Universidade da Beira Interior

# Desenho de Linguagens de Programação e de Compiladores

Simão Melo de Sousa

Aula 8 - Compilacão de linguagens funcionais

na aula de hoje, concentramo-nos sobre
a compilacão das **linguagens funcionais**

vamos em particular explicar

- a compilacão das funcões como valores de primeira classe
- a optimizacão das chamadas terminais
- o filtro (*pattern matching*)

# funcões como valores de primeira classe

consideramos um mini-fragmento de OCaml

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

como no mini-pascal, as funcões podem ser aninhadas

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

e o porte estático é o mesmo

mas é igualmente possível passar funcões como argumentos

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

e retornar funcões

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

em particular aplicacões parciais

```
let f x =
  let g y = x * y in g
```

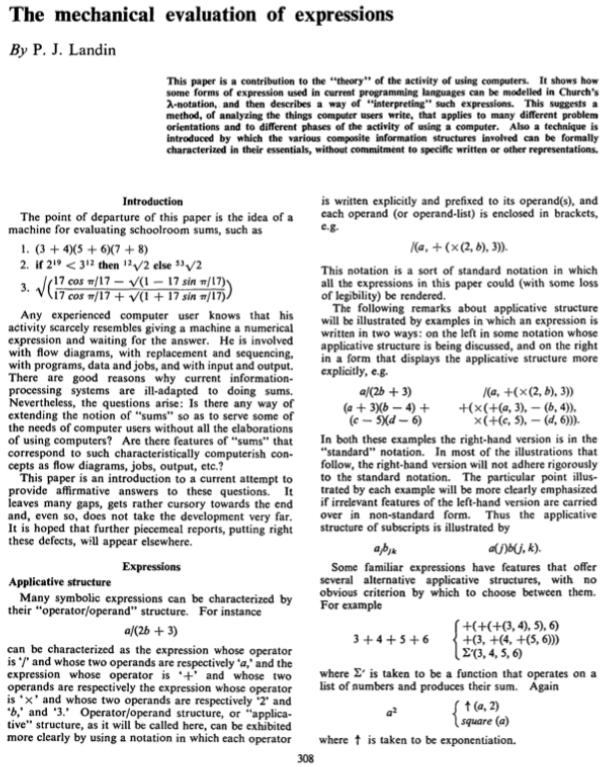
neste ultimo caso, o valor devolvido por `f` é uma funcão que utiliza `x` mas a tabela de activacão de `f` acaba precisamente de terminar !

não podemos assim compilar funcões como no caso de Pascal

a solucão consiste em utilizar um **fecho**, isto é, uma estrutura de dados alocada na *heap* (para poder sobreviver às chamadas de funcão) contendo

- um **apontador para o código** da funcão por chamar
- o valor das variáveis susceptíveis de ser utilizado por este código ; esta parte do fecho é designado de **ambiente**

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



**The mechanical evaluation of expressions**

*By* P. J. Landin

This paper is a contribution to the "theory" of the activity of using computers. It shows how some forms of expression used in current programming languages can be modelled in Church's λ-notation, and then describes a way of "interpreting" such expressions. This suggests a method, of analyzing the things computer users write, that applies to many different problem orientations and to different phases of the activity of using a computer. Also a technique is introduced by which the various composite information structures involved can be formally characterized in their essentials, without commitment to specific written or other representations.

**Introduction**

The point of departure of this paper is the idea of a machine for evaluating schoolroom sums, such as

1. $(3 + 4)(5 + 6)(7 + 8)$
2. **if** $2^{19} < 3^{12}$ **then** $\sqrt[12]{2}$ **else** $\sqrt[53]{2}$
3. $\sqrt{\left(\frac{17\ cos\ \pi/17 - \sqrt{(1 - 17\ sin\ \pi/17)}}{17\ cos\ \pi/17 + \sqrt{(1 + 17\ sin\ \pi/17)}}\right)}$

Any experienced computer user knows that his activity scarcely resembles giving a machine a numerical expression and waiting for the answer. He is involved with flow diagrams, with replacement and sequencing, with programs, data and jobs, and with input and output. There are good reasons why current information-processing systems are ill-adapted to doing sums. Nevertheless, the questions arise: Is there any way of extending the notion of "sums" so as to serve some of the needs of computer users without all the elaborations of using computers? Are there features of "sums" that correspond to such characteristically computerish concepts as flow diagrams, jobs, output, etc.?

This paper is an introduction to a current attempt to provide affirmative answers to these questions. It leaves many gaps, gets rather cursory towards the end and, even so, does not take the development very far. It is hoped that further piecemeal reports, putting right these defects, will appear elsewhere.

**Expressions**

**Applicative structure**

Many symbolic expressions can be characterized by their "operator/operand" structure. For instance

$$a/(2b + 3)$$

can be characterized as the expression whose operator is '/' and whose two operands are respectively '$a$,' and the expression whose operator is '+' and whose two operands are respectively the expression whose operator is '×' and whose two operands are respectively '2' and '$b$,' and '3.' Operator/operand structure, or "applicative" structure, as it will be called here, can be exhibited more clearly by using a notation in which each operator is written explicitly and prefixed to its operand(s), and each operand (or operand-list) is enclosed in brackets, e.g.

$$/(a, +(\times(2, b), 3)).$$

This notation is a sort of standard notation in which all the expressions in this paper could (with some loss of legibility) be rendered.

