INF05516 - Semântica formal N
Ciência da Computação - UFRGS
2006-2

Marcus Ritt
mrpritt@inf.ufrgs.br

28/08/2006

Agenda

Última aula:

- Semântica operacional estrutural
- Características da semântica operacional estrutural
- Estudo de caso: Paralelismo

Hoje:

- Introdução em OCaml

OCaml - vista geral

OCaml

- é uma linguagem funcional (ou aplicativa) e um dialeto de ML ("meta-linguagem");
- é tipada estáticamente e tem inferência de tipos;
- tem elementos imperativos;
- tem elementos da orientação a objetos;
- tem uma semântica formal [1];
- é software livre: `http://caml.inria.fr` tem versões para Linux, MacOs X e Windows.

OCaml: Interpredador

- O interpredador `ocaml` permite uma interação direta.
- Ele mostra o prompt # e entra na interação com o usuário.
- Ocaml aceita expressões terminadas com `;;`

```
# 2+3;;
- : int = 5
```

- A diretiva `#quit` fecha a sessão.

```
# #quit;;
```

- Se a avaliação não termina, `Ctrl-C` interrompe.
- Comentários tem a forma `(* ... *)`.

OCaml: Compilador

- `ocamlc` compila código fonte para código byte.
- `ocamlrun` é o interpredador do código byte.

```
ocamlc IMP.ml -o IMP
```

Tipos básicos

Tipo	Função	Literais
`bool`	Valores booleanas	`true`,`false`
`int`	Números inteiros (31/63 bits+sinal)	-34
`float`	Números de ponto flutuante (IEEE 754 precisão dupla, 64 bits)	`-0.34e2`
`char`	caracteres (8 bit)	`'a'`, `'\''`
`string`	Cadeias	`"xurumbambo"`
`unit`	Tipo com um valor único	`()`

- _ permite escrever números mais legíveis: 1_048_576.

Expressões

- As expressões são parecidas com outras linguagens: 2+3, 5*(3/4)
- Os operadores básicos de ponto flutuante são `+.`, `-.`, `*.` e `/.`

```
# 2. /. 3. ;;
- : float = 0.66666666666666663
```

- O operador ^ junta cadeias

```
# "casa" ^ "mata" ;;
- : string = "casamata"
```

Definições globais

- `let <nome> = <expressão>` defina nomes globais para qualquer valor

```
# let x=5;;
val x : int = 5
# let y="casamata";;
val y : string = "casamata"
# x;;
- : int = 5
# y;;
- : string = "casamata"
```


Definições locais

- `let <nome> = <expressão1> in <expressão2>` defina nomes locais para a avaliação da expressão2

```
# let z = 5 in 2+z;;
- : int = 7
# z;;
Unbound value z
```

Funções

- Funções simplesmente são valores do tipo função
- `function ... -> ...` constrói as literais (funções)

```
# function x -> 2*x;;
- : int -> int = <fun>
```

- A aplicação é a juxtaposição da função com o argumento

```
# (function x->2*x) 4;;
- : int = 8
```

- Usando `let` podemos dar nomes para funções

```
# let twice = function x -> 2*x;;
val twice : int -> int = <fun>
# twice 4;;
- : int = 8
```


Tipos avançados 13 / 29

Tuplas

- Tuplas (ou tipos produtos) combinam um número fixo de outros tipos
- O construtor é `,`

```
# 2,3;;
- : int * int = (2, 3)
# 2,"casa";;
- : int * string = (2, "casa")
```

- Para pares, `fst` e `snd` extraiem o primeiro o segundo componente

```
# fst (2,"casa");;
- : int = 2
# snd (2,"casa");;
- : string = "casa"
```

Listas

- Listas contém uma serie de valores do mesmo tipo
- Os construtores são `::` (prefixo) e `[]` (lista vazia)

```
# 2::3::4::[];;
- : int list = [2; 3; 4]
```

- `[...;...]` é uma abreviatura

```
# let l = [2;3;4];;
- : int list = [2; 3; 4]
```

- O módulo `List` contém varias funções como, por exemplo, `List.hd` (primeiro elemento) e `List.tl` (resto da lista)

```
# List.hd l;;
- : int = 2
# List.tl l;;
- : int list = [3; 4]
```


Variantes

- Podemos definir novos tipos com `type`.
- Uma *variante* (tipo soma, inglês: variant) permite valores diferentes; cada valor tem seu *construtor*:

```
type sinal = Vermelho | Amarelo | Verde;;
# Vermelho;;
- : sinal = Vermelho
```

- `of` defina construtores com argumentos:

```
# type intlist =
    Empty | Cons of int * intlist;;
type intlist = Empty | Cons of int * intlist
# Cons(2,Cons(3,Empty));;
- : intlist = Cons (2, Cons (3, Empty))
```

- O nome de um construtor tem que começar com uma letra maiúscula

Variantes...

