Basic Notions II. Functions

Adrian Groza

Department of Computer Science
Technical University of Cluj-Napoca
Recall...

Basic Types
- integer, real, boolean, char, string, enumeration.

Recursion Rules
1. Know when to stop
2. Decide how to take one step
3. Break the journey down into that step plus a smaller journey
Outline

1. Expression Evaluation
   - Call by value
   - Call-by-name
   - Call-by-need

2. Tuples

3. Functions and Operators
   - Infix Operators
   - Local Declarations

4. Food for Thought
Parameter evaluation

Example

fun sq(z):int = z * z;
▷ val sq = fn: int -> int

Example

fun zero(x:int) = 0;
▷ val zero = fn: int -> int

Question

When calling a function, when are the actual parameters evaluated: before or after they replace the formal parameters?
**Call-by-value**

**Definition**

In case of "call-by-value", the **value** of the parameter is sent.

**Example (fun sq(z):int = z * z;)**

- `sq(sq(sq(2))) =
- `sq(sq(2 * 2)) =
- `sq(sq(4)) =
- `sq(4 * 4) =
- `sq(16) =
- `16 * 16 =
- `256

**Example (fun zero(x:int) = 0;)**

- `zero(sq(sq(sq(2)))) =
- `zero(sq(sq(2 * 2))) =
- `zero(sq(sq(4))) =
- `zero(sq(4 * 4)) =
- `zero(sq(16)) =
- `zero(16 * 16) =
- `zero(256) = 0

Extra computations may occur.
Call by value

Parameter evaluation

Example

fun fact(n) = if n = 0 then 1 else n * fact(n-1);
▷ val fact = fn: int -> int

Example

fun facti(n,p) = if n = 0 then p else facti(n-1,n*p);
▷ val facti = fn: int*int -> int
Parameter evaluation

\textbf{fun} \ fact(n) = if n = 0 then 1 else n * fact(n-1);

\begin{align*}
\text{fact}(4) & = 4 \ast \text{fact}(4-1) \\
& = 4 \ast \text{fact}(3) \\
& = 4 \ast (3 \ast \text{fact}(3-1)) \\
& = 4 \ast (3 \ast \text{fact}(2)) \\
& = 4 \ast (3 \ast (2 \ast \text{fact}(2-1))) \\
& = 4 \ast (3 \ast (2 \ast \text{fact}(1))) \\
& = 4 \ast (3 \ast (2 \ast (1 \ast \text{fact}(1-1)))) \\
& = 4 \ast (3 \ast (2 \ast (1 \ast \text{fact}(0)))) \\
& = 4 \ast (3 \ast (2 \ast (1 \ast 1))) \\
& = 4 \ast (3 \ast (2 \ast 1)) \\
& = 4 \ast (3 \ast 2) \\
& = 4 \ast 6 \\
& = 24
\end{align*}
Call by value

**Parameter evaluation**

fun facti(n, p) = if n = 0 then p else facti(n-1,n*p);

fact(4,1) = fact(4-1,4*1)
= fact(3,4)
= fact(3-1,3*4)
= fact(2,12)
= fact(2-1,2*12)
= fact(1,24)
= fact(1-1,1*24)
= fact(0,24)
= 24

No extra memory is needed.
Call-by-value and the Conditionals

if E then ET else EF

1. E gets evaluated, then ...
2. either ET or EF is evaluated, but **never** both of them

▷ if 1 == 0 then *I am the Pope* else *someone else is the Pope*
▷ if if 1 < 2 then 3 < 2 else 4 < 5 then ’a’ else ’b’
▷ 1 + if True then 1 else 2

▷ *fun cond(p,x,y):int=if p then x else y;*
  *val cond=fn:bool*int*int-*int*

*fun facter n = cond(n=0,1,n*facter(n-1));*
  *facter(0) ⇔ cond(true,1,0*facter(-1))*
  *⇔ cond(true,1,0*(cond(false,1,-1*facter(-2))))*

⚠️ In case of call by value there is no function *cond* such that *cond(E, E₁, E₂)* to be evaluated as a conditional.
Call-by-value and the Conditionals

andalso A andalso B ⇔ if A then B else false
orelse A orelse B ⇔ if A then true else B

▷ fun even n = (n mod 2 = 0);
val even = fn: int-> bool
▷ fun ev n = (n = 1) orelse (even(n) andalso ev (n div 2));
val ev = fn: int-> bool

ev(6) = (6=1) orelse (even(6) andalso (ev (6 div 2))
  = even(6) andalso ev(6 div 2)
  = ev(3)
  = even(3) andalso ev(3 div 2)
  = false
Call-by-name

Definition
The expression of the parameter is sent.