The following remarks about applicative structure will be illustrated by examples in which an expression is written in two ways: on the left in some notation whose applicative structure is being discussed, and on the right in a form that displays the applicative structure more explicitly, e.g.

$$\begin{array}{ll} a/(2b + 3) & /(a, +(\times(2, b), 3)) \\ (a + 3)(b - 4) + & +(\times(+(a, 3), -(b, 4)), \\ (c - 5)(d - 6) & \times(+(c, 5), -(d, 6))). \end{array}$$

In both these examples the right-hand version is in the "standard" notation. In most of the illustrations that follow, the right-hand version will not adhere rigorously to the standard notation. The particular point illustrated by each example will be more clearly emphasized if irrelevant features of the left-hand version are carried over in non-standard form. Thus the applicative structure of subscripts is illustrated by

$$a_j b_{jk} \qquad a(j)b(j, k).$$

Some familiar expressions have features that offer several alternative applicative structures, with no obvious criterion by which to choose between them. For example

$$3 + 4 + 5 + 6 \quad \begin{cases} +(+(+(3, 4), 5), 6) \\ +(3, +(4, +(5, 6))) \\ \Sigma'(3, 4, 5, 6) \end{cases}$$

where $\Sigma'$ is taken to be a function that operates on a list of numbers and produces their sum. Again

$$a^2 \quad \begin{cases} \uparrow(a, 2) \\ square\ (a) \end{cases}$$

where $\uparrow$ is taken to be exponentiation.

308

quais são, precisamente, as variáveis que devem constar do ambiente do fecho representando $\texttt{fun}\ x \to e$ ?

são as **variáveis livres** de $\texttt{fun}\ x \to e$

lembrete : o conjunto das variáveis livres de uma expressão calcula-se da seguinte forma

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

consideremos o programa seguinte que aproxima $\int_0^1 x^n dx$

```
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
```

facamos aparecer a construcão `fun` explicitamente e examinemos os diferentes fechos

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

- o primeiro fecho, `fun i ->`, o ambiente $\{\texttt{pow}\}$
- no segundo, `fun x ->`, vale $\{\texttt{i}, \texttt{pow}\}$

```
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
```

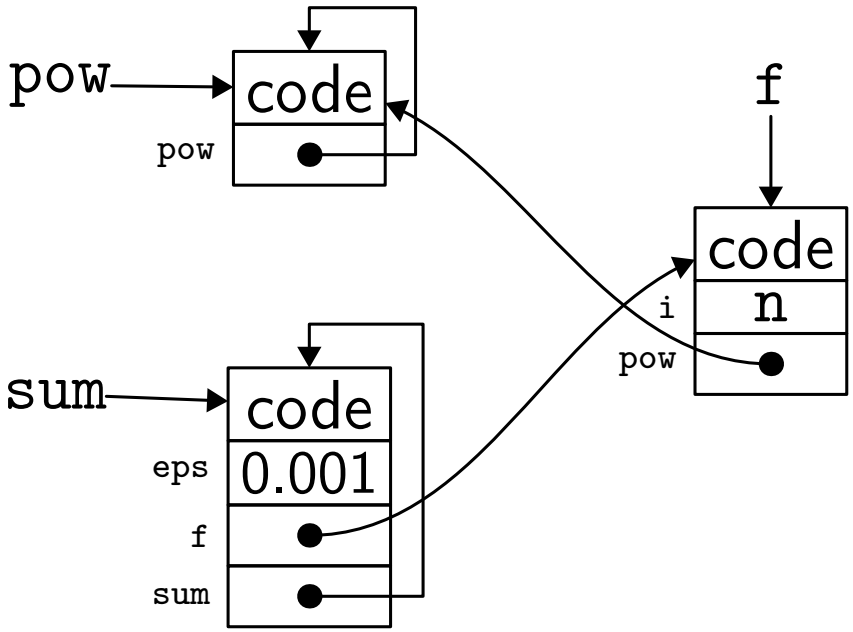
- para `fun n ->`, o ambiente vale $\{\texttt{pow}\}$
- para `fun x ->`, o ambiente vale $\{\texttt{eps}, \texttt{f}, \texttt{sum}\}$

o fecho pode ser representado da forma seguinte :

- um único bloco na *heap*, em que :
- o primeiro campo contém o endereco do código
- os campos seguintes contemplam os valores das variáveis livres, e unicamente essas

(outras solucões são possíveis : o ambiente num segundo bloco ; fechos em cadeia (em lista ligada) ; fecho contendo todas as variáveis ligadas ao ponto da criacão)

```
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
```



uma forma relativamente simples de compilar fechos consiste em proceder em dois passos

1. procuramos no código todas as construcões `fun` $x \rightarrow e$ e e substituímo-las com uma operacão explícita de construcão de fecho

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

onde os $y_i$ são as variáveis livres de `fun` $x \rightarrow e$ e $f$ o nome dado a uma declaracão global de funcão da forma

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

onde $e'$ é obtido a partir de $e$ por remocão recursiva das construcões `fun` (*closure conversion*)

