduced by `let in`
  (allocated in the stack frame / a register)
- a variable contained in a closure
- the argument of a function (the `x` of `fun x -> e`)

Each function has a single argument, passed in register `%rdi`.

The closure is passed in register `%rsi`.

The stack frame is as follows,
where $v_1, \ldots, v_m$ are the local variables

	return address
`%rbp` $\rightarrow$	saved `%rbp`
	v_1
	$\vdots$
	v_m
	$\vdots$
	$\downarrow$

Let us detail how to compile

- the construction of a closure `clos` $f\ [y_1, \dots, y_n]$
- a function call $e_1\ e_2$
- the access to a variable x
- a function declaration `letfun` $f\ [y_1, \dots, y_n]\ x = e$

To compile

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

we proceed as follows:

1. we allocate a block of size $n+1$ on the heap (with `malloc`)
2. we store the address of f in field 0
   (f is a label in the assembly code and we get its address with $\$f$)
3. we store the values of the variables $y_1, \ldots, y_n$ in fields 1 to n
4. we return a pointer to the block

Note: we delegate the deallocation of the block to the Garbage Collector.

To compile a function call

$$e_1 \ e_2$$

we proceed as follows:

1. we compile e_1 into register `%rsi`
   (its value is a p_1 to a closure)
2. we compile e_2 into register `%rdi`
3. we call the function whose address is contained in the first field of the closure, with `call *(%rsi)`

   this is a jump to dynamic address
   (similar to what we did last week to compile OO languages)

To compile the access to the variable x, we distinguish four cases

global variable
: the value is stored at the address given by label x

local variable
: the value is at n(`%rbp`) / in a register

variable contained in a closure
: the value is at n(`%rsi`)

function argument
: the value is in register `%rdi`

Function Declaration

	return address
`%rbp` $\rightarrow$	saved `%rbp`
	v_1
	$\vdots$
	v_m
	$\vdots$

Last, to compile the declaration

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

we proceed as for a usual function declaration:

1. save and set `%rbp`
2. allocate the frame (for the local variables of e)
3. evaluate e in register `%rax`
4. delete the stack frame and restore `%rbp`
5. execute `ret`

It is rather inefficient to allocate intermediate closures for a call where n arguments are given

$$f \ e_1 \ \dots \ e_n$$

and where function f is defined by

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

Hence, a «traditional» call can be done, where all the arguments are given at once.

On the other hand, a partial application of f must produce a closure.

OCaml implements such an optimization ; for «first-order» code we get the same efficiency as for a non-functional language.

Today we find closures in

- Java (since 2014 and Java 8)
- C++ (since 2011 and C++11)

In these languages, anonymous functions are called lambdas.

A function is a regular object, with a method `apply`

```
LinkedList<B> map(LinkedList<A> l, Function<A, B> f) {
  ... f.apply(x) ...
}
```

An anonymous function is introduced with `->`

```
map(l, x -> { System.out.print(x); return x+y; })
```

The compiler builds a closure object (here capturing the value of `y`) with a method `apply`.

An anonymous function is introduced with []

```
for_each(v.begin(), v.end(), [y](int &x){ x += y; });
```

We specify the variables captured in the closure (here y).

The default behavior is to capture by value.

We may specify a capture by reference instead (here of s).

```
for_each(v.begin(), v.end(), [y,&s](int x){ s += y*x; });
```

Tail Call Optimization

Tail Call

Definition

We say that a function call $f(e_1, \ldots, e_n)$ that appears in the body of a function g is a tail call if this is the last thing that g computes before it returns.

By extension, we can say that a function is a tail recursive function if it is a recursive function whose recursive calls are all tail calls.

A tail call is not necessarily a recursive call

```
int g(int x) {
  int y = x * x;
  return f(y);
}
```

In a recursive function, we may have recursive calls that are tail calls and others that are not

```
int f91(int n) {
  if (n > 100) return n - 10;
  return f91(f91(n + 11));
}
```

What is the interest of tail calls?

We can delete the stack frame of the function performing the tail call before we make the call, since it is not needed afterwards.

Better, we can reuse it to make the tail call (in particular, the return address is the right one).

Said otherwise, we can make a `jump` rather than a `call`.

```c
int fact(int acc, int n) {
  if (n <= 1) return acc;
  return fact(acc * n, n - 1);
}
```

Traditional compilation

```
fact: cmpq  $1, %rsi
      jle   L0
      imulq %rsi, %rdi
      decq  %rsi
      call  fact
      ret
L0:   movq  %rdi, %rax
      ret
```

Optimization

```
fact: cmpq  $1, %rsi
      jle   L0
      imulq %rsi, %rdi
      decq  %rsi
      jmp   fact    # <-

L0:   movq  %rdi, %rax
      ret
```

The result is a loop.

The code is indeed identical to the compilation of

```
int fact(int acc, int n) {
  while (n > 1) {
    acc *= n;
    n--;
  }
  return acc;
}
```

The compiler `gcc` optimizes tail calls when we pass option `-foptimize-sibling-calls` (included in option `-O2`).

