Clever Programming With Functions

Prof. Robert “Corky” Cartwright
Department of Computer Science
Rice University

Using Functions to Represent Objects

- How can we represent a pair in Scheme so that the only operations that code can perform on pairs are:

```
(make-pair x y)
(pair-first p)
(pair-second p)
(pair-equal? p1 p2)
```

- What if we represent a pair as a list? As a struct? Structs are not as robust as you might think. In the advanced language level try:

```
(define-struct Pair (first second))
(define p (make-Pair 1 2))
(set-Pair-first! P 17)
p
```

Objects as closures

```
(define (make-pair x y)
  (lambda (msg)
    (cond [(equal? msg 'first) (lambda () x)]
          [(equal? msg 'second) (lambda () y)]
          [(equal? msg 'equal)
           (lambda (p)
             (and (equal? (pair-first p) x)
                  (equal? (pair-second p) y)))])))
(define (pair-first p) ((p 'first)))
(define (pair-second p) ((p 'second)))
(define (pair-equal? p1 p2) ((p1 'equal) p2))
```

This representation trick is very important. It shows how closures (functions with free variables treated as first-class data values) can be used to represent abstract (black-box) data types.

Useful Functionals

- What is a *functional*? A function that takes a function as a argument and often returns a function. The differential and integral operators in calculus are functionals.

- Important functionals in functional programming:

  `map filter foldr foldl curry`

The Idea Behind `curry`

- Every function of the form

  `A  B -> C`

  can be converted to a function of type

  `A -> (B -> C)`

  which is often more convenient.
- In set theory, here is an isomorphism between

  $A \times B \to C$

  and

  $A \to B \to C$
- This correspondence *roughly* holds for programming language types.

A Simple Example

```
map : A            B   ->   C
map :  (X - >Y) (list-of X) -> (list-of Y)

map' : A    -> (B   ->   C)
map' :  (X -> Y) -> (list-of X) -> (list-of Y)
```

Standard Map

```
(define map
  (lamba (f l)
    (cond
      [(empty? l) empty]
      [else
        (cons (f (first l))
                 (map f (rest l)))]))
```

Curried Map

- A definition in terms of **`map`**

```
(define (map′ f)
  (lambda (l) (map f l)))
```

- When written from scratch, it looks almost exactly like **`map`**:

```
(define map′
  (lambda (f)
    (lambda (l)
      (cond
        [(empty? l) empty]
        [else (cons (f (first l))
                    (map f (rest l)))]))))
```

Can We Define a Functional that Curries?

Unfortunately, we need a separate curry function for each function arity >= 2.

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

Uncurry

- Question: See if you can write

  `uncurry : (A -> (B -> C)) -> (A  B -> C)`

- Note the equational properties:

  ```
  curry (uncurry f) = f
  uncurry (curry f) = f
  ```

- These are laws in mathematics, but the first fails in programming languages even when **`f`** is restricted to a value. It doesn't hold in either CBN or CBV. Why?
  The left-hand side never throws an exception or diverges on the first application.
- Both equations fail if **`f`** can be an expression rather than a value of the appropriate type. Why? The evaluation of the left hand side never diverges or generates an exception.
- Yet these equations are widely taught by PL experts as if PL domain theory was set theory. They are NOT identities for PL code!

The Crux of The Difference

- Why don't functional languages obey standard laws from set theory?
- *Eta*-conversion fails in the PL world which admits divergent definitions.
- *Eta*-conversion is often added as an axiom to the λ-calculus. It does not disturb the major properties.
- *Eta*-conversion asserts:
  $\lambda x.\ Ex = E$ (where x does not occur free in E)
- It fails even when the type of E is restricted to a unary function type.

Bonus Material
Another Important Functional: **Y**

- lambda-notation (as in Scheme) indirectly supports recursion. How? A clever construction based on sophisticated mathematics (lambda-calculus).
- Short story: solutions to recursion equations are "fixed points". Given the equation
  $f(x) = E_{f,}$ (which is equivalent to $f = \lambda x.\ E_f$)
  what is the least solution f^*? Under proper conditions,
  $f^* =$ **lub** $F^i(\perp)$
  where $F(f) = \lambda f.\lambda x.\ E_f)$) and $\perp$ is the least-defined function (i.e., the function denoted by Ω). **Y** is defined by **Y**$(F) = f^*$ where (f^* is least solution of $F(f^*) = f^*$).

Defining Y

- λ-calculus programming trick: use a variation on

  $\Omega = (\mathit{self}\ \mathit{self})\ ^k[(\lambda x.\ f(x\ x))\ (\lambda x.\ f(x\ x))] = \ldots(\text{self self})$

  $(\lambda x.\ f(x\ x))\ (\lambda x.\ f(x\ x)) = f[(\square x.\ f(x\ x))\ (\lambda x.\ f(x\ x))]$

  $= f^2[(\lambda x.\ f(x\ x))\ (\lambda x.\ f(x\ x))] = \ldots$

  $= f^k[(\lambda x.\ f(x\ x))\ (\lambda x.\ f(x\ x))] = \ldots$
- In CBN languages

  $\mathbf{Y} = \lambda f\ .\ (\lambda x.\ f(x\ x))\ (\lambda x.\ f(x\ x))$
- CBV is slightly harder and messier because $\mathbf{Y}F$ does not terminate. Trick: convert the term $(\lambda x.\ f(x\ x))$ to $(\lambda x.\ [\lambda y.f(x\ x))]y)$ (eta-conversion of the diverging term).
- Default for λ-calculus is CBN. Default for programming languages is CBV.

Bonus Material: Other Powerful Functionals: S,K

- Every closed λ-expression can be written without **any** variables given the three primitive functionals **S**, **K**, **I** where
  - **S** = $\lambda x.\lambda y.\lambda z.\ (xz)(yz)$
  - **K** = $\lambda x.\lambda y.\ X$
  - **I** = $\lambda x.x$
- In fact, you only need two because
  - **I** = **S**(**KK**)
- Functionals defined by closed λ terms (and nothing else are called **combinators**. **Y** (in all its varieties) is a combinator.

For Next Class

- Homework due Friday
- Review of Scheme material in lecture on Wednesday and Friday.

- Reading:
  - Review for coming exam which will be distributed on Friday.