2. compila-se o código obtido, que não contém mais nenhuma declaracão da forma `letfun`

no caso do exemplo anterior, isso dá

```
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]
```

antes

```
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
```

depois

```
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
```

em particular, um identificador pode representar

- `Vglobal` *s* : uma variável global (introduzida por `let`) com nome *s*
- `Vlocal` *n* : uma variável local (introduzida por `let in`), na posicão *n* na tabela de activacão
- `Vclos` *n* : uma variável contida no fecho, na posicão *n*
- `Varg` : o único argumento da funcão (o `x` de `fun x -> e`)

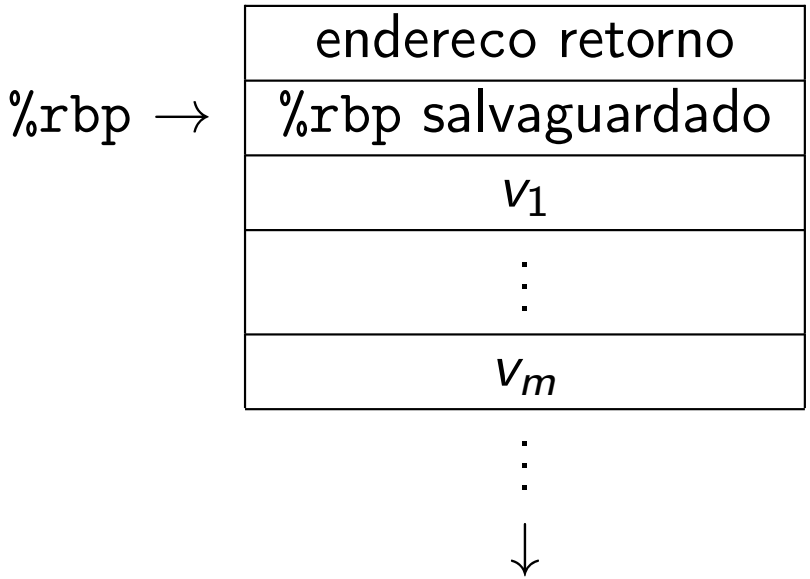
a analise de porte pode ser aplicada em conjunto com a explicitacão dos fechos

cada funcão tem um argumento único (`Varg`), que passaremos pelo registo `%rdi`

o fecho será passado via `%rsi`

a tabela de activacão assemelha-se ao seguinte, onde $v_1, \ldots, v_m$ são as variáveis locais

é integralmente construído pelo *callee*



expliquemos agora como compilar

- a construcão de um fecho $\texttt{Eclos}(f, l)$
- uma chamada de funcão $\texttt{Eapp}(e_1, e_2)$
- o acesso a uma variável $\texttt{Evar}\ x$
- uma declaracão de funcão $\texttt{Letfun}(f, e)$

para compilar a construcão

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

procedemos da seguinte forma

1. alocamos um bloco de tamanho $n + 1$ na *heap* (com `malloc`)
2. arquivamos o endereco de $f$ no campo 0
   ($f$ é uma etiqueta no código e obtemos o seu endereco com $\$f$)
3. arquivamos os valores das variáveis $y_1, \ldots, y_n$ nos campos 1 a $n$
4. retornamos o apontador para o bloco

nota : assumimos a existência do GC que tratará de libertar este bloco quando possível (o funcionamento dum GC será explicado mais adiante)

para compilar uma chamada da forma

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

procedemos da seguinte forma

1. compilamos $e_1$ no registo `%rsi`
   (o seu valor é um apontador $p_1$ para um fecho)
2. compilamos $e_2$ no registo `%rdi`
3. chamamos a funcão cujo endereco está no primeiro campo do fecho,
   com `call *(%rsi)`
   (salto para um endereco calculado, indirecto)

para compilar um acesso a uma variável

$$\texttt{Evar}\ x$$

distinguimos quarto casos, consoante o valor de $x$

- `Vglobal` $s$ : o valor encontra-se no endereco dado pela etiqueta $s$
- `Vlocal` $n$ : o valor encontra-se no endereco dado por $n$(`%rbp`)
- `Vclos` $n$ : o valor encontra-se no endereco dado por $n$(`%rsi`)
- `Varg` : o valor encontra-se em `%rdi`

finalmente, para compilar a declaracão

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

procedemos como para uma habitual declaracão de funcão

1. alocamos uma tabela de activacão que contém em particular o espaco para as variáveis locais de $e$
2. salvaguarda-se ali `%rbp` e posiciona-se `%rbp`
3. avaliamos $e$
   - em particular, procuramos o valor de $y_i$ no ambiente, cujo endereco é dado pelo primeiro argumento (`$a0`)
   - o valor de $x$ é directamente dado pelo segundo argumento (`$a1`)
4. restabelecemos `%rbp` e desalocamos a tabela de activacão
5. executamos `ret`

é dispendioso criar os fechos intermédios numa chamada onde $n$ argumentos são fornecidos

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

e onde a funcão $f$ é definida por

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

uma chamada « tradicional » poderia ser aqui feita, onde todos os argumentos são passados de uma vez

Por outro lado, uma aplicacão parcial de $f$ produziria um fecho

OCaml faz esta optimizacão ; sobre código de « primeira ordem » obtemos a mesma eficiência do que no caso de uma linguagem não funcional.

uma outra optimizacão é possível : quando sabemos que um fecho não irá sobreviver à funcão na qual foi criada, esta pode ser alocada na pilha no lugar da *heap*.

