


**Introduction to the λ -calculus
(Part II)**

CS386L- Programming Languages

Dr. Greg Lavender
 Department of Computer Sciences
 The University of Texas at Austin


CS386L: Lecture-04 - Introduction to the λ -calculus

IsZero

- isZero predicate
 - $\text{iszero} = \lambda n.n(\lambda x.F)T$
 - $\text{iszero } \underline{0} = (\lambda n.n(\lambda x.F)T) \lambda sz.z$
 $\Rightarrow (\lambda sz.z) (\lambda x.F)T$
 $\Rightarrow T$
 - $\text{iszero } \underline{S0} = (\lambda n.n(\lambda x.F)T) \lambda sz.s(z)$
 $\Rightarrow (\lambda sz.s(z)) (\lambda x.F)T$
 $\Rightarrow (\lambda x.F)T$
 $\Rightarrow F$
 - convince yourself that $\text{iszero } \underline{S_n}$ always returns F

2/19/04
2

Ordered Pairs

- Define lambda expressions for ordered pairs
 - $\text{fst} = \lambda p.p \text{ T}$
 - $\text{snd} = \lambda p.p \text{ F}$
 - $(E_1, E_2) = \lambda f.f E_1 E_2$
 - $\text{fst} (E_1, E_2) = (\lambda p.p \text{ T}) (E_1, E_2)$
 - $\Rightarrow (E_1, E_2) \text{ T}$
 - $\Rightarrow (\lambda f.f E_1 E_2) \text{ T}$
 - $\Rightarrow \text{T } E_1 E_2$
 - $\Rightarrow (\lambda t f.t) E_1 E_2$
 - $\Rightarrow E_1$
 - Similarly for $\text{snd} (E_1, E_2)$

n-Tuples

- Tuples are defined in terms of pairs
 - $(E_1, E_2, \dots, E_n) = (E_1, (E_2, (\dots(E_{n-1}, E_n)\dots)))$
 - $E^1 = \text{fst } E$
 - $E^2 = \text{fst}(\text{snd } E)$
 - $E^i = \text{fst}(\text{snd}(\text{snd}(\dots(\text{snd } E) \dots)))$ if $i < n$
 - $E^n = \text{snd}(\text{snd}(\dots(\text{snd } E)\dots))$
 - $n-1$ applications of snd
 - $(E_1, E_2, \dots, E_n)^2 = (E_1, (E_2, (\dots)))^2$
 - $\Rightarrow \text{fst} (\text{snd}(E_1, (E_2, (\dots))))$
 - $\Rightarrow \text{fst} (E_2, (\dots))$
 - $\Rightarrow E_2$
 - Prove that $(E_1, E_2, \dots, E_n)^i \Rightarrow E_i$ for $1 \leq i \leq n$

Predecessor function

- Now that we have defined booleans, a conditional test, ordered pairs, and functions to access elements of a pair, we can define a predecessor function!
- $\text{pred} = \lambda n f x. \text{snd}(n (\text{prefn } f) (T, x))$
 - where prefn is defined as:
 - $\text{prefn} = \lambda f p. (F, (\text{fst } p \rightarrow \text{snd } p \mid (f (\text{snd } p))))$
 - where $(E \rightarrow E_1 \mid E_2) = (\text{test } E E_1 E_2)$
 - Show the following:
 - $\text{pred} (\text{succ } \underline{n}) = \underline{n}$
 - $\text{pred } \underline{0} = \underline{0}$

Fixed points

- A "fixed point" is a value (x) in the domain of a function that is the same in the range ($f(x)$).
- Every value in the domain of the identity function is a fixed point
 - $\lambda x.x = x$
- can you think of others?
 - $\text{factorial}(1) = 1$
 - $\text{fibonacci}(0) = 0, \text{fibonacci}(1) = 1$
 - $\text{square}(0) = 0, \text{square}(1) = 1$
 - $\text{abs}(x) = x, \text{ if } x \geq 0$
 - $\text{sin}(0) = 0$
- Functorials may also have fixed points
 - $D_x(e^x) = e^x$

