

Recursive Definitions, Fixed Points and the Y Combinator

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

Recursive Self-Reference

- Recursive self-reference occurs regularly
 - it has been studied extensively within the context of logic & computability
 - S. Kleene, W.V.O. Quine, R. Smullyan, etc.
- Kleene's Recursion Theorem
 - fundamental theorem in computability
- Fixed Point Theory
 - fixed point operators
 - Y combinator

Recursive Definition

- Computable functions are often defined recursively.
 - how do we define them syntactically?
 - need a way to deal with the issue of self-referential naming as part of writing down a function definition.
 - can't define something in terms of itself without some mechanism to escape from the apparent infinite loop
 - Example:
 - let `fact n = if (n == 0) then 1 else n * fact(n-1) in fact(5);`
 - there is a technical syntactic definitional problem with this recursive expression
 - we cannot define the name `fact` in terms of itself because the expression will never be completed statically:
 - We don't know how far to "unroll" the definition unless we "run it"
 - recursive self-reference introduces a challenge to the language designer and compiler/interpreter implementer

2/7/05

3

Recursive Definition

- To illustrate the recursive definition problem more practically, consider the problem of writing a program in some language that prints a copy of itself when evaluated.
 - let's try it with C:
 - `main() {printf("main() {printf("main() ... "); }`
 - how do you complete the program?
 - you have to have some mechanism for escaping out of the infinite recursive definition
 - usually there is some "trick" that is employed that uses some form of "syntactic escape"

2/7/05

4

Syntactic Escape Tricks

- A “quine” is a program that prints an exact syntactic copy of itself.
 - an interesting exercise is to try to write the shortest quine in a variety of languages; for example:
- A C quine in 65 characters:
 - ```
main(a){printf(a="main(a){
printf(a=%c%s%c,34,a,34);} ",34,a,34);}
```
- A Scheme quine in 46 characters:
  - ```
((lambda(x) `( ,x ',x)) '(lambda(x) `( ,x ',x)))
```
- A λ -calculus quine in 14 characters:
 - ```
(λ x.xx)(λ x.xx)
```
- For more examples, see <http://www.nyx.net/~gthomps/quine.htm>

2/7/05

5

## Lambda Calculus

- A formal notation for expressing computable functions
  - lambda definability
    - the class of functions definable in the lambda calculus are exactly the computable functions
    - led to Church to state his famous thesis:
      - $\lambda$ -calculus  $\Leftrightarrow$  Turing Machines  $\Leftrightarrow$  ...
      - Church-Turing thesis

2/7/05

6

# Various Function Notations

## Set Theoretic:

$\{(x,y) \mid \forall x,y \in \mathbb{N}: y = x^2\}$

## Algebraic:

$f: \mathbb{N} \rightarrow \mathbb{N}$

$f(x) = x^2$ ;

## Type-free $\lambda$ -notation:

$\lambda x. x * x$

## Typed $\lambda$ -notation:

$\lambda x:\text{int}. x * x$

## Polymorphic $\lambda$ -notation:

$\lambda x:\alpha. x * x$

## Scheme:

```
(define square
 (lambda (x) (* x x)))
```

2/7/05

## Algol-60:

```
integer procedure square(x); integer x;
begin square := x * x end;
```

## Pascal:

```
function square (x:integer) : integer;
begin square := x * x end;
```

## K&R C:

```
square(x) int x; { return (x * x); }
```

## StdC/C++/Java:

```
int square(int x) { return (x * x); }
```

## ML97:

```
fun square x = x * x;
fun square (x:int) = x * x;
val square = fn x => x * x;
```

## Haskell:

```
square :: Int->Int
square x = x * x
map (\x -> x * x) [0..]
[(x,y) | x <- [0..], y <- [x * x]]
```

7

# Uses of the $\lambda$ -calculus

- A formal notation, theory, and model of computation
- Conceptual foundation for functional programming
  - ISWIM, Lisp, Scheme, ML, Haskell, ...
- Natural model for many programming constructs
  - higher-order functions, variables, block scopes, expressions, ordered pairs, lists, records, recursion
  - calling conventions: call-by-value, call-by-need, call-by-name
- Type inferencing & polymorphic type systems
  - Hindley-Milner type system
- Notation for Scott-Strachey denotational semantics
  - domain theory of Dana Scott