é o caso do fecho para `f` em

```
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
```

mas para assegurar que esta optimizacão é possível, é necessário efectuar uma optimizacão não trivial (*escape analysis*)

encontramos fechos noutras linguagens de programacão como, por exemplo

- Java, desde 2014 na versão 8
- C++, desde 2011 na versão C++11

nestas linguagens, as funcões anónimas são conhecidas por **lambdas**

uma funcão é um objecto como qualquer outro, dispondo de um método `apply`

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

uma funcão anónima é introduzida via o operador `->`

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

o compilador contrói um objecto **fecho** (que captura aqui a variável livre y) que contempla um método `apply`

uma funcão anónima é introduzida com a sintaxe `[]`

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

podemos especificar as variáveis capturadas no fecho (aqui `y`)

podemos especificar uma captura por referência (aqui, de `s`)

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

## optimização das chamadas terminais

## Definicão

*Dizemos que uma* ***chamada*** $(f\ e_1\ \ldots\ e_n)$ *que aparece no corpo de uma funcão g é* ***terminal*** *(tail call) se é a última coisa que g calcula antes de retornar o seu resultado.*

por extensão, podemos que uma funcão é **recursiva terminal** (*tail recursive function*) se se trata de uma funcão recursiva cujas chamadas recursivas são todas chamadas terminais

a chamada terminal não é necessariamente uma chamada recursiva

```
let g x = let y = x * x in f y
```

dentro de uma funcão recursiva podemos ter chamadas recursivas terminais e outras que não o são

```
let rec f91 n =
  if n <= 100 then f91 (f91 (n + 11)) else n - 10
```

que interesse do ponto de vista da compilacão ?

podemos destruir a tabela de activacão da funcão onde se encontra a chamada **antes** de fazer a chamada, visto que não ser mais utilizada

melhor, podemos reutilizá-la para a chamada terminal que devemos fazer (em particular, o endereco de retorno aí arquivado continua a ser o endereco certo)

dito de outra forma, podemos fazer o salto (`jump`) em vez de uma chamada (`call`)

consideremos

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

uma compilacão clássica dá

```
fact:
        movq    $1, %rdx
        cmpq    %rdx, %rsi
        jle     L0              # n <= 1 ?
        imulq   %rsi, %rdi      # acc <- acc * n
        decq    %rsi            # n <- n - 1
        call    fact
        ret
L0:     movq    %rdi, %rax
        ret
```

optimizando a chamada terminal, obtemos

```
fact:   movq    $1, %rdx
        cmpq    %rdx, %rsi
        jle     L0              # n <= 1 ?
        imulq   %rsi, %rdi      # acc <- acc * n
        decq    %rsi            # n <- n - 1
        jmp     fact
L0:     movq    %rdi, %rax
        ret
```

o resultado é assim um **ciclo**

o código é de facto idêntico ao código obtido por uma compilacão de um programa C de forma

```
while (n > 1) {
  acc = acc * n;
  n   = n - 1
}
```

e isto, sem que hajam construcões imperativas na linguagem considerada !

o programa obtido é mais eficiente

em particular porque acedemos menos à memória (não se utiliza mais nem **call** nem **ret**)

**o espaco utilizado na pilha torna-se constante**

em particular, evita-se os *stack overflow* que resultariam de um número demasiado grande de chamadas recursivas aninhadas.

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

```
Fatal error: exception Stack_overflow
```

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

```
Segmentation fault
```

etc.

é importante notar que a nocão de chamada terminal

- não é específica das linguagens funcionais (não lhe são próprias)
  $\Rightarrow$ a sua compilacão pode ser optimizada no contexto de qualquer linguagem
  (por exemplo, `gcc -O2` realiza tal optimizacão)

- não está ligada a recursividade
  (mesmo se é verdade que os casos de *stack overflow* são muitas vezes causados por funcões recursivas)

**exercício** dado o seguinte tipo das árvores binárias

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

escrever uma funcão que calcula a altura de uma árvore

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

**requisito implícito**: deve funcionar para qualquer árvore

o código natural

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

provoca um erro de transbordamento de pilha (em inglês, **stack overflow**) quando aplicada a árvores de grande tamanho

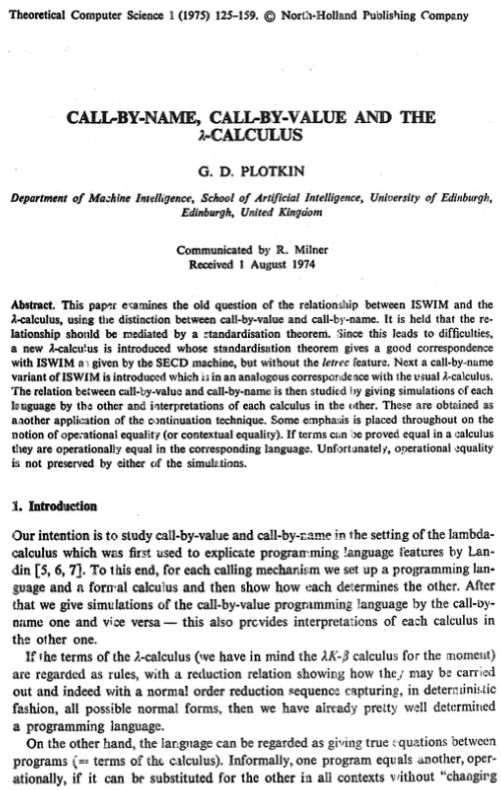
em vez de calcular a altura *a* de uma árvores, calculemos $a(k)$ para uma função *k* qualquer, designada de **continuação**
esta continução captura a noção de: ***o que fazer a seguir***

```
val height = 'a treee -> (int -> 'b) -> 'b
```

