301AA - Advanced Programming

Lecturer: **Andrea Corradini**

andrea@di.unipi.it

http://pages.di.unipi.it/corradini/

***AP-14**: Lambda Calculus, Haskell, Call by need*

Summary

- Motivation: Laziness in Haskell
- Lambda Calculus
- Parameter passing mechanisms
  - Call by sharing
  - Call by name
  - Call by need

On laziness in Haskell

- Haskell is a **lazy** language
- Functions and data constructors don't evaluate their arguments until they need them
- In several languages there are forms of lazy evaluations (**if-then-else**, shortcutting **&&** and **||**)

```
if (x != 0) return y/x; else return 0; //ok
if (x !=0 && y/x > 5) return 0; else return 1; //ok
if (x !=0 & y/x > 5) return 0; else return 1; //no

int choose(boolean e1, boolean e2){
   if (e1 && e2) return 0; else return 1;
}
choose(x!=0, y/x>5) // ???
```

- Ok in Haskell, thanks to **Normal Order evaluation** and **Call by Need** parameter passing…

λ-calculus: syntax

λ-terms: $t ::= x \mid \lambda x.t \mid t\ t \mid (t)$

- x *variable, name, symbol,…*
- $\lambda x.t$ *abstraction,* defines an anonymous function
- $t\ t'$ *application* of function t to argument t'

Terms can be represented as abstract syntax trees

Syntactic Conventions

- Applications associates to left
  $t_1\ t_2\ t_3 \equiv (t_1\ t_2)\ t_3$
- The body of abstraction extends as far as possible
  - $\lambda x.\ \lambda y.\ x\ y\ x \equiv (\lambda x.\ (\lambda y.\ (x\ y)\ x))$

A simple tutorial on lambda calculus:
`http://www.inf.fu-berlin.de/lehre/WS03/alpi/lambda.pdf`

Free vs. Bound Variables

- An occurrence of x is free in a term t if it is not in the body of an abstraction $\lambda x.\ t$
  - otherwise it is bound
  - λx is a binder
- Examples
  - $\lambda z.\ \lambda x.\ \lambda y.\ x\ (y\ z)$
  - $(\lambda x.\ x)\ x$
- Terms without free variables are combinators
  - Identity function: $id = \lambda x.\ x$
  - First projection: $fst = \lambda x.\ \lambda y.\ x$

Operational Semantics

[β-reduction] *function application*

redex $\boxed{(\lambda x.t)\ t'} = t\ [t'/x]$

$(\lambda\ x.\ x)\ y \to \quad y$

$(\lambda\ x.\ x\ (\lambda\ x.\ x)\ )\ (u\ r) \to \quad u\ r\ (\lambda\ x.x)$

$(\lambda\ x.\ (\lambda w.\ x\ w))\ (y\ z) \to \quad \lambda w.\ y\ z\ w$

$(\lambda\ x.\ x\ x)(\lambda x.\ x\ x) \to \quad (\lambda x.\ x\ x)\ (\lambda x.\ x\ x)$

Other relevant concepts:

- Normal Forms, α-conversion, η-reduction

λ-calculus as a functional language

Despite the simplicity, we can encode in λ-calculus most concepts of functional languages:

- Functions with several arguments
- Booleans and logical connectives
- Integers and operations on them
- Pairs and tuples
- Recursion
- …

Functions with several arguments

- A definition of a function with a single argument associates a name with a λ-abstraction

```
f x = <exp>      -- is equivalent to
f = λx.<exp>
```

- A function with several argument is equivalent to a sequence of λ-abstractions

```
f(x,y) = <exp>  -- is equivalent to
f = λx. λy.<exp>
```

- "Currying" and "Uncurrying"

```
curry :: ((a, b) -> c) -> a -> b -> c
curry f x y = f(x,y)
uncurry :: (a -> b -> c) -> (a, b) -> c
uncurry f (x,y) = f x y
```

Church Booleans

- `T = λt.λf.t -- first`
- `F = λt.λf.f -- second`
- `and = λb.λc.bcF`
- `or = λb.λc.bTc`
- `not = λx.xFT`
- `test =λl.λm.λn.lmn`

```
and T F
→ (λb.λc.bcF) T F
→ (λc.TcF) F
→ TFF
→ F
```

```
not F
→ (λx.xFT) F
→ FFT
→ T
```

```
test F u w
→ (λl.λm.λn.lmn) F u w
→ (λm.λn.Fmn) u w
→ (λn.Fun) w
→ Fuw
→ w
```

Pairs

- `pair = λf.λs.λb.b f s`
- `fst = λp.p T`
- `snd = λp.p F`

```
fst (pair u w)
→ (λp.p T) (pair u w)
→ (pair u w) T
→ (λf.λs.λb.b f s) u w T
→ (λs.λb.b u s) w T
→ (λb.b u w) T
→ T u w
→ u
```

Church Numerals

Higher order functions:
n takes a function ***s*** as argument and returns the *n*-th composition of ***s*** with itself, $\boldsymbol{s^n}$

- **0** = `λs. λz. z`
- **1** = `λs. λz. s z`
- **2** = `λs. λz. s (s z)`
- **3** = `λs. λz. s (s (s z))`

A first simple function:

$\boldsymbol{s^n}$

- `succ = λn. λs. λz. s (n s z)`

```
succ 2
→ (λn. λs. λz. s (n s z)) 2
→ (λs. λz. s (2 s z))
→ (λs. λz. s ((λs. λz. s (s z)) s z))
→ (λs. λz. s (s (s z)))   = 3
```

applies the function one more time

Arithmetics with Church Numerals

Addition:

- `plus = λm. λn. λs. λz.` $\underbrace{\texttt{m s}}_{}$ `(n s z)` — s^m

Multiplication:

- `times =  λm. λn. λs. λz. m (n s) z` — s^n $(s^n)^m = s^{n*m}$

Exponentiation:

- `pow =  λm. λn. λs. λz. n m s z` — m^n

Test by zero:

- `Z = λx. x F not F`
- `Z` **0** `= ((`**0** `F) not) F = not F = T`
- `Z` **n** `= ((`**n** `F) not) F =` F^n`(not) F = F`

Fix-point combinator and recursion

The following ***fix-point combinator*** **Y**, when applied to a function **R**, returns a fix-point of **R**, i.e. **R(YR) = YR**

- **Y = (λy.(λx.y(x x))(λx.y(x x)))**
- **YR** = (λx.R(x x))(λx.R(x x))

  = R((λx.R(x x))(λx.R(x x))) = **R(YR)**

A recursive function definition (like *factorial*) can be read as a higher-order transformation having a function as first argument, and the desired function is its fix-point.

Fix-point combinator and recursion

A recursive definition:

- `sums(n) = (n==0 ? 0 : n + sums(n-1))`
- `sums = \n -> (n == 0 ? 0 : n + sums(n-1))`

`sums` is the fix-point of the following higher-order function:

- `R = \F -> \n -> (n == 0? 0 : n + F(n-1))`
- `R=(λr.λn.Z n 0 (n S (r (`**`P`**` n))))` // in λ−calculus

Example of application

```
(Y R) 3 = R (Y R) 3 =
(3 == 0? 0 : 3 + (Y R) (3-1)) =
3 + (Y R) 2 =
3 + R (Y R) 2 =
3 + (2 == 0? 0 : 2 + (Y R) (2-1)) =
3 + 2 + (Y R) 1 =
... 3 + 2 + 1 + 0 = 6
```

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

Define **Ω = (λx.x x)**

Then

ΩΩ = (λx.x x) (λx.x x)

→ x x [(λx.x x)/x]

→ (λx.x x) (λx.x x) = ΩΩ

→ … *non-terminating*

(λx. 0) (ΩΩ)

→ { *Applicative order*}

… *non-terminating*

(λx. 0) (ΩΩ)

→ { *Normal order*}

0

β-conversion

(λx.t) t' = t [t'/x]

Applicative order

(λx.(+ x x)) (+ 3 2)

→ (λx.(+ x x)) 5

→ (+ 5 5)

→ 10

Normal order

(λx.(+ x x)) (+ 3 2)

→ (+ (+ 3 2) (+ 3 2))

→ (+ 5 (+ 3 2))

→ (+ 5 5)

→ 10

Parameter passing mechanism in Haskell: *Call by need*

- Haskell realizes ***lazy evaluation*** by using ***call by need*** parameter passing: an expression passed as argument is bound to the formal parameter, but it is evaluated only if its value is needed.
- The argument is evaluated *only the first time*, using the ***memoization technique***: the result is saved and further uses of the argument do not need to re-evaluate it

Call by need (cont.)

- 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.
- We will address this fact when introducing Hakell's IO-Monad

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***
  - ***a*** is an ***l-value***, an expression denoting a **location**, e.g.
    - an array element **a[2]**
    - a variable **foo**
    - a dereferenced pointer ***p**
    - a more complex expression like **(f(a)+3)->b[c]**
  - ***b*** is an ***r-value***: any syntactically valid expression with a type compatible to that of **a**
- Languages that adopt the ***value model*** of variables copy the value of ***b*** into the location of ***a***
- Languages that adopt the ***reference model*** of variables copy references, resulting in shared data values via multiple references

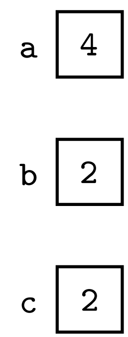
Value Model vs. Reference Model in some programming languages

- 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
- **Java** uses the value model for built-in types and the reference model for class instances
- C# uses value model for value types, reference model for reference types

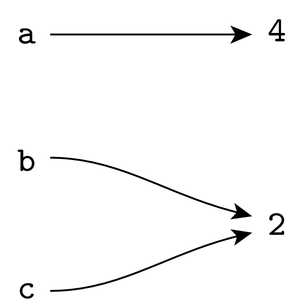
Assignment in Value Model vs. Reference Model

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

Value model



Reference model



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 implementations 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, `(high-low+1) * expr`

An example of Call by Name: Jensen's device

- But: `y := sum(3*x*x-5*x+2,x,1,10)`

```
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 x := low step 1 until high do
    rtn := rtn + 3*x*x-5*x+2;
  sum := rtn         return value by assigning to function name
end sum
```

- It computes $$y = \sum_{x=1}^{10} 3x^2\text{-}5x+2$$

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

Call by name & Lazy evaluation (*call by need*)

- 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.
- We will address this fact when introducing Hakell's IO-Monad

Summary of Parameter Passing Modes

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