Recursion

- How do we define recursive expressions?
 - will this work?
 - $\text{mult} = \lambda m n. (\text{iszero } m \rightarrow 0 \mid \text{add } n (\text{mult } (\text{pred } m) n))$
 - No, there is a problem with this!
 - We can't define `mult` in terms of itself because the expression will never be complete
 - recursive self-reference introduces a challenge
 - this is similar to the problem of defining a quine in scheme or C, which is doable but requires special syntax

```
((lambda (x) `(,x ',x)) '(lambda (x) `(,x ',x)))
```

```
main(a){printf(a="main(a){printf(a=%c%s%c,34,a,34);}",34,a,34);}
```

Defining Recursive Functions

- Consider a fixed point operator `Fix`
 - $\text{Fix } E = E (\text{Fix } E)$
 - `Fix` is a function that takes a function as argument and repeats itself
 - there are many such fixed point operators
- The fixed point operator is used in the lambda calculus is called the **Y** combinator
 - $\mathbf{Y} = (\lambda f. (\lambda x. f(x x)) (\lambda x. f(x x)))$
 - $\mathbf{Y} E = (\lambda f. (\lambda x. f(x x)) (\lambda x. f(x x))) E$
 - $\Rightarrow (\lambda x. E(x x)) (\lambda x. E(x x))$
 - $\Rightarrow E (\lambda x. E(x x)) (\lambda x. E(x x))$
 - $\Rightarrow E (\mathbf{Y} E)$

Using the Y combinator

- Any expression of the form
 - $f x_1 \dots x_n = E$
is called recursive if f occurs free in E
 - if you want: $f x_1 \dots x_n = \dots f \dots$ then define
 - $F = Y (\lambda f x_1 \dots x_n. \dots f \dots)$
 - for example:
 - $\text{mult} = \lambda m n. (\text{iszero } m \rightarrow \underline{0} \mid \text{add } n (\text{mult } (\text{pred } m) n))$
 - $\text{multfn} = \lambda f m n. (\text{iszero } m \rightarrow \underline{0} \mid \text{add } n (f \text{ pred } m) n))$
 - $\text{mult} = Y \text{ multfn}$
 - $\text{mult } m n = (Y \text{ multfn}) m n$
 $\Rightarrow \text{multfn } (Y \text{ multfn}) m n$
 $\Rightarrow \text{multfn mult } m n$
 $\Rightarrow (\lambda f m n. (\text{iszero } m \rightarrow \underline{0} \mid \text{add } n (f \text{ pred } m) n)) \text{ mult } m n$
 $\Rightarrow (\text{iszero } m \rightarrow \underline{0} \mid \text{add } n (\text{mult } (\text{pred } m) n))$

2/19/04

9

Homework Problems

- Show the following:
 - $\text{mult } \underline{2} \underline{2} = \underline{4}$
 - Construct a λ -expression eq such that
 - $\text{eq } m n = (\text{iszero } m \rightarrow \text{iszero } n \mid$
 $\text{iszero } n \rightarrow F \mid \text{eq } (\text{pred } m) (\text{pred } n))$
- Show that the following expressions are fixed point operators:
 - $T_{\text{fix}} = (\lambda x y. y (x \times y)) (\lambda x y. y (x \times y))$ [due to Turing]
 - Given
 - $L = \lambda \text{abcdefghijklmnopqstuvwxyzr}. r(\text{thisisafixedpointcombinator})$
 - $B_{\text{fix}} = \text{LLLLLLLLLLLLLLLLLLLLLLLLLLLLLLLL}$
 - i.e., B_{fix} is L repeated 26 times

2/19/04

10

Currying/Uncurrying

- We saw previously examples of curried functions in ML, Haskell and Scheme
 - in the lambda calculus, we can define functions that curry and uncurry a function
 - $\text{curry} = \lambda f x_1 x_2. f (x_1, x_2)$
 - $\text{uncurry} = \lambda f p. f (\text{fst } p) (\text{snd } p)$
- Show that:
 - $\text{curry}(\text{uncurry } E) = E$
 - $\text{uncurry}(\text{curry } E) = E$