este estilo de programação é conhecido por

**programação por continuações** (*continuation passing style*, em inglês)

o programa visado deduz-se com a continuação *identidade*, isto é,
`height t (fun a -> a)`



Theoretical Computer Science 1 (1975) 125–159. © North-Holland Publishing Company

CALL-BY-NAME, CALL-BY-VALUE AND THE λ-CALCULUS

G. D. PLOTKIN

*Department of Machine Intelligence, School of Artificial Intelligence, University of Edinburgh, Edinburgh, United Kingdom*

Communicated by R. Milner
Received 1 August 1974

**Abstract.** This paper examines the old question of the relationship between ISWIM and the λ-calculus, using the distinction between call-by-value and call-by-name. It is held that the relationship should be mediated by a standardisation theorem. Since this leads to difficulties, a new λ-calculus is introduced whose standardisation theorem gives a good correspondence with ISWIM as given by the SECD machine, but without the *letrec* feature. Next a call-by-name variant of ISWIM is introduced which is in an analogous correspondence with the usual λ-calculus. The relation between call-by-value and call-by-name is then studied by giving simulations of each language by the other and interpretations of each calculus in the other. These are obtained as another application of the continuation technique. Some emphasis is placed throughout on the notion of operational equality (or contextual equality). If terms can be proved equal in a calculus they are operationally equal in the corresponding language. Unfortunately, operational equality is not preserved by either of the simulations.

**1. Introduction**

Our intention is to study call-by-value and call-by-name in the setting of the lambda-calculus which was first used to explicate programming language features by Landin [5, 6, 7]. To this end, for each calling mechanism we set up a programming language and a formal calculus and then show how each determines the other. After that we give simulations of the call-by-value programming language by the call-by-name one and vice versa — this also provides interpretations of each calculus in the other one.

If the terms of the λ-calculus (we have in mind the λK-β calculus for the moment) are regarded as rules, with a reduction relation showing how they may be carried out and indeed with a normal order reduction sequence capturing, in deterministic fashion, all possible normal forms, then we have already pretty well determined a programming language.

On the other hand, the language can be regarded as giving true equations between programs (= terms of the calculus). Informally, one program equals another, operationally, if it can be substituted for the other in all contexts without "changing

o código tem então a seguinte forma

```
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 hk)))
```

constatamos que todas as chamadas a `height` e `k` são chamadas **terminais**

o cálculo de `height` faz-se en consequência usando espaço em pilha constante!

substituimos o espaço na pilha por espaço **na heap**

este é utilizado pelos fechos (das continuações)

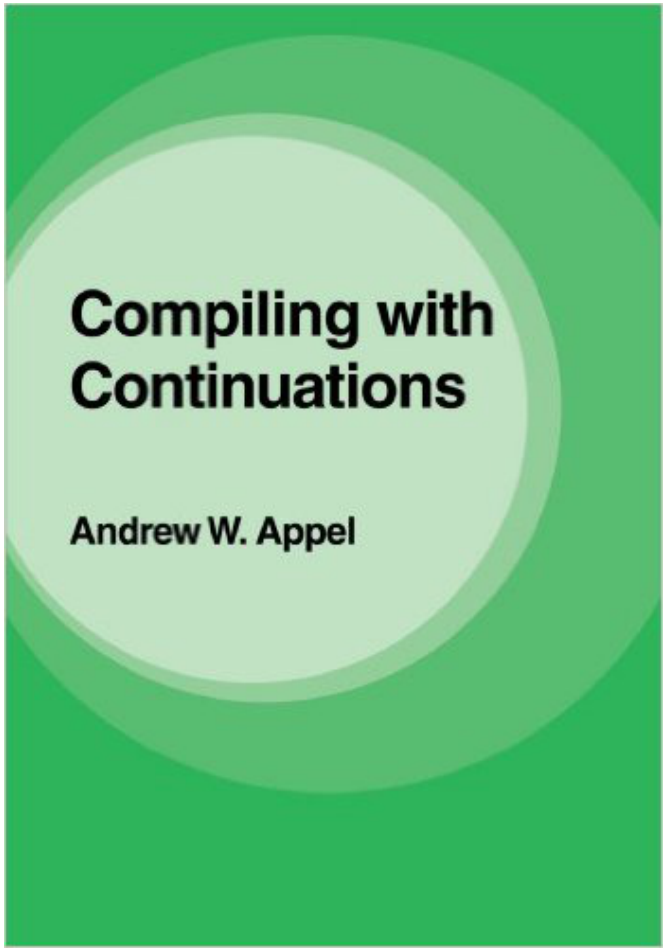
o primeiro fecho integra `r` e `k`, o segundo `hl` e `k`

obviamente, há outras soluções, ad-hoc, para calcular a altura de uma árvore sem que estas provocam *stack overflow*

por exemplo na base de um percurso em profundidade

existam igualmente outras soluções se o tipo das árvores é mais complexo (árvores mutáveis, altura arquivada nos nodos, apontadores para os nodos pais, etc.)