- Os argumentos de uma função, em particular no caso das variantes podem ser analisados usando casamento de padrões (inglês: pattern matching)
- O padrão para variantes é simplesmente o nome do construtor (e argumentos, se tem)
- Um casamento de padrões tem a forma `match <expressão> with p1 -> e1 | p2 -> e2 | ...`

```
let rec soma = function list ->
    match list with
      Empty -> 0
    | Cons(n,l) -> n+(soma l)
;;
# soma (Cons(4,Cons(5,Cons(6,Empty))));;
- : int = 15
```


Conceitos avançados 18 / 29

Definições recursivas

Usando `let rec` uma definição pode ser recursiva também

```
# let rec um_infinito = 1::um_infinito;;
val um_infinito : int list =
  [1; 1; 1; 1; 1; 1; 1; 1; 1; 1; 1; ...]
# let rec fat =
    function x -> if x = 0 then 1 else x*fat(x-1);;
val fat : int -> int = <fun>
# fat 5;;
- : int = 120
# fat (-1);;
Stack overflow during evaluation
  (looping recursion?).
```

Tratamento de erros

- Ocaml tem exeções. `exception` permite definir novas exeções. Eles tem o tipo `exn`.

```
# exception Argument_out_of_range;;
exception Argument_out_of_range
# Argument_out_of_range;;
- : exn = Argument_out_of_range
```

- Exeções são jogadas com `raise`

```
# let rec fat = function x ->
    if x < 0
      then raise Argument_out_of_range
      else begin
        if x = 0 then 1 else x*fat(x-1)
      end;;
val fat : int -> int = <fun>
# fat (-1);;
Exception: Argument_out_of_range.
```


Tratamento de erros...

- Usando `try ... with` podemos apanhar exeções

```
# try fat(-1) with Argument_out_of_range -> 0;;
- : int = 0
```

Sintaxe: Expressões aritméticas

```
(* categoria sintatica Num *)
type num = int;;
(* categoria sintatica Ident *)
type ident = string;;
(* categoria sintatica Bool: usamos bool *)

(* categoria sintatica AExpr *)
type aExpr =   Num of num | Ident of ident
             | Sum of aExpr * aExpr
             | Diff of aExpr * aExpr
             | Prod of aExpr * aExpr;;
```


Sintaxe: Expressões booleanas

```
(* categoria sintatica BExpr *)
type bExpr =   Bool of bool
             | Eq of aExpr * aExpr
             | Le of aExpr * aExpr
             | Not of bExpr
             | And of bExpr * bExpr
             | Or of bExpr * bExpr;;
```

Sintaxe: Comandos

```
(* categoria sintatica Com *)
type com =   Skip
           | Assign of loc * aExpr
           | Seq of com * com
           | Cond of bExpr * com * com
           | While of bExpr * com;;
```


Estado

```
(* o estado: uma função das identificadores
   para números inteiros *)
type sigma = ident -> num;;

(* o estado inicial *)
let empty : sigma = function x -> 0;;

(* um estado modificado em um ponto:
   sigma[l -> n] *)
let mapsto f l n a = if a = l then n else f a;;
```

Avaliação de expressões

```
(* avaliação das expressões aritméticas *)
let rec eval_aExpr (a,sigma) =
  match a with
    Num n -> n
  | Ident x -> sigma x
  | Sum (a1,a2) ->
      (eval_aExpr (a1,sigma))+
      (eval_aExpr (a2,sigma))
  | Diff (a1,a2) ->
      (eval_aExpr (a1,sigma))-
      (eval_aExpr (a2,sigma))
  | Prod (a1,a2) ->
      (eval_aExpr (a1,sigma))*
      (eval_aExpr (a2,sigma))
;;
```


Avaliação de expressões

```
# eval_aExpr(Num 5,empty);;
- : num = 5
# eval_aExpr(Ident "x", empty);;
- : num = 0
# eval_aExpr(Ident "x", mapsto empty "x" 3);;
- : num = 3
```

Referências

[1] Didier Rémy. Using, understanding, and unraveling the OCaml language. In G. Barthe, P. Dybjer, L. Pinto, and J. Saraiva, editors, *Applied Semantics*, number 2395 in Lecture Notes in Computer Science, pages 413–536. Springer, 2002. INF: 681.32.06(063) I61as.