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”
- 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:

```
; add1Each : number-list -> number-list
; adds one to each number in list
(define (add1Each l)
   (cond [(empty? l) empty]
         [else
            (cons (add1 (first l))
                  (add1Each (rest l)))]))
```

Look for the pattern

Another function:

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

Codify the pattern

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

```
; map : (X -> X), X-list -> X-list
; applies f to each element in l
(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 map need to be of the same type?

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

(define (map f l)
  (cond [(empty? l) empty]
        [else (cons (f (first l))
                    (map f (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` | `number-list` | `boolean-list` | `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 sqrt l)`
- `(map length l)`
- `(map first l)`
- `(map symbol? l)`

Note: Other recursive data types also have maps!

More about `map`

- Powerful tool for parallel computing!
- Has elegant properties (from mathematics):
  - `(map f (map g l)) = (map (compose f g) l)`
  - Soon we will see how to define `compose`
- For fun: Checkout Google's "map/reduce"

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:

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

- Can we express all functions we've written using `foldr`? What is `foldl`? `foldr` is right-associative. `foldl` is left-associative.
- How can we compute `foldl` efficiently?

map in terms of foldr

Can we write map in terms of foldr? Yes.

```
map : (X->Y) X-list -> Y-list
(define (map f l)
  (foldr (lambda (x l)(cons (f x) l))
         empty
         l))
```

What is the type of `foldr`?

```
foldr: (X  Y → Y)  Y  X-list → Y
(foldr op init (list e1 .. en))
     = (op e1 ( .. (op en init) .. ))
     = e1 op ( .. (en op init) .. ))  [infix]
```

Reasoning: in `(foldr op init l)`, `l` is an `X-list`, where `X` is determined by the value of `l`. `op` is applied to `(first l)` and `(foldr op init (rest l))`, implying `op` has inputs `e1` and `y` of type `X` and `Y`.

If `op` is a group operation, then `init` is the identity.

What is the type of `foldl`?

```
foldl: (X  Y → Y)  Y  X-list → Y
(foldl op init (list e1 .. en))
     = (op en ( .. (op e1 init) .. ))
     = (..((e1 op init) op e2).. op en)  [infix]
```

Reasoning: in `(foldl op init l)`, `l` is an `X-list`, where `X` is determined by the value of `l`. `op` is initially applied to `(first l)` and `init`, implying `op` has inputs `e1` and `y` of type `X` and `Y`.

If `op` is a group operation, then `init` is the identity.

How does foldl process elements in reverse order?

Key Insight: Use a help function with an accumulator.

Unexpected Payoff; the help function is tail-recursive which can be critical in processing long lists.

Constraint: since elements are processed in reverse order, any order dependence in the accumulated answer is reversed. In some cases, like the example below, the accumulated answer is a list where order does matter, reversal of the initial singleton lists is inconsequential in bottom-up mergeSort, which first creates a list of singleton lists using an auxiliary function drop. The naïve coding of this function has catastrophic behavior on long input lists.

Example:

```
drop: alpha-list -> alpha-list-list
(define (drop (loa)
   (if (empty? loa) empty (cons (list (first loa)) (drop (rest loa))))
(check-expect (drop ‘(1 2 3)) ‘((1) (2) (3)))
```

What is the Help Function for Drop

Insight: a help function that processes list elements in left-to-right order relies on an accumulator parameter to hold the accumulated answer which is returned when all of the elements in the list have been processed.

We can write such a help function for `drop` recognizing that the version relying on a tail-recursive help function will reverse the order of the resulting list.

```
dropHelp: alpha-list alpha-list-list -> alpha-list-list
(define (dropHelp loa accum)
   (if (empty? loa) empty
       (dropHelp (rest loa) (cons (list (first loa)) accum))))
drop: alpha-list -> alpha-list-list
(define (drop loa) (dropHelp loa empty))

(check-expect (drop ‘(1 2 3)) ‘((3) (2) (1)))
```

Comparing foldr and foldl

- Efficiency: `foldl` is better both in space (where the difference is enormous [small constant vs. linear!]) and time (where the difference is modest because tail calls [jumps!] are cheaper to execute than conventional function calls) *at the cost of processing the elements in reverse order.* For very long input lists, `foldr` may be unacceptable.
- Semantics: performing the aggregation operation (the function parameter) in reverse order may or may not affect the answer. For associative operations, by definition, it does not matter. But the aggregation operations passed to with `foldr`, `foldl` may not be associative. For example, what happens to `map` if we use our definition based on `foldr` and replace `foldr` with `foldl`? The resulting list is reversed!