Normal Form Properties

- if $E_1 \Rightarrow^* E_2$ then E_2 was obtained by a sequence of reduction steps starting with E_1
- If E_2 can not be further reduced by β (or η) conversion, then E_2 is in normal form
- Normal form just means that we can do no further evaluation, i.e., we have the "answer"
- For example:
 - Church booleans and numerals are already in normal form
 - $(\lambda x.x) \underline{0}$ is not in normal form

Evaluation order

- We can evaluate an expression left-to-right
 - this is called "normal order" evaluation
 - $(\lambda y.\underline{1})(\lambda x.xx)(\lambda x.xx) \Rightarrow \underline{1}$
 - also often called lazy or non-strict evaluation
- Or we can evaluate right-to-left
 - this is called "applicative order" evaluation
 - $(\lambda y.\underline{1})(\lambda x.xx)(\lambda x.xx) \Rightarrow (\lambda y.\underline{1})((\lambda x.xx)(\lambda x.xx)) \Rightarrow \dots$
 - also often called eager or strict evaluation
- Note that evaluations could proceed in parallel

Important Results

- Church-Rosser Theorem (1937)
 - states that a term has at most one normal form
 - a normal form may not exist, i.e., $(\lambda x.xx)(\lambda x.xx)$
 - if two different reductions of E terminate in normal form, then the two normal forms are identical upto α -conversion
 - if $E_1 = E_2$ then there exists an E such that $E_1 \Rightarrow^* E$ and $E_2 \Rightarrow^* E$
 - This says that lambda expressions can be evaluated in any order and a normal form is irreducible
 - A λ -expression E has a normal form if $E = E'$ for E' in normal form

Church-Rosser Property

- Corollaries
 - if E has a normal form then $E \Rightarrow^* E'$ for some E' has a normal form
 - if E has a normal form and $E = E'$ then E' has a normal form
 - if $E = E'$ and E and E' are both in normal form, then E and E' are identical up to α -conversion

Combinators

- Combinatory logic
 - H.B. Curry & R. Feys, *Combinatory Logic*, Vol. I, 1958, Vol II, 1972.
 - J.R. Hindley & J. P. Seldin, *Introduction to Combinators and λ -Calculus*, 1986
- S, K & I combinators
 - $K = \lambda x y. x$
 - $S = \lambda f g x. (f x) (g x)$
 - $I = S K K = \lambda x. x$
 - $S K K = (\lambda f g x. (f x) (g x)) (\lambda ab. a) (\lambda uv. u)$
 - $\Rightarrow ((\lambda ab. a) x) ((\lambda uv. u) x)$
 - $\Rightarrow (\lambda ab. a) x x$
 - $\Rightarrow x$
 - combinators were first used by David Turner to implement a combinator reduction machine for a functional language by exploiting "functional completeness" --- which says that any lambda-expression can be translated to an equivalent combinatory expression.

Some computability theory

- **Godel numbering**
 - every program is represented by a finite string of symbols
 - an effective procedure can be defined that converts any program into a unique natural number e , called the godel number of the program
 - Hence, for any Turing machine M we can assign it a godel number e and denote the machine by M_e
 - Let U be a Turing machine that computes an input e and x . Call e the program and x the input to program e
 - $U =$ if (e is a program) then $M_e(x)$ else output 0
 - U is a universal turing machine and led J. von Neumann to "invent" the stored program concept used in modern computers

2/19/04

17

Kleene's Recursion Theorem

- **Also called Kleene's fixed point theorem**
 - let $\phi_e = \lambda x. U(e, x)$
 - For every computable function f there is a number n such that $\phi_n = \phi_{f(n)}$
- **Corollary**
 - There is a godel number n such that ϕ_n is the constant function with output n
 - Hence, n is the godel number of a "self-reproducing" program. I.e., a Turing machine whose code n does nothing on any input x except print its own code
 - That is, it is a quine!
 - If you construct a quine in some programming language, you have done a constructive proof
 - Recent research in linguistics, cognitive science and neuroscience indicates that humans are uniquely recursive in our abilities. Primates do not exhibit this cognitive ability! 18

2/19/04