Example (fun zero(z:int) = 0;)
zero(sq(sq(sq(2)))) = 0

Example (fun sq(z):int = z * z;)

\[
\begin{align*}
sq(sq(sq(2))) &= \\
&= sq(sq(2)) \times sq(sq(2)) \\
&= (sq(2) \times sq(2)) \times sq(sq(2)) \\
&= ((2 \times 2) \times sq(2)) \times sq(sq(2)) \\
&= (4 \times sq(2)) \times sq(sq(2)) \\
&= (4 \times (2 \times 2)) \times (sq(sq(2)) = \\
&= ... \\
\end{align*}
\]

Disadvantages when the same argument appears more than once in function's body.
Call-by-need

**Definition**

Call-by-need ("lazy evaluation") is similar to call-by-name, but evaluates an expression **only once, when necessary**.

**Notation:** We write \([x=E]\) in order to say that all occurrences of \(x\) in the function’s body have access to \(E\).

**Example (fun sq(z):int=z*z)**

\[
\begin{align*}
sq(sq(sq(2))) & = z*z & [z=sq(sq(2))] \\
& = z*z & [z=y*y][y=sq(2)] \\
& = z*z & [z=y*y][y=2*2] \\
& = z*z & [z=4*4] \\
& = 16 * 16 \\
& = 256
\end{align*}
\]
Call-by-need

Advantage: no needed computations are eliminated.
\[ f \ n = (2 \ast n, g \ n) \]
\[ g = \text{a difficult-to-compute function} \]
\[ h(x,y) = x \]
Hugs \( \triangleright h(f \ 10) \)
20

Disadvantage: sometimes extra space needed.
fun facti(n,p) = if n = 0 then p else facti(n-1,n*p);
facti(4,1) = facti(4-1,4*1)
= facti(3-1, 3 \ast (4 \ast 1))
= facti(2-1, 2 \ast (3 \ast (4 \ast 1)))
= facti(1-1, 1 \ast (2 \ast (3 \ast (4 \ast 1))))
= facti(0, 1 \ast (2 \ast (3 \ast (4 \ast 1))))
= 1 \ast (2 \ast (3 \ast (4 \ast 1)))
Outline

1. Expression Evaluation
   - Call by value
   - Call-by-name
   - Call-by-need

2. Tuples

3. Functions and Operators
   - Infix Operators
   - Local Declarations

4. Food for Thought
**Tuples**

**Example**

Hugs ▷ (1.3, 29.5)
(1.3,3.4) :: (Double, Double)
Hugs ▷ (3, True)
(3,True) :: (Integer, Bool)
Hugs ▷ (3.1, (False,0))
(3.1,(False, 0)) :: (Double, (Bool,Integer))
Hugs ▷ (17-8, 4>5)
(9,False) :: (Integer, Bool)

**Definition**

A tuple is an anonymous constructor containing at least two components; component order matters.
Vectors

Example

type Vector = (Double, Double)

Observation

This is just a shortcut for a pair of real numbers.
Where to use tuples?

Example

- fun scvec r (x,y):vector = (r*x,r*y);
- scvec 2.0 (3.5, 2.4);
- val it = (7.0, 4.8) : vector

Observation

Tuples are useful when a function needs to be applied on any number of arguments and to return any number of results.
Let me introduce Brad ...

data Sex = M | F
type Person = (String, Int, Sex, String)
a1 :: Person
a1 = ("Brad", 46, M, "L. A.")
(name, age, sex, location) = a1

Hugs> age
46 :: Int
Hugs> location
"L.A." :: [Char]
Hugs> sex

ERROR - Cannot find "show" function for:
*** Expression : sex
*** Of type : Sex

Printing components of a tuple

The system can print a String, but not a user-defined data.
Printing components of a tuple

Example

printSex :: Sex -> String
printSex s = case s of
  M -> "Male"
  F -> "Female"
Hugs> printSex sex
Male :: [Char]
Example

datatype sex = m | f;
type person = string * int * sex * string;
val a1 : person = ("Brad", 46, m, "L. A.");
val n = #1 a1;
> n;
val it = "Brad" : string
Outline

1. Expression Evaluation
   - Call by value
   - Call-by-name
   - Call-by-need

2. Tuples

3. Functions and Operators
   - Infix Operators
   - Local Declarations

4. Food for Thought
An infix operator is a function written between its arguments. The reason for doing this is **syntactic sugaring**.

**Example**

- infix xor;
- fun x xor y = (x orelse y) andalso not (x andalso y);
- val xor = fn : bool * bool -> bool
- true xor true;
- val it = false : bool
Infix Operators

Operators Associativity in ML

Infix and infixr mean left and right associativity

We can also specify the operator’s precedence.

