301AA - Advanced Programming

Lecturer: **Andrea Corradini**
andrea@di.unipi.it
http://pages.di.unipi.it/corradini/

AP-17*: Lambda Calculus, Haskell, Call by need*

Summary

- Lambda Calculus
- Parameter passing mechanisms
  - Call by sharing
  - Call by name
  - Call by need

λ-calculus: syntax

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

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

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
  - $\lambda z.\ \lambda x.\ \lambda y.\ x\ (y\ z)$
  - $(\lambda x.\ x)\ x$

Operational Semantics

[β-reduction] *function application*

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

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

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

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

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

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, 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:

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:

- $\texttt{plus} = \lambda m.\ \lambda n.\ \lambda s.\ \lambda z.\ \overset{s^m}{\boxed{m\ s}}\ (n\ s\ z)$

Multiplication:

- $\texttt{times} = \lambda m.\ \lambda n.\ \lambda s.\ \lambda z.\ \boxed{m\ (\overset{s^n}{\boxed{n\ s}})}\ z$ $\quad (s^n)^m = s^{n*m}$

Exponentiation:

- $\texttt{pow} = \lambda m.\ \lambda n.\ \lambda s.\ \lambda z.\ \overset{m^n}{\boxed{n\ m}}\ s\ z$

Test by zero:

- $Z = \lambda x.\ x\ F\ \texttt{not}\ F$
- $Z\ \mathbf{0} = ((\mathbf{0}\ F)\ \texttt{not})\ F = \texttt{not}\ F = T$
- $Z\ \mathbf{n} = ((\mathbf{n}\ F)\ \texttt{not})\ F = F^n(\texttt{not})\ F = F$

def of numerals: n = \s. \z. s^n (z)

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**

def of fix-point combinator

- $\mathbf{Y = (\lambda y.(\lambda x.y(x\ x))(\lambda x.y(x\ x)))}$

- $\mathbf{YR} = (\lambda x.R(x\ x))(\lambda x.R(x\ x))$
  $= R((\lambda x.R(x\ x))(\lambda x.R(x\ x))) = \mathbf{R(YR)}$

theorem: R(YR) = 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

def of fixpoint

```
(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 (aka Lazy Evaluation)
  - Function evaluated first, arguments if and when needed
  - Sort of *parameter passing by name*
  - Some evaluation can be repeated
- Church-Rosser Theorem
  - 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

Warning!

- *The next slides about Parameter Passing Mechanisms where not presented during the course. They are included in this document for the interested reader*