mas a solução baseada no CPS tem o mérito de **ser mecânica**



e se a linguagem optimiza as chamadas recursivas mas não dispõe do mecanismo de funções anónimas (como a linguagem C, por exemplo)

basta construir explícitamente "*à mão*" os fechos

podemos até fazê-lo com base num tipo ad-hoc

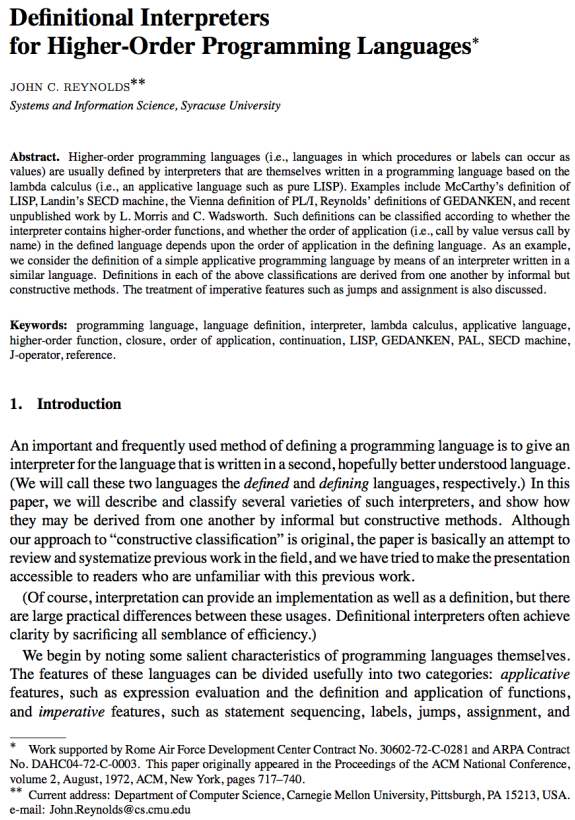
```
enum kind { Kid, Kleft, Kright };

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

e uma função para a aplicar

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

está técnica tem por nome **defoncionalização**
(Reynolds, 1972 - reeditado em 1998)



**Definitional Interpreters for Higher-Order Programming Languages***

JOHN C. REYNOLDS**
*Systems and Information Science, Syracuse University*

**Abstract.** Higher-order programming languages (i.e., languages in which procedures or labels can occur as values) are usually defined by interpreters that are themselves written in a programming language based on the lambda calculus (i.e., an applicative language such as pure LISP). Examples include McCarthy's definition of LISP, Landin's SECD machine, the Vienna definition of PL/I, Reynolds' definitions of GEDANKEN, and recent unpublished work by L. Morris and C. Wadsworth. Such definitions can be classified according to whether the interpreter contains higher-order functions, and whether the order of application (i.e., call by value versus call by name) in the defined language depends upon the order of application in the defining language. As an example, we consider the definition of a simple applicative programming language by means of an interpreter written in a similar language. Definitions in each of the above classifications are derived from one another by informal but constructive methods. The treatment of imperative features such as jumps and assignment is also discussed.

**Keywords:** programming language, language definition, interpreter, lambda calculus, applicative language, higher-order function, closure, order of application, continuation, LISP, GEDANKEN, PAL, SECD machine, J-operator, reference.

**1. Introduction**

An important and frequently used method of defining a programming language is to give an interpreter for the language that is written in a second, hopefully better understood language. (We will call these two languages the *defined* and *defining* languages, respectively.) In this paper, we will describe and classify several varieties of such interpreters, and show how they may be derived from one another by informal but constructive methods. Although our approach to "constructive classification" is original, the paper is basically an attempt to review and systematize previous work in the field, and we have tried to make the presentation accessible to readers who are unfamiliar with this previous work.

(Of course, interpretation can provide an implementation as well as a definition, but there are large practical differences between these usages. Definitional interpreters often achieve clarity by sacrificing all semblance of efficiency.)

We begin by noting some salient characteristics of programming languages themselves. The features of these languages can be divided usefully into two categories: *applicative* features, such as expression evaluation and the definition and application of functions, and *imperative* features, such as statement sequencing, labels, jumps, assignment, and

* Work supported by Rome Air Force Development Center Contract No. 30602-72-C-0281 and ARPA Contract No. DAHC04-72-C-0003. This paper originally appeared in the Proceedings of the ACM National Conference, volume 2, August, 1972, ACM, New York, pages 717–740.
** Current address: Department of Computer Science, Carnegie Mellon University, Pittsburgh, PA 15213, USA. e-mail: John.Reynolds@cs.cmu.edu

# filtro

nas linguagens funcionais encontramos em geral uma construcão chamada **filtro** (ou ainda concordância de padrão, ou *pattern-matching* em inglês), utilizada nas

- definicões de funcões

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

- nas condicionais generalizadas

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

- nos gestores de excepcões