2/7/05

8

## Definitions

- $\lambda$ -calculus is a formal notation for defining functions
  - expressions in this notation are called  $\lambda$ -expressions
  - every  $\lambda$ -expression denotes a function that is “out there” in the platonistic sense
  - a  $\lambda$ -expression consists of 3 kinds of terms:
    - **variables:**  $x, y, z, \text{ etc.}$ 
      - we use  $V, V_1, V_2, \text{ etc.}$ , for arbitrary variables
    - **abstractions:**  $\lambda V.E$ 
      - Where  $V$  is some variable and  $E$  is another  $\lambda$ -term
    - **applications:**  $(E_1 E_2)$ 
      - Where  $E_1$  and  $E_2$  are  $\lambda$ -terms
      - applications are sometimes called combinations

2/7/05

9

## Formal Syntax in BNF

```
< λ -term> ::= <variable>
 | λ <variable> . < λ -term>
 | (< λ -term> < λ -term>)
```

```
<variable> ::= x | y | z | ...
```

Or, more compactly:

```
E ::= V | $\lambda V.E$ | ($E_1 E_2$)
```

```
V ::= x | y | z | ...
```

Where  $V$  is an arbitrary variable and  $E_i$  is an arbitrary  $\lambda$ -expression.

We call  $\lambda V$  the **head** of the  $\lambda$ -expression and  $E$  the **body**.

2/7/05

10

# Variables

- variables can be bound or free
- the  $\lambda$ -calculus assumes an infinite universe of free variables
- they are bound to functions in an *environment*
- they become bound by usage in an abstraction

– for example, in the  $\lambda$ -expression:

$\lambda x. x * y$

x is bound by  $\lambda$  over the body  $x * y$ , but y is a free variable. I.e., lexically scoped. Compare this to scheme:

```
(define z 3)
(define x 2)
(define y 2)
(define multi-by-y (lambda (x) (* x y)))
(multi-by-y z) => 6
```

2/7/05

11

# Abstractions

- if  $\lambda V.E$  is an abstraction
  - V is a bound variable over the body E
  - it denotes the function that when given an actual argument 'a', evaluates to the function E' with all occurrences of V in E replaced with 'a', written  $E[a/V]$
  - For example the abstraction:

$\lambda x. x$

is the identity function that can be applied to either values or other lambda abstractions:

```
($\lambda x. x$) 1 => 1
($\lambda x. x$) a => a
($\lambda x. x$) ($\lambda x. x$) => ($\lambda x. x$)
```

2/7/05

12

# Applications

- If  $E_1$  and  $E_2$  are  $\lambda$ -expressions, so is  $(E_1 E_2)$ 
  - application is basically function evaluation
  - apply the function  $E_1$  to the argument  $E_2$ 
    - $E_1$  is called the **rator** (ope-rator)
    - $E_2$  is called the **rand** (ope-rand)
- For example:

$(\lambda x. xx) 1 \Rightarrow 11$

$(\lambda x. xx) a \Rightarrow aa$

$(\lambda x. xx) (\lambda x. xx) \Rightarrow (\lambda x. xx) (\lambda x. xx)$

this last example is a quine and it doesn't terminate. It keeps duplicating itself ad infinitum. In this example, we don't care, but in real programming we do care about non-terminating evaluations.