Have a look at the code produced by `gcc -O2` on programs such as `fact` or those of slide 34.

In particular, we notice that

```
int f91(int n) {
  if (n > 100) return n - 10;
  return f91(f91(n + 11));
}
```

is compiled exactly as if we were compiling

```
int f91(int n) {
  while (n <= 100)
    n = f91(n + 11);
  return n - 10;
}
```

The OCaml compiler optimizes tail calls by default.

The compilation of

```
let rec fact acc n =
  if n <= 1 then acc else fact (acc * n) (n - 1)
```

is a loop, as with the C program.

Even if we started with a functional program (variables acc and n are immutable).

With tail call optimization, we get a more efficient compiled program

since we have reduced memory access (we do not use `call` and `ret` anymore, which manipulate the stack).

On the `fact` example, the stack space becomes constant

In particular, we avoid any stack overflow due to a too large number of nested calls.

```
Stack overflow during evaluation (looping recursion?).
```

```
Fatal error: exception Stack_overflow
```

```
Exception in thread "main" java.lang.StackOverflowError
```

```
Segmentation fault
```

etc.

If we implement quicksort as follows

```
void quicksort(int a[], int l, int r) {
  if (r - l <= 1) return;
  // partition a[l..r[ in three
  //     l        lo        hi        r
  //    +----+----+-----+
  //   a|...<p...|...=p...|...>p...|
  //    +----+----+-----+
  ...
  quicksort(a, l, lo);
  quicksort(a, hi, r);
}
```

we can overflow the stack.

But if we make the first recursive call on the smallest half

```
void quicksort(int a[], int l, int r) {
  ...
  if (lo - l < r - hi) {
    quicksort(a, l, lo);
    quicksort(a, hi, r);
  } else {
    quicksort(a, hi, r);
    quicksort(a, l, lo);
  }
}
```

the second call is a tail call and a logarithmic stack space is now guaranteed.

What if my compiler does not optimize tail calls (e.g. Java)?

No problem, do it yourself!

```
void quicksort(int a[], int l, int r) {
  while (r - l > 1) {
    ...
    if (lo - l < r - hi) {
      quicksort(a, l, lo);
      l = hi;
    } else {
      quicksort(a, hi, r);
      r = lo;
    }
  }
}
```

It is important to point out that the notion of tail call

- could be optimized in any language
  (but Java and Python do not, for instance)

- is not related to recursion
  (even if it is likely that a stack overflow is due to a recursive function)

It is not always easy to turn calls into tail calls.

Example: given a type for immutable binary trees, such as

```
type 'a tree = Empty | Node of 'a tree * 'a * 'a tree
```

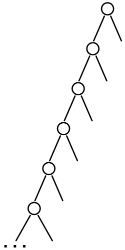
implement a function to compute the height of a tree

```
val height: 'a tree -> int
```

The natural code

```
let rec height = function
| Empty           -> 0
| Node (l, _, r) -> 1 + max (height l) (height r)
```

causes a stack overflow on a tree with a large height



Instead of computing the height h of the tree, let us compute $k(h)$ for some arbitrary function k, called a continuation

```
val height: 'a tree -> (int -> 'b) -> 'b
```

We call this continuation-passing style (or CPS).

The height of a tree is then obtained with the identity continuation

```
height t (fun h -> h)
```

The code looks like

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

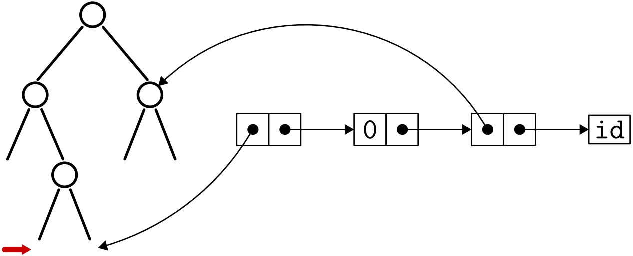
We note that all calls to `height` and `k` are tail calls.

Thus, `height` runs in constant stack space.

We have traded stack space for heap space.

Now, we have allocated a few closures.

The first closure captures `r` and `k`, the second one captures `hl` and `k`.



Note

Of course, there are other, ad hoc, solutions to compute the height of a tree without overflowing the stack (e.g. a breadth-first traversal).

Similarly, there are solutions for mutable trees, trees with parent pointers, etc.

But the CPS-based solution is systematic.

And what if the compiler optimizes tail calls but the language does not feature anonymous functions (e.g. C)?

We simply have to build closures by ourselves, manually
(a structure with a function pointer and an environment).

We can even introduce some ad-hoc data type for closures.

```
enum kind { Kid, Kleft, Kright };

struct Kont {
  enum kind kind;
  union { struct Node *r; int hl; };
  struct Kont *kont;
};
```

together with a function to apply it

```
int apply(struct Kont *k, int v) { ... }
```

this is called defunctionalization (Reynolds 1972).