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

objectivo do compilador : transformar estas construcões de alto nível para sequências de **testes elementares** (testes de constructores e comparacões de valores constantes) e acesso a campos de valores estruturados

no que se segue, consideramos a construcão

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

(para a qual é fácil chegar, com a ajuda de um `let`)

um **padrão** (*pattern*) é definido pela sintaxe abstracta

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

onde $C$ é um **constructor**, que pode ser

- uma constante tal que `false`, `true`, 0, 1, `"hello"`, etc.
- um constructor constante de tipo soma, tal que `[]` ou por exemplo `Empty` declarado por `type t = Empty | ...`
- um constructor de aridade $n \geq 1$ tal que `::` ou por exemplo `Node` declarado por `type t = Node of t * t | ...`
- um constructor de $n$-tuplos, com $n \geq 2$

## Definicão (padrão linear)

*Dizemos que um padrão p é **linear** se toda variável aparece no máximo uma vez em p.*

exemplo : o padrão $(x, y)$ é linear, mas $(x, x)$ não o é

nota : OCaml só admite padrões não lineares nos padrões OU

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

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

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

na sequência desta exposicão, consideraremos somente padrões lineares (e assim, não são considerados padrões OU)

os valores aqui considerados são aqui

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

onde $C$ designa o mesmo conjunto de constantes e de construtores que os que são considerados na definicão dos padrões

## Definicão (filtro)

*Dizemos que um valor v* ***filtra*** *um padrão p se existe uma substituicão $\sigma$ de variáveis por valores tais que $v = \sigma(p)$.*

nota : podemos supor igualmente que o domínio de $\sigma$, isto é o conjunto das variáveis $x$ tais que $\sigma(x) \neq x$, está incluído no conjunto das variáveis de $p$

é óbvio que todo o valor filtra $p = x$ (basta considerar $\sigma$ tal que $\sigma(x) = v$) ; mais,

## Proposicão

*Um valor $v$ filtra $p = C(p_1, \ldots, p_n)$ se e só se $v$ é da forma $v = C(v_1, \ldots, v_n)$ com $v_i$ que filtra $p_i$ para todo $i = 1, \ldots, n$.*

demonstracão :

- seja $v$ que filtra $p$ ; temos então $v = \sigma(p)$ para um certo $\sigma$, seja $v = C(\sigma(p_1), \ldots, \sigma(p_n))$ e definimos então $v_i = \sigma(p_i)$
- de forma recíproca, se $v_i$ filtra $p_i$ para todo $i$, então existe $\sigma_i$ tais que $v_i = \sigma_i(p_i)$ ; como $p$ é linear, os domínios dos $\sigma_i$ disjuntos entre si e temos então $\sigma_i(p_j) = p_j$ se $i \neq j$

  definindo $\sigma = \sigma_1 \circ \cdots \circ \sigma_n$, temos $\sigma(p_i) = \sigma_1(\sigma_2(\ldots \sigma_n(p_i)) \ldots)$ logo
  $= \sigma_1(\sigma_2(\ldots \sigma_i(p_i)) \ldots)$
  $= \sigma_1(\sigma_2(\ldots v_i) \ldots)$
  $= v_i$

  $\sigma(p) = C(\sigma(p_1), \ldots, \sigma(p_n)) = C(v_1, \ldots, v_n) = v$ □

## Definicão

*No filtro*

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

*se $v$ é o valor de $x$, dizemos que $v$ filtra o caso $p_i$ se $v$ filtra $p_i$ e se $v$ não filtra $p_j$ para todo o $j < i$.*

*O resultado do filtro é então $\sigma(e_i)$, onde $\sigma$ é a substituicão tal que $\sigma(p_i) = v$.*

se $v$ não filtra nenhum $p_i$, o filtro resulta num erro de execucão (excepcão `Match_failure` em OCaml)

consideremos um primeiro algoritmo de compilacão do filtro

assumimos que dispomos de

- $\mathit{constr}(e)$, que retorna o constructor do valor $e$,
- $\#_i(e)$, que retorna a sua $i$-ésima componente

ou seja, se $e = C(v_1, \ldots, v_n)$ então $\mathit{constr}(e) = C$ e $\#_i(e) = v_i$

nota : já vimos como os valores de OCaml estavam representados, deduz-se daí como implementar estas funcões

comecamos pela compilacão de uma linha de filtro

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

onde a funcão de compilacão $F$ está definida da seguinte forma :

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

consideremos por exemplo

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

a sua compilacão devolve o (pseudo-)código seguinte :

