301AA - Advanced Programming
[AP-2017]

Lecturer: Andrea Corradini andrea@di.unipi.it
Tutor: Lillo Galletta galletta@di.unipi.it

Department of Computer Science, Pisa
Academic Year 2017/18

AP-2017-16: Haskell, Call by need
Applicative and Normal Order evaluation

- **Applicative Order** evaluation
  - Arguments are evaluated before applying the function – aka *Eager evaluation, parameter passing by value*
- **Normal Order** evaluation
  - Function evaluated first, arguments if and when needed
  - Sort of *parameter passing by name*
  - Some evaluation can be repeated
- Church-Rosser
  - If evaluation terminates, the result (*normal form*) is unique
  - If some evaluation terminates, normal order evaluation terminates

\[ \beta\text{-conversion} \]
\[
(\lambda x. t) \; t' = t [t'/x]
\]

**Applicative order**
\[
(\lambda x. (+ x x)) \; (+ 3 2) \\
\to (\lambda x. (+ x x)) \; 5 \\
\to (+ 5 \; 5) \\
\to 10
\]

**Normal order**
\[
(\lambda x. (+ x x)) \; (+ 3 2) \\
\to (+ (+ 3 \; 2) \; (+ 3 \; 2)) \\
\to (+ 5 \; (+ 3 \; 2)) \\
\to (+ 5 \; 5) \\
\to 10
\]

Define \( \Omega = (\lambda x. x \; x) \)

Then
\[
\Omega \; \Omega = (\lambda x. x \; x) \; (\lambda x. x \; x) \\
\to x \; x \; [(\lambda x. x \; x)/x] \\
\to (\lambda x. x \; x) \; (\lambda x. x \; x) = \Omega \; \Omega \\
\to \cdots \text{ non-terminating}
\]

\[
(\lambda x. \; 0) \; (\Omega \; \Omega) \\
\to \{ \text{Applicative order} \} \\
\to \{ \text{Normal order} \} \\
\to 0
\]

Non-terminating
A digression on parameter passing
Parameter Passing Mechanisms

• Parameter passing modes
  – In
  – In/out
  – Out

• Parameter passing mechanisms
  – Call by value (in)
  – Call by reference (in+out)
  – Call by result (out)
  – Call by value/result (in+out)
  – Call by need (in)
  – Call by sharing (in/out)
  – Call by name (in+out)
L-Values vs. R-Values and Value Model vs. Reference Model

• Consider the assignment of the form:   \(a := b\)
  – The left-hand side \(a\) of the assignment is an \textit{l-value} which is an expression that should denote a location, e.g. array element \(a[2]\) or a variable \texttt{foo} or a dereferenced pointer \(\ast p\) or a more complex expression like \((f(a)+3)\rightarrow b[c]\)
  – The right-hand side \(b\) of the assignment is an \textit{r-value} which can be any syntactically valid expression with a type that is compatible to the left-hand side

• Languages that adopt the \textit{value model} of variables copy the value of \(b\) into the location of \(a\)

• Languages that adopt the \textit{reference model} of variables copy references, resulting in shared data values via multiple references
  – Lisp/Scheme, ML, Haskell, Smalltalk adopt the reference model. They copy the reference of \(b\) into \(a\) so that \(a\) and \(b\) refer to the same object
  – Most imperative programming languages use the value model
  – \texttt{Java} uses the value model for built-in types and the reference model for class instances
  – \texttt{C#} uses value model for value types, reference model for reference types
Assignment in
Value Model vs. Reference Model

b := 2;
c := b;
a := b + c

Value model

Reference model

a := b + c
b := 2
c := b

a 4
b 2
c 2

a 4
b 2
c 2
References and pointers

• Most implementations of PLs have as target architecture a Von Neumann one, where memory is made of cells with addresses.
• Thus implementations use the value model of the target architecture.
• Assumption: every data structure is stored in memory cells.
• We “define”:
  – A reference to $X$ is the address of the (base) cell where $X$ is stored.
  – A pointer to $X$ is a location containing the address of $X$.
• Value model based implementation can mimic the reference model using pointers and standard assignment.
  – Each variable is associated with a location.
  – To let variable $y$ refer to data $X$, the address of (reference to) $X$ is written in the location of $y$, which becomes a pointer.
Parameter Passing by Sharing