Pattern-Matching

In functional languages, we often find syntactic constructions named pattern-matching used in

- function definitions

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

- generalized conditions

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

- exceptions management

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

Compiler's goal: transform such high-level constructions into sequence of elementary tests (constructor tests and constants comparison) and access to fields of structured values.

In the following, we consider the construction

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

A pattern is defined by the abstract syntax

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

where C is a constructor that can be

- a constant such as `false`, `true`, 0, 1, `"hello"`, etc.
- a constant constructor of an algebraic type, such as `[]` or, for instance, `Empty` as in `type t = Empty | ...`
- a constructor with arguments such as `::` or, for instance, `Node` as in `type t = Node of t * t | ...`
- a constructor of a n-tuple, with $n \geq 2$

Definition (linear pattern)

We say a pattern p is linear if every variable is used at most once in p.

Example: pattern (x, y) is linear, but (x, x) is not.

Note : OCaml only allows non-linear patterns in `OR` patterns.

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

```
Variable x is bound several times in this matching
```

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

In the following, we only consider linear patterns and do not consider `OR` patterns.

We now include patterns in values:

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

where C stands for the same set of constants and constructors used in the definition of patterns.

Definition (matching)

We say a value matches a pattern p if there exists a substitution σ, of variables to values, such that $v = \sigma(p)$.

Note: to make it easier, we can assume the domain of σ to be exactly that of the set of variables in p.

It is straightforward that every value matches $p = x$; on the other hand

Proposition

A value v matches $p = C(p_1, \ldots, p_n)$ if and only if v is of the form $v = C(v_1, \ldots, v_n)$ with v_i matching p_i for every $i = 1, \ldots, n$.

Definition

In the matching

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

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

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

If v does not filter any p_i, the matching leads to a runtime error (exception `Match_failure` in OCaml).

Let us consider a first algorithm for compiling pattern-matching.

We assume we have

- *constr*(e), that returns the constructor of a value e,
- $\#_i(e)$, that returns the i-th component

In other words, if $e = C(v_1, \ldots, v_n)$ then $\mathit{constr}(e) = C$ and $\#_i(e) = v_i$.

We begin by compiling a single line of pattern-matching

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

where the compilation function F is defined as follows:

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

Let us consider the example

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

Its compilation produces the following (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
```

To match several lines, we replace *error* by continuing to the next line

$$\begin{array}{l} \mathit{code}(\texttt{match}\ x\ \texttt{with}\ p_1 \rightarrow e_1 \mid \ldots \mid p_n \rightarrow e_n) = \\ \qquad F(p_1, x, e_1, F(p_2, x, e_2, \ldots F(p_n, x, e_n, \mathit{error}) \ldots)) \end{array}$$

where compilation function f is now defined as

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

The compilation of

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

produces the following code

```
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
```

This algorithm is not very efficient since

- we do several times the same tests (one line after the other)
- we do redundant tests (if $\mathit{constr}(e) \neq$ `[]` then it must be the case that $\mathit{constr}(e) = ::$)

We propose a different algorithm, that tackles the problem of multiple-lines pattern-matching as a whole.

We represent the problem as a matrix

$$\begin{vmatrix} e_1 & e_2 & \dots & e_m & & \\ p_{1,1} & p_{1,2} & \dots & p_{1,m} & \rightarrow & action_1 \\ \vdots & \vdots & \ddots & \vdots & \vdots & \vdots \\ p_{n,1} & p_{n,2} & \dots & p_{n,m} & \rightarrow & action_n \end{vmatrix}$$

whose meaning is

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

The F algorithm traverses the matrix recursively

- $n = 0$

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

- $m = 0$

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

If every column on the left hand side is made up of variables

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

we eliminate such column and introduce `let` bindings

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

Otherwise, the left column contains, for sure, at least one pattern.

Let us suppose, for instance, that in this column there are three different constructors, C with one argument, D with no arguments, and E with two arguments.

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

For each constructor C, D and E, we build the sub-matrix corresponding to the matching of a value by such a constructor.

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

where

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

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

where

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

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

where

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

We finally define a sub-matrix for every other value (different from C, D and E); in other words, for the variables.

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

At last

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

The type of expression e_1 allows to optimize the construction

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

in many cases:

- no test if there is only a constructor (*e.g.*, tuples): $F(M) = F(M_C)$
- no `otherwise` case when C, D and E are the only constructors involved
- a simple `if then else` when there are only two constructors
- a jumping table when there is a finite number of constructors
- a binary tree or a hashing-table when there is an infinite number of constructors (*e.g.*, strings)

Let us consider

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

This gives the matrix

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

We get

```
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
```

- Detecting redundant cases when some action does not occur in the produced code

  Example

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

  gives

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

- Detecting non-exhaustive matches when *error* occurs in the produced code. Example

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

  gives

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