```
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
```

nota : $\#_2$(x) é calculada várias vezes $\Rightarrow$ poderíamos introduzir os `let` necessários na definicão de $F$ para remediar esta situacão

mostremos que se $e \xrightarrow{\star} v$ então

$$F(p, e, action) \xrightarrow{\star} \sigma(action) \qquad \text{se existe } \sigma \text{ tal que } v = \sigma(p)$$
$$F(p, e, action) \xrightarrow{\star} error \qquad \text{senão}$$

demonstracão : por recorrência sobre $p$

- $p = x$ ou $p = C$ : trivial.
- $p = C(p_1, \ldots, p_n)$ :
  - se $constr(v) \neq C$, não existe $\sigma$ tal que $v = \sigma(p)$ e $F(C(p_1, ..., p_n), e, action) = error$
  - se $constr(v) = C$, temos $v = C(v_1, \ldots, v_n)$ ; $\sigma$ tal que $v = \sigma(p)$ existe se e só se existem $\sigma_i$ tais que $v_i = \sigma_i(p_i)$

    se um dos $\sigma_i$ não existe, então a chamada $F(p_i, \#_i(e), \ldots)$ se reduz para $error$ e $F(p, e, action)$ igualmente

    se todos os $\sigma_i$ existem, então por hipótese de indução

$$\begin{aligned} F(p, e, action) &= F(p_1, \#_1(e), F(p_2, \#_2(e), \ldots F(p_n, \#_n(e), action) \ldots) \\ &= F(p_1, \#_1(e), F(p_2, \#_2(e), \ldots \sigma_n(action)) \ldots) \\ &= \sigma_1(\sigma_2(\ldots \sigma_n(action) \ldots)) = \sigma(action) \end{aligned}$$

□

para filtrar várias linhas, substituímos *error* pela passagem à linha seguinte

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

onde a funcão de compilacão $F$ é agora definida por :

$$\begin{aligned} &F(x, e, successo, falha) = \\ &\quad \texttt{let } x = e \texttt{ in } successo \\ &F(C, e, successo, falha) = \\ &\quad \texttt{if } constr(e) = C \texttt{ then } successo \texttt{ else } falha \\ &F(C(p_1, \ldots, p_n), e, successo, falha) = \\ &\quad \texttt{if } constr(e) = C \texttt{ then} \\ &\qquad F(p_1, \#_1(e), F(p_2, \#_2(e), \ldots F(p_n, \#_n(e), successo, falha)\ldots, falha) \\ &\quad \texttt{else } falha \end{aligned}$$

a compilacão de

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

dá o código seguinte

```
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
```

este algoritmo é pouco eficaz porque

- efectuamos várias vezes os mesmos testes (de uma linha para a outra)
- efectuamos testes redundantes (se $constr(e) \neq []$ então necessariamente $constr(e) = ::$)

consideramos outro algoritmo que considera o problema do filtro de $n$ linhas na sua globalidade

representamos o problema na forma de uma **matriz**

$$\left|\begin{array}{lllll} e_1 & e_2 & \ldots & e_m & \\ p_{1,1} & p_{1,2} & \ldots & p_{1,m} & \rightarrow \; action_1 \\ \vdots & \vdots & \ddots & \vdots & \vdots \\ p_{n,1} & p_{n,2} & \ldots & p_{n,m} & \rightarrow \; action_n \end{array}\right|$$

cujo significado é

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

o algoritmo $F$ actua recursivamente sobre a matriz

- $n = 0$

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

- $m = 0$

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

se toda a coluna esquerda é composta de **variáveis**

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

eliminamos esta coluna pela introducão dos `let`

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

senão, é porque a coluna esquerda contém pelo menos um padrão construído

supomos, por exemplo, que haja nesta coluna três construtores diferentes, $C$ de aridade 1, $D$ de aridade 0 e $E$ de aridade 2

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

para cada construtor $C$, $D$ e $E$, construímos a sub-matriz que corresponde ao filtro de um valor para este construtor

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

donde

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

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

donde

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

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

done

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

finalmente definimos uma sub-matriz para os outro valores ( de construtores diferentes de $C$, $D$ e $E$), isto é, para as variáveis

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

definimos então

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

este algoritmo termina

de facto, a medida

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

é natural ($\in \mathbb{N}$) e diminuí estritamente a cada chamada recursiva de $F$, se definimos

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

a correcão deste algoritmo é deixada em exercício

indicacão : utilizar a interpretacão da matriz como

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

o tipo das expressões $e_1$ permite optimizar a construcão

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

em numerosos casos :

- sem teste se um só construtor (por ex. $n$-tupolo) : $F(M) = F(M_C)$
- sem caso `otherwise` quando $C$, $D$ e $E$ são os únicos construtores
- um simples `if then else` quando só há dois construtores
- uma tabela de salto quando a um número finito de construtor
- uma árvore binária ou uma tabela de dispersão quando há uma infinidade de construtores (no caso, por exemplo, das strings)

consideremos

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

isto é, a matriz

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

obtemos

```
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
```

- detecção dos **caso redundantes**
  quando uma acção não aparece no código produzido

  exemplo

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

  dá

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

- detecção dos **filtros não exaustivos**
  quando *error* aparece no código produzido

  exemplo

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

  dá

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

# conclusão

- práticas: continuação da prática anterior
- próxima aula teórica
  - compilacão das linguagens orientadas a objecto

estes acetatos resultam essencialmente de uma adaptação do material pedagógico gentilmente cedido pelo Jean-Christophe Filliâtre (link1, link2)

adicionalmente poderá consultar as obras seguintes

- **Modern Compilers: Principles, Techniques, and Tools**, Alfred V. Aho, Monica S. Lam, Ravi Sethi et Jeffrey D. Ullman
- **Types and Programming Languages**, Benjamin C. Pierce
- **Modern Compiler Implementation**, Andrew W. Appel (3 versões: ML, C, Java)