Example

▷ infix 6 plus;
▷ fun x plus y = "( ^ x ^ )+" ^ y ^ "");
val plus = fn : string * string -> string
▷"1" plus "2" plus "3";
val it = "((1+2)+3)" : string

Example

▷infixr 6 plusr;
▷ fun x plusr y = "( ^ x ^ )+" ^ y ^ "");
val plusr = fn : string * string -> string
▷"1" plusr "2" plusr "3";
val it = "(1+(2+3))" : string
Operators Associativity in Haskell

`infixl` and `infixr` mean left and right associativity

**Example**

```haskell
infixl 6 

x \#\# y = ")++x++"++y++")"

Hugs \(1\#\#2\#\#3\)

"((1+2)+3)" :: [Char]
```

**Remark**

An infix operator could be used as prefix function.

**Example**

```haskell
\(\text{op} \#\# \text{("John ", "Doe")};\)

"John Doe" :: string

5 'div' 2 same as div 5 2
```
Local Declarations

Bottom-up vs. top-down

```haskell
let D in E
E where D
```

Example

```haskell
f x = let doubleof z = 2 * z
ten = 10
(a,b) = x
in doubleof(ten * (a + b))
```

Hugs ▶ f(2,3)
100
Local Declarations: ML

Local declarations

1. `let D in E end;`
2. `local D₁ in D₂ end;`

Example

```ml
fun gcd (m, n) = if m = 0 then n else gcd (n mod m, m);

fun ratio (n, d) = let val c = gcd (n, d)
                           in (n div c, d div c) end;
```
Local Declarations

Private Functions

Example

```
local
  fun fibit n p c = if n=0 then c
                     else fibit (n-1) c (p+c)
  in
  fun fib n = fibit n 1 0
  end;
```

*fibit* is a private function of *fib*. 
Simultaneous declarations

Example

PolyML> val x = 1;
val x = 1 : int
PolyML> val y = 2;
val y = 2 : int
PolyML> val x = y and y = x;
val x = 2 : int
val y = 1 : int

Note

All values on the right hand side are evaluated and after the identifiers on the left hand side get the corresponding values.
Mutual Recursive Functions

\[
\frac{\pi}{4} = 1 - \frac{1}{3} + \frac{1}{5} - \frac{1}{7} + \cdots + \frac{1}{4k+1} - \frac{1}{4k+3}
\]

Partial sums up to a positive term

\[
\begin{align*}
\text{poz } 1 &= 1 \\
\text{poz } 5 &= 1 - \frac{1}{3} + \frac{1}{5} \\
\text{poz } 9 &= 1 - \frac{1}{3} + \frac{1}{5} - \frac{1}{7} + \frac{1}{9}
\end{align*}
\]

Negativ sums up to a positive term

\[
\begin{align*}
\text{neg } 3 &= 1 - \frac{1}{3} \\
\text{neg } 7 &= 1 - \frac{1}{3} + \frac{1}{5} - \frac{1}{7} \\
\text{neg } 11 &= 1 - \frac{1}{3} + \frac{1}{5} - \frac{1}{7} + \frac{1}{9} - \frac{1}{11}
\end{align*}
\]

Relationship between poz and neg

\[
\begin{align*}
\text{neg } d &= \text{poz } (d-2) - \frac{1}{d}, \text{ } d=4k+3, k \geq 0 \\
\text{poz } d &= \text{neg}(d-2) + \frac{1}{d}, \text{ } d=4k+1, k \geq 1 \\
\text{poz } 1 &= 1, \text{ neg } -1 = 0
\end{align*}
\]

fun poz d = neg (d-2.0)+1.0/d and neg d = if d > 0.0 then poz (d-2.0)-1.0/d else 0.0;

val poz d = fn:real→real; val neg d = fn:real→real;
poz = 1000; poz = 1000.0; poz = 1001.0
n = 10;
i = 1;
s = 0;
sum: if i <= n then
    s = s + i;
    goto sum1
else stop
sum1: i = i + 1;
goto sum

fun sum(n,i,s) = if i <= n then sum1(n,i,s + i)
    else (n,i,s)
and sum1(n,i,s) = sum(n,i + 1,s);
val sum = fn : int * int * int -> int * int * int
val sum1 = fn : int * int * int -> int * int * int
PolyML ▷ sum (4, 1,0);
(4, 5, 10)
Outline

1 Expression Evaluation
   - Call by value
   - Call-by-name
   - Call-by-need

2 Tuples

3 Functions and Operators
   - Infix Operators
   - Local Declarations

4 Food for Thought
Expression evaluation

Question
What is the evaluation mechanism in different programming languages?

Example
- ML - call by value
- Haskell - call by need

Example
- Java - ?
- C++, C# - ?
- Php, Perl - ?
Call-by-need (arguments evaluated just once, when needed) and call-by-value (arguments evaluated before being processed) have both advantages and disadvantages.

- In order to program efficiently you have to know how the parameters are evaluated.
Study ideas

Mandatory reading
Chapter 2 from Ioan Alfred Letia, Programare Functionala

For the hard working students
Paul Hudak, John Peterson, and Joseph H. Fasel, A Gentle Introduction to Haskell 98, 1999