- **Call by sharing**: parameter passing of data in the reference model
- The value of the variable is passed as actual argument, which in fact is a reference to the (shared) data
  - Essentially this is call by value of the variable!
- Java uses both pass by value and pass by sharing
  - Variables of primitive built-in types are passed by value
  - Class instances are passed by sharing
  - The implementation is identical
Parameter Passing in Algol 60

- Algol 60 uses *call by name* by default, but also *call by value*
- Effect of *call by name* is like β-reduction in λ-calculus: the actual parameter is *copied* wherever the formal parameter appears in the body, then the resulting code is executed
- Thus the actual parameter is evaluated a number of times (0, 1, ...) that depends on the logic of the program
- Since the actual parameter can contain names, it is passed in a *closure* with the environment at invocation time (called a *thunk*)
- Call by name is powerful but makes programs difficult to read and to debug (think to λ-calculus...): dismissed in subsequent versions of Algol
An example of Call by Name: Jensen’s device

• What does the following Algol 60 procedure compute?

```
real procedure sum(expr, i, low, high);
  value low, high;         low and high are passed by value
  real expr;               expr and i are passed by name
  integer i, low, high;
begin
  real rtn;
  rtn := 0;
  for i := low step 1 until high do
    rtn := rtn + expr;
  sum := rtn               return value by assigning to function name
end sum
```

• Apparently, \((\text{high} - \text{low} + 1) \times \text{expr}\)
An example of Call by Name: Jensen’s device

• But: \[ y := \text{sum}(3\times x^2 - 5x + 2, x, 1, 10) \]

```
real procedure sum(expr, i, low, high);
  value low, high;        \textit{low and high are passed by value}
  real expr;             \textit{expr and i are passed by name}
  integer i, low, high;
begin
  real rtn;
  rtn := 0;
  for x := low step 1 until high do
    rtn := rtn + 3\times x^2 - 5x + 2;
  sum := rtn          \textit{return value by assigning to function name}
end sum
```

• It computes \[ y = \sum_{x=1}^{10} 3x^2 - 5x + 2 \]
Call by name & Lazy evaluation (*call by need*)

- In *call by name* parameter passing (default in Algol 60) arguments (like expressions) are passed as a **closure** ("thunk") to the subroutine
- The argument is (re)evaluated each time it is used in the body
- Haskell realizes **lazy evaluation** by using *call by need* parameter passing, which is similar: an expression passed as argument is evaluated only if its value is needed.
- Unlike *call by name*, the argument is evaluated **only the first time**, using **memoization**: the result is saved and further uses of the argument do not need to re-evaluate it
- Combined with **lazy data constructors**, this allows to construct potentially infinite data structures and to call infinitely recursive functions without necessarily causing non-termination
- **Note**: lazy evaluation works fine with **purely functional** languages
- Side effects require that the programmer reasons about the order that things happen, not predictable in lazy languages.
### Summary of Parameter Passing Modes

