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: $t := x \mid λx.t \mid tt \mid (t) \equiv$

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

Syntactic Conventions

Applications associates to left

$$t_1 t_2 t_3 \equiv (t_1 t_2) t_3 \equiv$$

- The body of abstraction extends as far as possible
 - λx . λy . $x y x = \lambda x$. $(\lambda y$. (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 λx . t
 - otherwise it is bound
 - $-\lambda x$ is a binder
- Examples
 - $-\lambda z. \lambda x. \lambda y. x (y z)$
 - (λx. x) x ≡

Operational Semantics

[β-reduction] function application
redex
$$(\lambda x.t) t' = t [t'/x]$$

$$(\lambda x. x) y \rightarrow y$$

$$(\lambda x. x (\lambda x. x)) (u r) \rightarrow u r (\lambda x. x) =$$

$$(\lambda x. (\lambda w. x w)) (y z) \rightarrow \lambda w. y z w$$

$$(\lambda x. x x)(\lambda x. x x) \rightarrow (\lambda x. x x) (\lambda x. x x)$$

Other relevant concepts:

• Normal Forms, α -conversion, η -reduction \equiv

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 = \langle exp \rangle -- is equivalent to
f = \lambda x . \langle exp \rangle
```

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

```
f(x,y) = \langle exp \rangle -- is equivalent to f = \lambda x \cdot \lambda y \cdot \langle exp \rangle
```

"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 = \lambda t \cdot \lambda f \cdot t -- first
• F = \lambda t \cdot \lambda f \cdot f -- second
```

- and = $\lambda b \cdot \lambda c \cdot b c F$
- or = $\lambda b \cdot \lambda c \cdot bTc$
- not = $\lambda x \cdot xFT$
- test = $\lambda 1.\lambda m.\lambda n.lmn$

```
test F u w
```

- \rightarrow (λ 1. λ m. λ n.lmn) F u w
- \rightarrow (λ m. λ n.Fmn) u w
- \rightarrow (λ n.Fun) w
- \rightarrow Fuw
- \rightarrow w

```
not F
```

- \rightarrow ($\lambda x.xFT$) F
- \rightarrow FFT
- \rightarrow T

Pairs

```
pair = \lambda f \cdot \lambda s \cdot \lambda b \cdot b f s \equiv
fst = \lambda p.p T
|snd| = |\lambda p \cdot p| F
 fst (pair u w)
  \rightarrow (\lambda p.p T) (pair u w)
  → (pair u w) T
  \rightarrow (\lambda f.\lambda s.\lambda b.b f s) u w T
  \rightarrow (\lambdas.\lambdab.b u s) w T
  \rightarrow (\lambda b.b.u.w) T
  \rightarrow T u w
  \rightarrow u
```

Church Numerals =

```
• 0 = \lambda s. \lambda z. z
• 1 = \lambda s. \lambda z. s z
```

•
$$\mathbf{2} = \lambda \mathbf{s} \cdot \lambda \mathbf{z} \cdot \mathbf{s} (\mathbf{s} \mathbf{z})$$

• 3 = λs . λz . s (s (s z))

A first simple function: ≡

• succ = $\lambda n \cdot \lambda s \cdot \lambda z \cdot s$

Higher order functions:

n takes a function s as argument and returns the n-th composition of s with itself, $s^n \equiv$

applies the function one more time

Arithmetics with Church Numerals

```
Addition:
• plus = \lambda m. \lambda n. \lambda s. \lambda z. m s (n s z)
Multiplication: |
• times = \lambda m. \lambda n. \lambda s. \lambda z. m (n s) z (s^n)^m = s^{n*m}
Exponentiation:
• pow = \lambda m. \lambda n. \lambda s. \lambda z. n m s z \equiv
Test by zero:
 Z = \lambda x. x F not F \rightleftharpoons
 Z O = ((O F) not) F = not F = T \blacksquare
   Z n = ((n F) not) F = F^n(not) F = F
 def of numerals: n = \frac{1}{2} \cdot \frac{7}{2} \cdot \frac{5}{2}
```

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. $R(YR) = YR \equiv$

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

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

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

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)) =

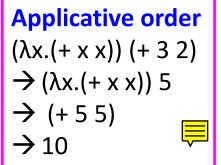
3 + 2 + (Y R) 1 =

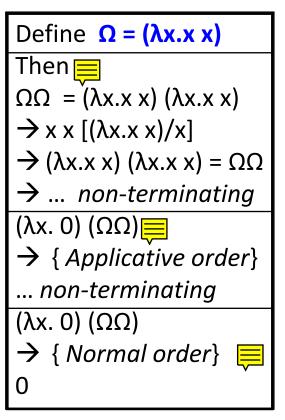
... 3 + 2 + 1 + 0 = 6
```

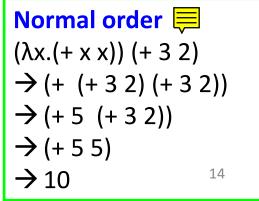
Applicative and Normal Order evaluation

- 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

β-conversion ($\lambda x.t$) t' = t [t'/x]







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 reevaluate 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