2/7/05

13

# Conversion/reduction rules

- $\alpha$ -conversion: any abstraction  $\lambda V.E$  can be converted to  $\lambda V'.E[V'/V]$  iff  $[V'/V]$  in  $E$  is valid
- $\beta$ -conversion ( $\beta$ -reduction): any application  $(\lambda V.E_1) E_2$  can be converted to  $E_1[E_2/V]$  iff  $[E_2/V]$  in  $E_1$  is valid
- $\eta$ -conversion: any abstraction  $\lambda V.(E V)$  where  $V$  has no free occurrences in  $E$  can be converted to  $E$

2/7/05

14

# Conversion rule notation

|                                |                                                   |
|--------------------------------|---------------------------------------------------|
| $E_1 \xrightarrow{\alpha} E_2$ | bound variable renaming to avoid naming conflicts |
| $E_1 \xrightarrow{\beta} E_2$  | like a function call evaluation                   |
| $E_1 \xrightarrow{\eta} E_2$   | elimination of irrelevant information             |

2/7/05

15

# $\alpha$ -redex

|                                |                                                                                                                                             |
|--------------------------------|---------------------------------------------------------------------------------------------------------------------------------------------|
| $E_1 \xrightarrow{\alpha} E_2$ | $\alpha$ -reduction is <u>bound</u> variable renaming applied to an $\alpha$ -redex iff no naming conflicts<br>redex = reducible expression |
|--------------------------------|---------------------------------------------------------------------------------------------------------------------------------------------|

$$\lambda x.x \xrightarrow{\alpha} \lambda y.y \quad (\lambda x.x)[y/x]$$

$$\lambda x.f x \xrightarrow{\alpha} \lambda y.f y \quad (\lambda x.f x)[y/x]$$

$$\lambda x.\lambda y.x+y \xrightarrow{\alpha} \lambda y.\lambda y.f y+y \quad \text{not valid since } y \text{ is already a bound variable in } E_{16}$$

2/7/05

## $\beta$ -redex

$$E_1 \xrightarrow{\beta} E_2$$

$$(\lambda x. f x) E \xrightarrow{\beta} f E$$

$$(\lambda x. (\lambda y. x + y)) \underline{3} \xrightarrow{\beta} \lambda y. \underline{3} + y$$

$$(\lambda y. \underline{3} + y) \underline{4} \xrightarrow{\beta} \underline{3} + \underline{4}$$

2/7/05

17

## Fixed Points

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

2/7/05

18

## Fixed Point Operators

- Consider a fixed point operator  $\text{Fix}$  such that
  - $F (\text{Fix } F) = \text{Fix } F$
  - or equivalently  $\text{Fix } F = F (\text{Fix } F)$ 
    - $\text{Fix}$  is a function that takes any function  $f$  an argument and then repeats the function  $f$  applied to  $(\text{Fix } F)$
    - there are many such fixed point operators
- The “lazy”  $\mathbf{Y}$  combinator is a fixed point operator
  - $\mathbf{Y} = (\lambda f. (\lambda x. f(x x)) (\lambda x. f(x x)))$
  - $\mathbf{Y} F = (\lambda f. (\lambda x. f(x x)) (\lambda x. f(x x))) F$ 
    - $\Rightarrow (\lambda x. F(x x)) (\lambda x. F(x x))$
    - $\Rightarrow F (\lambda x. F(x x)) (\lambda x. F(x x))$
    - $\Rightarrow F (\mathbf{Y} F)$

2/7/05

19

## Using the lazy $\mathbf{Y}$ 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
    - $F = \mathbf{Y} (\lambda f x_1 \dots x_n. \dots f \dots)$
  - for example:
    - $\text{fact } n = \text{if } (n==0) \text{ then } 1 \text{ else } n * \text{fact}(n-1)$
    - Becomes
      - $F = \lambda f n. \text{if } (n==0) \text{ then } 1 \text{ else } n * f(n-1)$
      - $\text{fact} = \mathbf{Y} F$
      - $\text{fact } 2 = (\mathbf{Y} F) 2 \Rightarrow F (\mathbf{Y} F) 2$ 
        - $\Rightarrow F \text{ fact } 2$
        - $\Rightarrow (\lambda f n. \text{if } (n==0) \text{ then } 1 \text{ else } n * f(n-1)) \text{ fact } 2$
        - $\Rightarrow (\text{if } (2==0) \text{ then } 1 \text{ else } 2 * \text{fact}(2-1))$
        - $\Rightarrow (\text{if } (2==0) \text{ then } 1 \text{ else } 2 * (\mathbf{Y} F)(1))$
        - $\Rightarrow (\text{if } (2==0) \text{ then } 1 \text{ else } 2 * ((\lambda f n. \text{if } (n==0) \text{ then } 1 \text{ else } n * f(n-1)) \text{ fact } 1)$
        - $\Rightarrow \dots$