<table>
<thead>
<tr>
<th>parameter mode</th>
<th>representative languages</th>
<th>implementation mechanism</th>
<th>permissible operations</th>
<th>change to actual?</th>
<th>alias?</th>
</tr>
</thead>
<tbody>
<tr>
<td>value</td>
<td>C/C++, Pascal, Java/C# (value types)</td>
<td>value</td>
<td>read, write</td>
<td>no</td>
<td>no</td>
</tr>
<tr>
<td>in, const</td>
<td>Ada, C/C++, Modula-3</td>
<td>value or reference</td>
<td>read only</td>
<td>no</td>
<td>maybe</td>
</tr>
<tr>
<td>out</td>
<td>Ada</td>
<td>value or reference</td>
<td>write only</td>
<td>yes</td>
<td>maybe</td>
</tr>
<tr>
<td>value/result</td>
<td>Algol W</td>
<td>value</td>
<td>read, write</td>
<td>yes</td>
<td>no</td>
</tr>
<tr>
<td>var, ref</td>
<td>Fortran, Pascal, C++</td>
<td>reference</td>
<td>read, write</td>
<td>yes</td>
<td>yes</td>
</tr>
<tr>
<td>sharing</td>
<td>Lisp/Scheme, ML, Java/C# (reference types)</td>
<td>value or reference</td>
<td>read, write</td>
<td>yes</td>
<td>yes</td>
</tr>
<tr>
<td>in out</td>
<td>Ada</td>
<td>value or reference</td>
<td>read, write</td>
<td>yes</td>
<td>maybe</td>
</tr>
<tr>
<td>name</td>
<td>Algol 60, Simula</td>
<td>closure (thunk)</td>
<td>read, write</td>
<td>yes</td>
<td>yes</td>
</tr>
<tr>
<td>need</td>
<td>Haskell, R</td>
<td>closure (thunk) with memoization</td>
<td>read, write*</td>
<td>yes*</td>
<td>yes*</td>
</tr>
</tbody>
</table>
Back to Haskell
Laziness

- Haskell is a **lazy** language
- Functions and data constructors (also user-defined ones) don’t evaluate their arguments until they need them

```
cond True  t e = t
cond False t e = e
cond :: Bool -> a -> a -> a
cond True  []  [1..] => []
```

- Programmers can write control-flow operators that have to be built-in in eager languages

```
(||) :: Bool -> Bool -> Bool
True || x = True
False || x = x
```
List Comprehensions

• Notation for constructing new lists from old ones:

```python
twiceData = [2 * x | x <- myData]  
-- [2,4,6,8,10,12,14]

twiceEvenData = [2 * x | x <- myData, x `mod` 2 == 0]  
-- [4,8,12]
```

• Similar to “set comprehension”

\[ \{ x | x \in A \land x > 6 \} \]
More on List Comprehensions

ghci> [ x | x <- [10..20], x /= 13, x /= 15, x /= 19]  
[10,11,12,14,16,17,18,20]  -- more predicates

ghci> [ x*y | x <- [2,5,10], y <- [8,10,11]]  
[16,20,22,40,50,55,80,100,110]  -- more lists

length xs = sum [1 | _ <- xs]  -- anonymous (don’t care) var

-- strings are lists...
removeNonUppercase st = [ c | c <- st, c `elem` ['A'..'Z']]
Datatype Declarations

• Examples

```haskell
data Color = Red | Yellow | Blue

elements are Red, Yellow, Blue
```

```haskell
data Atom = Atom String | Number Int

elements are Atom “A”, Atom “B”, ..., Number 0, ...
```

```haskell
data List = Nil | Cons (Atom, List)

elements are Nil, Cons(Atom “A”, Nil), ...
Cons(Number 2, Cons(Atom(“Bill”), Nil)), ...
```

• General form

```haskell
data <name> = <clause> | ... | <clause>
<clause> ::= <constructor> | <constructor> <type>
```

– Type name and constructors must be Capitalized.
Datatypes and Pattern Matching

• Recursively defined data structure

\[
\text{data Tree} = \text{Leaf Int} \mid \text{Node (Int, Tree, Tree)}
\]

Node\((4, \text{Node}(3, \text{Leaf} 1, \text{Leaf} 2), \text{Node}(5, \text{Leaf} 6, \text{Leaf} 7))\)

• Constructors can be used in Pattern Matching

• Recursive function

\[
\begin{align*}
\text{sum (Leaf n)} &= n \\
\text{sum (Node(n,t1,t2))} &= n + \text{sum(t1)} + \text{sum(t2)}
\end{align*}
\]
Case Expression

