Lambda the Ultimate

Corky Cartwright
Stephen Wong
Department of Computer Science
Rice University

Motivation for λ-notation

- Often, functions are used only once
- Examples: arguments to functions like
  - **`map`**,
  - **`filter`**,
  - **`fold`**,
  - and many more "higher-order" functions
- Sometimes we want to build new functions in the middle of a computation.
- **`local`** suffices but it is notationally clumsy for this purpose.
- λ provides simpler, more concise notation

Background

- λ-notation was invented by mathematicians. For example, given
  $f(x) = x^2 + 1$
  what is f? f is the function that maps x to $x^2 + 1$ which we might write as
  $x \to x^2 + 1$
  The latter avoids naming the function. The notation $\lambda x\,.\,x^2 + 1$ evolved instead of $x \to x^2 + 1$
- In Scheme, we write `(lambda (x) (+ (* x x) 1)))` instead of $\lambda x\,.\,x^2 + 1$.
- `(define (f x) (+ (* x x) 1))` abbreviates

  `(define f (lambda (x) (+ (* x x) 1)))`

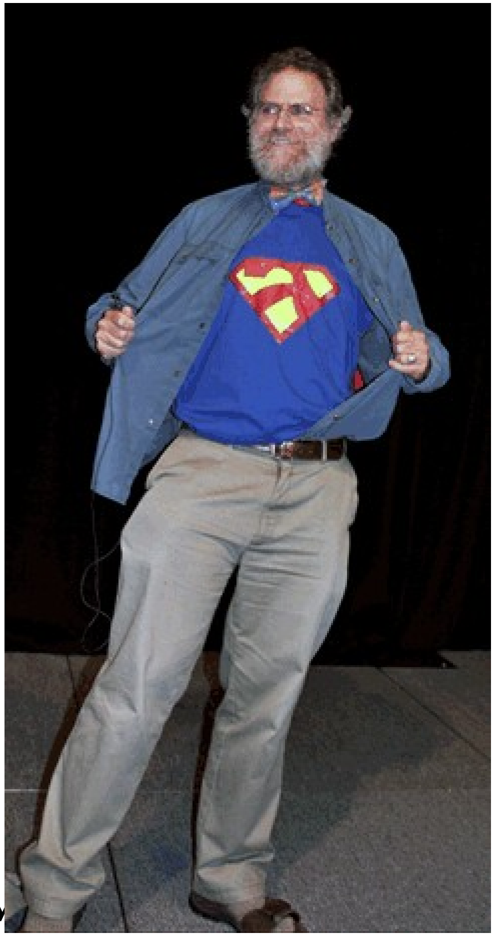
Why λ?

- The name was used by its inventor
  - Alonzo Church, logician, 1903-1995.
  - Princeton University Mathematics Department
  - Introduced lambda in 1930’s in an attempt to formalize mathematics using functions rather than sets

- Church is Corky's academic great-grandfather

Alonzo Church -> Hartley Rogers -> David Luckham -> Corky Cartwright

Many PL researchers are crazy about λ!



Prof. Phil Wadler from University of Edinburgh, Scotland

Scope for a Lambda Abstraction

- Argument scope:
  `(lambda (`*x*$_1$ `...` *x*$_n$`)` *body*`)` introduces the variables *x*$_1$ `...` *x*$_n$ which have *body* as their scope (except holes)
- Example:
  `(lambda (x) (+ (* x x) 1)))`
- Scope for variable introduced by **`define`**. At the top-level,
  `(define` *f* *rhs*`)`

introduces the variable *f* which is visible everywhere (except in holes introduced by local definitions in *f*). Inside
  `(local [(define` *f*$_1$ *rhs*$_1$`) ... (define` *f*$_n$ *rhs*$_n$`)]` *body*`)`
  the variables **`f`**$_1$ `...` **`f`**$_n$ have the **`local`** expression as their scope.

- Recursion comes from **`define`** and not **`lambda`**!
- Challenge: define factorial using only **`lambda if zero? * sub1 1`**

Clear Statement of Challenge

- Define an expression equivalent to

```
(local
    [(define fact
      (lambda (n)
        (if (zero? n) 1
            (* n (fact (sub1 n))))))]
  fact)
```

without using **`define`** or **`local`**

A hard problem

Example

Now we can write the following program

```
(define l '(1 2 3 4 5))
(define a
  (local ((define (square x)
             (* x x)))
    (map square l)))
```

concisely as

```
(define l '(1 2 3 4 5))
(define a (map (lambda (x) (* x x)) l))
```