2/7/05

20

## Implementing Fix in Haskell

- The lazy Y combinator can only be implemented in a language that supports lazy (non-strict) evaluation

– Haskell allows this:

```
fix f = f (fix f)
```

```
fact :: Integer->Integer
fact = fix (\f n ->
 if n==0 then 1 else n*f (n-1))
fact 5 => 120
```

2/7/05

21

## More Fix Examples in Haskell

- greatest common divisor

```
gcd :: Integral a => a -> a -> a
gcd = fix (\f x y -> case (abs x, abs y) of
 (x,0) -> x
 (x,y) -> f y (x `rem` y))
```

- map function

```
map :: (a->b) -> [a] -> [b]
map = fix (\f g xs -> case xs of
 [] -> []
 (x:xs) -> g x : f g xs)
```

- foldl function

```
foldl :: (a -> b -> a) -> a -> [b] -> a
foldl = fix (\f g z xs -> case xs of
 [] -> z
 (x:xs) -> f g (g z x) xs)
```

2/7/05

22

## Lazy Y Combinator in Scheme

- We can write the Y combinator in Scheme:

```
(define Y
 (lambda (f)
 ((lambda (x) (f (x x)))
 (lambda (x) (f (x x))))))
```

- but it results in an infinite loop if we try to define a recursive function with it!
- we cannot use the lazy Y combinator in a language with eager (strict) evaluation order
- the problem is with the last lambda expression

2/7/05

23

## Eager Fixed Point Combinator

- Due to Turing
  - $Y_{\text{eager}} = (\lambda x y. y (x x y)) (\lambda x y. y (x x y))$
- We can translate this into Scheme and use it to define (curried) recursive functions:

```
(define Y
 (lambda (f)
 ((lambda (x) (f (lambda (y) ((x x) y))))
 (lambda (x) (f (lambda (y) ((x x) y))))))

(define fact
 (Y (lambda (f)
 (lambda (n)
 (if (zero? n) 1 (* n (f (- n 1))))))))
```

2/7/05

24

## Modified Turing Combinator

- We can simplify the previous definition of Y by recognizing that we don't need to delay the evaluation of the first lambda expression

```
(define T
 (lambda (f)
 ((lambda (x) (f (x x)))
 (lambda (x) (f (lambda (y) ((x x) y)))))))
```

2/7/05

25

## Named Let and Letrec

- In Scheme, the “named let” is used to define a local recursive function:

```
(define fact (lambda (n)
 (let tfact ((n n) (m 1))
 (if (zero? N) 1 (tfact (- n 1) (* m n))))))
```

- Named let is just a macro using letrec, which appears to be a sugared fixed point operator

```
(let name ((var val) ..) exp1 exp2 ...)
((letrec
 ((name (lambda (var ...) exp1 exp2 ...)) name)
 val ...)
```

2/7/05

26

## For Further Study

1. Chris Hankin, *An Introduction to Lambda Calculi for Computer Scientists*, Kings College London Press, 2004.
2. Raymond Smullyan, *Diagonalization and Self-Reference*, Oxford University Press, 1994.
3. Douglas Hofstadter, *Godel, Escher, Bach: An Eternal Golden Braid, 20<sup>th</sup> Anniv Edition*, Basic Books, 1999.
4. Don Blaheta, "The Lambda Calculus", see paper on course website.
5. Richard Gabriel, "The Why of Y," see paper on course website.