• Datatype

\[
\text{data Exp} = \text{Var Int} \mid \text{Const Int} \mid \text{Plus (Exp, Exp)}
\]

• Case expression

\[
\text{case } e \text{ of }
\begin{align*}
\text{Var } n & \rightarrow \ldots \\
\text{Const } n & \rightarrow \ldots \\
\text{Plus}(e1,e2) & \rightarrow \ldots
\end{align*}
\]

– Indentation matters in case statements in Haskell.
Function Types in Haskell

In Haskell, \( f :: A \rightarrow B \) means for every \( x \in A \),

\[
f(x) = \begin{cases} 
\text{some element } y = f(x) \in B \\
\text{run forever}
\end{cases}
\]

In words, “if \( f(x) \) terminates, then \( f(x) \in B \).”

In ML, functions with type \( A \rightarrow B \) can throw an exception or have other effects, but not in Haskell

```
Prelude> :t not  -- type of some predefined functions
not :: Bool -> Bool
Prelude> :t (+)
(+) :: Num a => a -> a -> a
Prelude> :t (:)
(:) :: a -> [a] -> [a]
Prelude> :t elem
elem :: Eq a => a -> [a] -> Bool
```

Note: if \( f \) is a standard binary function, \( \text{\textasciitilde f} \) is its infix version
If \( x \) is an infix (binary) operator, \( (x) \) is its prefix version.
Higher-Order Functions

• Functions that take other functions as arguments or return as a result are **higher-order functions**.
• Common Examples:
  – **Map**: applies argument function to each element in a collection.
  – **Reduce** (**foldl**, **foldr**): takes a collection, an initial value, and a function, and combines the elements in the collection according to the function.
  – **Filter**: takes a collection and a boolean predicate, and returns the collection of the elements satisfying the predicate

```haskell
Prelude> :t map
map :: (a -> b) -> [a] -> [b]
Prelude> let list = [1,2,3]
Prelude> map (\x -> x+1) list
[2,3,4]
Prelude> :t foldl
foldl :: (b -> a -> b) -> b -> [a] -> b
Prelude> foldl (\accum i -> i + accum) 0 list
6
```
From imperative to functional programming
Searching a substring: Java code

static int indexOf(char[] source, int sourceOffset, int sourceCount,
        char[] target, int targetOffset, int targetCount,
        int fromIndex) {

    char first = target[targetOffset];
    int max = sourceOffset + (sourceCount - targetCount);

    for (int i = sourceOffset + fromIndex; i <= max; i++) {
        /* Look for first character. */
        if (source[i] != first) {
            while (++i <= max && source[i] != first);
        }

        /* Found first character, now look at the rest of v2 */
        if (i <= max) {
            int j = i + 1;
            int end = j + targetCount - 1;
            for (int k = targetOffset + 1; j < end && source[j] ==
                target[k]; j++, k++) {

                if (j == end) {
                    /* Found whole string. */
                    return i - sourceOffset;
                }
            }
        }
    }

    return -1;
}
Searching a Substring:
Exploiting Laziness

```haskell
isPrefixOf :: Eq a => [a] -> [a] -> Bool
-- returns True if first list is prefix of the second
isPrefixOf [] x = True
isPrefixOf (y:ys) [] = False
isPrefixOf (y:ys)(x:xs) = 
  if (x == y) then isPrefixOf ys xs else False

suffixes:: String -> [String]
-- All suffixes of s
suffixes[]     = [[]]
suffixes(x:xs) = (x:xs) : suffixes xs

or :: [Bool] -> Bool
-- (or bs) returns True if any of the bs is True
or []       = False
or (b:bs)   = b || or bs

isSubString :: String -> String -> Bool
x `isSubString` s = or [ x `isPrefixOf` t |
  t <- suffixes s ]
```