Careful Definition of Syntax

- Formal specification of what expressions that use lambda can look like:
  - ***exp* = ... | (lambda (*var**) *exp*)**
- Interesting point. λ-abstraction can have
  - Can have multiple arguments
  - Can have no arguments
- Application of a function with no arguments
  - ```
    (define blowup (lambda () (/ 1 0)))
    (blowup)
    ```
    ```

Functions with Zero Arguments?

- We rarely see them in mathemaics
  - A 0-ary function always produces the same result, so it’s just a constant. In logic, constants are often formalized as 0-ary functions.
- In computing, 0-ary functions and constants are not the *same*. We use 0-ary functions:
  - To encapsulate an expression that is evaluated (if at all) on demand.
  - Once we introduce side-effects (destructive modification of data), procedures (the analogs of functions in the world of side effects) of no arguments are common.

`lambda` vs. `local`

- Recall that:

  `(lambda (x1 ... xn) exp)`

  is equivalent to

  `(local [(define (f x1 ... xn) exp)] f)`

- Is **`lambda`** as general as **`local`**? No!
  How do I introduce a recursive function definition using **`lambda`** alone?
  - It can be done but it involves deep, subtle, and messy use of λ-notation (hard challenge, topic in Comp 311). Not very efficient.
  - Direct formulations of recursion rely on the name of the defined function, which **`lambda`** lacks.

Evaluation of λ-expressions

How do we evaluate a λ-expression

(lambda (*x₁* ... *xₙ*) *body*) ?

It's a value!

What about λ-applications?

((lambda (x_1 ... x_n) *body*) V_1 ... V_n) (where)

=> *body*[x_1:=V_1 ... x_n:=V_n] (called β-reduction)

where V_1,...,V_n are values and ***body*[x_1:=V_1 ... x_n:=V_n]** means *body* with x_1 replaced by V_1, ..., x_n replaced by V_n.

Examples:

```
((lambda (x) (* x 5)) 4) => (* 4 5) => 20

  ((lambda (x) (x x)) (lambda (x) (x x)))
=> ((lambda (x) (x x)) (lambda (x) (x x)))
=> ... (cool?)
```

More Examples

```
    ((lambda (x y z) (+ x y z)) 1 2 3)
=> (+ 1 2 3)

    (((lambda (x) (lambda (y) (+ x y))) (* 2 3)) 4)
=> (((lambda (x) (lambda (y) (+ x y))) 6) 4)
=> ((lambda (y) (+ 6 y)) 4)
=> (+ 6 4)
=> 10
```

Fine Points of Substitution

- Only the *free* occurrences of a variable are replaced. A variable occurrence **`v`** in an expression **`E`** is *free* iff it does not refer to a variable bound in **`E`**. A n*on-free* (*bound*) variable occurrence **`v`** in expression **`E`** must be embedded in a **`local`** scope (defined by a **`lambda`** or a **`local`**) within **`E`**.
- Examples:
  - Neither occurrence of **`x`** is free in **`(lambda (x) x)`**
  - Neither occurrence of **`x`** is free in **`(local [(define x 12)] x)`**
  - **`x`** is free in **`(+ y x)`**
  - **`x`** is free in **`(lambda (y) (+ y x))`**
  - Only the first occurrence of **`x`** is free in **`((+ x (local [(define x 12)] (* x 13))`**

Fine Points of β-reduction

- In the context of the Scheme evaluation, the simple rules we have already given tell the whole story.
- β-reduction is a general transformation rule in the world of functional programming. In β-reductions

  ((lambda (*x_1* ... *x_n*) *M*) N_1 ... N_n)

  => *body*[*x_1*:=*v_1* ... x_n:=v_n]

ugly things can happen when **N_1 ... N_n** contain free variables. (In Scheme evaluation, values *never* contain free variables.)

Nesting λ

```
   (lambda (x) (lambda (y) (+ (* x y) (* 4 5)))
=> (lambda (x) (lambda (y) (+ (* x y) (* 4 5)))

   ((lambda (x) (lambda (y) (+ x 1))  5)
=> (lambda (y) (+ 5 1))

   ((lambda (x) (lambda (x) (+ x 1))  5)
=> (lambda (x) (+ x 1))

   ((lambda (x) (lambda (y) (y x))) (lambda (z) (+ y z)))
=> (lambda (y) (y (lambda (z) (+ y z))))
```

which is WRONG! This mistake and the change in the meaning/scope of **y** is called "capturing a bound variable". This terminology is a bit misleading because the free variable **y** is captured (becoming bound) in the erroneous transformation. We should say "capturing a *free* variable".

Safe Substitution

To salvage the correctness of β-reduction in the general case, we must stipulate that the rule uses safe substitution, where safe substitution renames local variables in the code body that is being modified by the substitution to avoid capturing free variables in the argument expression that is being substituted.

```
    ((lambda (x) (lambda (y) (y x)))   (lambda (z) (+ y z)))
 => ((lambda (x) (lambda (f) (f x)))   (lambda (z) (+ y z)))
 => (lambda (f) (f (lambda (z) (+ y z))))
```

We will hold you responsible on exams for understanding either safe substitution or the subtleties of β-reduction when the argument expressions contain free variables.

When Should I Use a Lambda?

- It makes sense to use a lambda instead define when
  - the function is not recursive;
  - the function is needed only once; and
  - the function is either
    - being passed to another function, or
    - being returned as the final result (contract returns “->”)
- Note: It is hard to read code when lambda is used at the head of an application
  - `((lambda (x) (* x x)) (+ 13 14))`
- We can rewrite this as:
  - `(local ((define x (+ 13 14)))`
    `(* x x))`

Lambda Becoming Pervasive in PL

Python

By popular demand, a few features commonly found in functional programming languages and Lisp have been added to Python [...]

– Guido van Rossum, 4.7.4 Lambda Forms, Python Tutorial

but very badly. Ask any functional programmer about Python and they will either say “What is Python” or laugh.

Java

Perhaps in Java 8. Major controversy over how comprehensive the construct should be.

For Next Class

- Homework due next Monday!
- Continue Reading and Reviewing:
  - Ch 21-22: Abstracting designs and first class functions