Lambda the Ultimate and Reduction Semantics

Corky Cartwright
Department of Computer Science
Rice University

Motivation for λ-notation

- In most functional languages, functions are data values. Origin: λ-calculus 1930's (Alonzo Church)
- 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

Basic Idea

- 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
  $x \rightarrow x^2 + 1$
  The latter avoids naming the function. The notation
  $\lambda x . x^2 + 1$ evolved instead of $x \rightarrow x^2 + 1$
- 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, NJ
  - Introduced lambda in 1930's to formalize mathematical proofs

Church is my academic great-grandfather

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

Scope for a Lambda Abstraction

- Argument scope: `(lambda (x1 ... xn) body)` introduces the variables x_1 ... x_n which have *body* as their scope (except for 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 inside holes introduced by local definitions of `f`). Inside
  ```
  (local [(define f1 rhs1) ... (define fn rhsn)) body)
  ```
- the variables f_1 ... f_n have the entire `local` as their scope.
- Recursion comes from `define` not `lambda`! It is possible to define recursive functions solely using `lambda` (and whatever primitive operations that appear in a `define` but it is surprisingly hard.

Some PL researchers are crazy about λ!

Prof.

Phil Wadlerb

University of Edinburgh

Example

Now we can write the following program concisely

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

as

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

Careful Definition of Syntax

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

Reduction Semantics

- Simple Reduction Semantics: Essence of Functional Programming
- Idea: Evaluation of expressions is a familiar idea from grammar school.
- Grammar school:
  evaluate parenthesized arithmetic expressions
- Functional programming:
  evaluate arbitrary (functional program) text

Synopsis

- Value are values are values …
- A value evaluates to itself so we stop evaluation when we reduce our original expression to a value.
- In most functional languages, always perform leftmost reductions because order matters

Evaluation of λ-expressions

- How do we evaluate a λ-expression

  `(lambda (`x_1 `...` x_n`)` *body*`)`

  It's a value!
- What about λ-applications?

  `((lambda (`x_1 `...` x_n`)` *body*`)` v_1 `...` v_n`)`

⇒ *body*[$x_1 \leftarrow v_1$ ... $x_n \leftarrow v_n$] (called β-reduction)

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)))
=> ((lambda (x) (x x)) (lambda (x) (x x)))
```

Capture!

```
...
((lambda (x) (lambda (y) (y x))) bb
   (lambda (z) (+ y z)))

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

WRONG!

The meaning of `y` has changed! But it can *never* happen in the evaluation of Racket program text if `lambda` is the only binding construct. Racket never reduces inside a lambda.

Safe Substitution

- Must rename 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))))
```

Comprehensive Reduction Rules

- The document LawsofEval.pdf is a comprehensive description of the reduction semantics of functional Racket.

  You need to understand it in detail. We will briefly review it now.