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) via type abstraction
- What we cover today for lists applies to *any* recursive (self-referential) type

Look for the repeated pattern

First 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

Second 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), (list-of X) -> (list-of X)
;; 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
;; contract: (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` to a specific 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 like various forms of trees also have mapping operations, but they are not generally in the Racket library. You must write them If needed.

More about `map`

- Powerful tool for parallel computing!
- Has elegant properties (from mathematics):
  - `(map f (map g l)) = (map (compose f g) l)`
  - We have already seen how to define `compose`

- For fun: Checkout Google's "map/reduce"

Templates as functions

Recall the template for processing lists using structural recursion:

```
; (define (list-fn l)
;  (cond
;    [(empty? l) ...]
;    [else ... (first l)
;          ... (list-fn (rest l))
;          ...]))
```

Can we construct a function **`foldr`** that takes the "…" (which *must* be constant) for **`empty?`** and the operation (function) "… … …" for **`else`** as parameters **`init`** and **`op`**? Yes! The **`op`** parameter is a binary function that takes **`(first l)`** and **`(list-fn (rest l))`** as arguments. Note that **`(list-fn (rest l))`** is **`init`** when **`l`** has one element.

Templates as functions

```
;; foldr: (s t -> t) t (list-of s) -> t
;; contract: given list l = (e1 ... en), (foldr op init l) =
;; (op e1 (op e2 ...  (op en init) ... ))
  (define (foldr op init l)
     (cond [(empty? l) init]
           [else
             (op (first l)
                 (foldr op init (rest l)))]))
```

- **`Infix formula for (foldr op init (list e1 ... en)) =`**
  `e1 op (e2 op ... (en op init))`
- Often the types **`s`** and **`t`** are the same.
- Can we express all functions we've written instantiating our template using **`foldr`**? Yes! Is there a **`foldl`** as well? Yes, but **`foldr`** is right-associative, while **`foldl`** is left-associative which means the contract is slightly different.
- How can we compute **`foldl`** efficiently? (Hint: use tail recursion)

`map` in terms of **`foldr`**

Can we write `map` in terms of **`foldr`**? Yes!

```
map : (X -> Y) (list-of X) -> (list-of Y)
Contract: given f: (X -> Y) and l = (list e1 ... en): (list-of Y),
(map f l) returns (list (f e1) ... (f en))

(define (map f l)
   (foldr (lambda (x l) (cons (f x) l))
          empty
          l))
```

Analysis of the Type of `foldr`

`foldr: (X  Y → Y)  Y  (list-of X) → Y`

`(foldr op init (list e1 .. en))` computes
`(op e1 ( .. (op en init) .. ))` corresponding to
`e1 op ( .. (en op init) .. )` in infix notation

Comments:

- The `map` example is confusing because `Y` is a `list` type.
- In `(foldr op init l)`, `l` is a `(list-of X)`, 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`.
- Connection to abstract algebra: if `op` is a group operation, then `init` is the identity.

Explication of foldl

```
;; foldl-help: (X Y → Y) (list-of X) Y → Y
;; Contract: given op, l = (list e1 ... en), and accum,
;; (foldl-help op l accum) = (op en (op en-1 ...  (op e1 accum) ... ))
;; = [in infix notation] ((... ((e1 op accum) op e2) ... op en-1) op en)
(define (foldl-help op l accum)
  (if (empty? l) accum)
      (foldl-help op (rest l) (op (first l) accum))))

;; foldl: (X Y → Y) Y (list-of X) → Y
;; Contract: given op, init, and l = (list e1 ... en),
;; (foldl op init l) = (op en (op en-1 ...  (op e1 init) ... ))
;; = [in infix notation] (e1 op ... (en-1 op (en op init)) ... )b
(define (foldl op init l) (foldl-help op l init))
```

Note: `foldl-help` above is identical to `foldl` except for argument order, so there is no need for a help function. We can fuse `init` with `accum`

Explication of `foldl`

```
;; foldl: (X Y → Y) Y (list-of X) → Y
;; Contract: given op, init, and l = (list e1 ... en),
;; (foldl op init l) returns (op e1 ( .. (op en init) .. )), which is
;; e1 op ( .. (en op init) .. ) in infix notation

(define (foldl op init l)
  (if (empty? l) init)
      (foldl op (op (first l) init) (rest l)))))
;; Note that the second argument in the recursive call is an accumulator.
```

Comments:

- What a hack!
- The type of **`foldl`** is the same as the type of **`foldr`**. Why? The only difference between **`foldr`** and **`foldl`** is the the association of the elements in the **`list l`**.
- Note: in some functional languages like Haskell, **`foldl`** reverses the order of the arguments for the `op` parameter. I dislike this convention (it breaks the simple connection between **`foldr`** and **`foldl`**) but Haskellites vigorously defend it.

Example comparing **`foldr`** and **`foldl`**

Key Insight: **`foldl`** effectively uses a help function with an accumulator.

Payoff; the help function is tail-recursive which is much more space efficient in processing long lists (constant space instead of linear space).

Constraint: since elements are processed in reverse order, the ordering of the accumulated result is reversed. **`foldl`** is preferred to **`foldr`** when this change in ordering does affect the correctness of app contracts. This condition clearly holds when the binary function passed to the fold operation is associative. Similarly, in bottom-up **`mergeSort,`** reversing the list of initial singleton lists is inconsequential. The naïve coding of **`drop`** behaves catastrophically on long input lists.

Example:

```
;; drop: (list-of alpha) -> (list-of (list-of alpha))
(define (naïve-drop (loa) (foldr (lambda (e l) (cons (list e) l)) loa))
(define (opt-drop loa) (foldl (lambda (e l) (cons (list e) l)) loa))
(check-expect (naïve-drop ‘(1 2 3)) ‘((1) (2) (3)))
(check-expect (opt-drop ‘(1 2 3)) ‘((3) (2) (1)))
```

What is equivalent direct code for **`opt-drop`**?

We can directly write efficient code for **`opt-drop`** by using a help function reverses the ordering of the list forming the computed result.

```
;; drop-help: (list-of alpha) (list-of (list-of alpha)) -> (list-of (list-of alpha))
(define (drop-help loa accum)
   (if (empty? loa) empty
       (drop-help (rest loa) (cons (list (first loa)) accum))))
;; drop: alpha-list -> alpha-list-list
(define (opt-drop loa) (drop-help loa empty))
```

Alternatively

```
(define opt-drop (foldl (lambda (init e) (cons (list e)
(check-expect (opt-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 **`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 result list is reversed!