Functions as Values

Corky Cartwright
Department of Computer Science
Rice University
Functional Abstraction

• A powerful tool
  • Makes programs more concise
  • Avoids redundancy
  • Promotes “single point of control” (no code duplication)

• Generally involves polymorphic contracts (contracts containing type variables)

• What we cover today for lists applies to any recursive (self-referential) type
Look for the pattern

- One function:

  ; add1-each : (list-of number) -> (list-of number)
  ; adds one to each number in list
  (define (add1-each l)
    (cond [(empty? l) empty]
          [else
           (cons (add1 (first l))
                 (add1-each (rest l)))])))
Look for the pattern

Another function function:

; not-each : (listOf boolean) -> (listOf boolean)
; complements each boolean in the list
(define (not-each l)
  (cond [(empty? l) empty]
    [else (cons (not (first l))
                (not-each (rest l)))]))
Codify the pattern

Abstracting with respect to `add1`, `not`, and the element type in the lists:

; \textit{map} : (X -> X), (listOf X) -> (listOf X)
; \textit{applies f to each element in l}

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

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

```lisp
(define (map f l)
    (cond [(empty? l) empty]
          [else (cons (f (first l))
                      (map f (rest l))))])
```
Generalize the pattern

Do all occurrences of \( x \) in contract of \( \text{map} \) need to be of the same type?

; \text{map} : (X -> Y) \ (\text{list-of X}) -> (\text{list-of Y})
; \ (\text{map} \ f \ l) \text{ returns the list consisting of } f
; \text{applied to each element in } l

(define (\text{map} \ f \ l)
  (cond [(\text{empty?} \ l) \text{empty}]
    [else (\text{cons} \ (f \ (\text{first} \ l))
      (\text{map} \ f \ (\text{rest} \ l))))]))
Tip on Generalizing Types

- When we generalize, we **only** replace
  - specific types (like `number` or `symbol`)
  - by type variables (like `x` or `y`)
- **We never** replace a type by the `any` type, which actually means
  `number` | `boolean` | `list-of number` | `list-of ...` | `number -> number` | ...
- What goes wrong if we use `any`? We cannot `instantiate` (bind) `any` as a custom type.
Use the pattern

- **map** can be used with *any* unary function.
- `(map not l)`
- `(map sqr l)`
- `(map length l)`
- `(map first l)`
- `(map symbol? l)`
- Note: other recursive data types also have maps!
More about \texttt{map}

- Powerful tool for parallel computing!

- Has elegant properties (from mathematics):
  - \((\text{map } f (\text{map } g l)) = (\text{map } (\text{compose } f g) l)\)
  - Soon we will see how to define \texttt{compose}

- For fun: Checkout Google’s “map/reduce”
Better notation for function values

- Assume we want to square all of the elements in a list. How can we do using \texttt{map} in a compact expression? We need simple notation for denoting new functions without using \texttt{local}. Alonzo Church invented such an notation in the 1930's called \textit{lambda}-notation. In Church's scheme \[ \lambda x. M \]
denotes the function \( f \) defined by the equation \( f(x) = M \).

- Lisp (the progenitor of Scheme) adopted this notation for new functions. In particular,

\[
(\text{lambda} \ (x_1 \ldots x_n) \ E)
\]
denotes the function \( f \) defined by:

\[
(\text{define} \ (f \ x_1 \ldots x_n) \ E)
\]
Examples of \texttt{lambda}

\begin{verbatim}
;; square the elements in a list
(map (lambda (x) (* x x)) '(1 2 3 4))

;; compose: \((Y \rightarrow Z) (X \rightarrow Y) \rightarrow (X \rightarrow Z)\)
(define (compose f g) (lambda (x) (f (g x))))
(map (compose add1 sub1) '(1 2 3 4))
\end{verbatim}

Expressing \texttt{lambda} using \texttt{local}

Straightforward, but ugly

\begin{verbatim}
(lambda (x_1 \ldots X_n) M) =>
(local [(define (new-v x_1 \ldots x_n) M)] new-v)
\end{verbatim}
Templates as functions

• Recall the template for lists:
  
  (define (fn l)
  
  (cond
    [(empty? l) ...]
    [else ... (first l)
    ... (fn (rest l))
    ...])

• Can we construct a function foldr that takes the "..." for empty? and the "..." for else as parameters init and op? Yes. The op parameter must be a function because it must process (first l) and (fn (rest l)).
Templates as functions

It would look just like this:

```scheme
;; the contract is not obvious;
(define (foldr op init l)
  (cond [(empty? l) init]
        [else
         (op (first l)
              (fold op init (rest l)))]))
```

Can we express all functions we’ve written using `foldr`?
Can we write \texttt{map} in terms of \texttt{foldr}?

\begin{verbatim}
map : (X->Y) (listOf X) -> (listOf Y)
(define (map f l)
  (foldr (lambda (x l)(cons (f x) l))
         empty
         l))
\end{verbatim}
What is the type of foldr?

;; foldr: (X  Y -> Y)  Y  (list-of X) -> Y
;; (foldr op init (list el ... en)) returns
;; (op el ( ... (op en init) ... )) which is
;; (el op ( ... (en op init) ... )) in infix notation

Reasoning: in (foldr op init alox), alox is a list-of X, where X is determined by the value of alox. op is applied to (first l) and (foldr op init (rest l)), implying op has inputs e and y of type X and Y.
For Next Class

• Homework due next Friday. Don't dally.

• Reading:
  • Ch 21-22: Abstracting designs and first class functions