Rewriting your function using map and foldr

CS 5010 Program Design Paradigms "Bootcamp"

Lesson 5.5

© Mitchell Wand, 2012-2014
This work is licensed under a Creative Commons Attribution-NonCommercial 4.0 International License.

Introduction

- In this lesson, we will learn to recognize when your function is suitable for replacement by one of the built-in HOFs (**map**, **filter**, **foldr**, etc.)

Learning Objectives

- At the end of this lesson you should be able to:
  - Use the built-in HOFs for processing lists found in ISL.
  - Recognize the HOF that's appropriate for your function.
  - Follow a recipe for converting your use of a template into a use of a HOF.
  - Follow a recipe for defining a new function using a HOF.

Introduction

- There are many ways to generalize functions built using the **ListOfX** template. The textbook refers to these as *the list abstractions*.
- We prefer the word *generalization*, since abstraction can mean many things.
- Chapter 18 of HtDP/2e gives a helpful list of the built-in list abstractions in ISL.
- We've seen most of these before, but let's look at them all together.

Pre-built HOFs for lists (1) (Chapter 18)

```
;; map : (X -> Y) ListOfX -> ListOfY
;; construct a list by applying f to each item of the given
;; list.
;; that is, (map f (list x_1 ... x_n))
;;              = (list (f x_1) ... (f x_n))
(define (map f alox) ...)

;; foldr : (X Y -> Y) Y ListOfX -> Y
;; apply f on the elements of the given list from right to
;; left, starting with base.
;; (foldr f base (list x_1 ... x_n))
;;   = (f x_1 ... (f x_n base))
(define (foldr f base alox) ...)
```

The book doesn't use the GIVEN/RETURNS form for purpose statements. But you still need to do so for the functions you write!

Pre-built HOFs for lists (2) (Chapter 18)

```
;; build-list : NonNegInt (NonNegInt -> X) -> ListOfX
;; construct (list (f 0) ... (f (- n 1)))
(define (build-list n f) ...)

;; filter : (X -> Boolean) ListOfX -> ListOfX
;; construct the list from all items on alox for which p
;; holds
(define (filter p alox) ...)
```

Pre-built HOFs for lists (3) (Chapter 18)

```
;; andmap : (X -> Boolean) ListOfX -> Boolean
;; determine whether p holds for every item on alox
;; that is, (andmap p (list x_1 ... x_n))
;;          = (and (p x_1) ... (p x_n))
(define (andmap p alox) ...)

;; ormap : (X -> Boolean) ListOfX -> Boolean
;; determine whether p holds for at least one item on alox
;; that is, (ormap p (list x_1 ... x_n))
;;          = (or (p x_1) ... (p x_n))
(define (ormap p alox) ...)
```

Which of these should I use?

- We can write a recipe to help you decide which of these functions to use, if any.

Recipe for Rewriting Your Function to Use the Pre-Built HOFs for lists

Recipe for Rewriting your function to use the Pre-Built HOFs for lists
1. Start with your function definition, written using the template for some list data.
2. Determine whether the function is a candidate for using one of the built-in HOFs
3. Determine which built-in HOF is appropriate
4. Rewrite your function using the built-in abstraction. The new strategy is "Use HOF ..."
5. Comment out the old definition. Do not change the contract, purpose statement, examples, or tests.

A candidate for **map** or **foldr** looks like this:

```
(define (f lst a b c)
  (cond
   [(empty? lst) ...]
   [else (...
            (first lst)
            (f (rest lst) a b c))]))
```

takes a list and some other arguments

recurs on the rest of the list; other arguments don't change

Here are the things that make your use of a template a candidate for one of the list abstractions.

A candidate for **map** looks like this:

```
;; f : ListOfX ...  -> ListOfY
(define (f lst a b c)
 (cond
  [(empty? lst) empty]
  [else (cons
          (... (first lst) a b c)
          (f (rest lst) a b c))]))
```

Function takes a list and some other arguments, and returns a list

empty here

cons here

A candidate for **andmap** looks like this:

```
;; f : ListOfX ... -> Bool
(define (f lst a b c)
 (cond
  [(empty? lst) true]
  [else (and
        (... (first lst) a b c)
        (f (rest lst) a b c))]))
```

Function takes a list and some other arguments, and returns a boolean

true here

and here

A candidate for **ormap** looks like this:

```
;; f : ListOfX ... -> Bool
(define (f lst a b c)
  (cond
    [(empty? lst) false]
    [else (or
            (... (first lst) a b c)
            (f (rest lst) a b c))]))
```

Function takes a list and some other arguments, and returns a boolean

False here

or here

A candidate for **`foldr`** looks like this:

```
;; f : ListOfX ... -> ??
(define (f lst a b c)
 (cond
  [(empty? lst) ...]
  [else (...
          (first lst)
          (f (rest lst) a b c))]))
```

and none of the above patterns (**map**, **andmap**, **ormap**) apply.

A candidate for **filter** looks like this:

```
;; f : ListOfX ... -> ListOfX
(define (f lst a b c)
  (cond
    [(empty? lst) empty]
    [else (if (... (first lst) a b c)
              (cons
                (first lst)
                (f (rest lst) a b c))
              (f (rest lst) a b c))]))
```

Function takes a list and some other arguments, and returns a list of the same type.

makes a decision based on the first element of the list

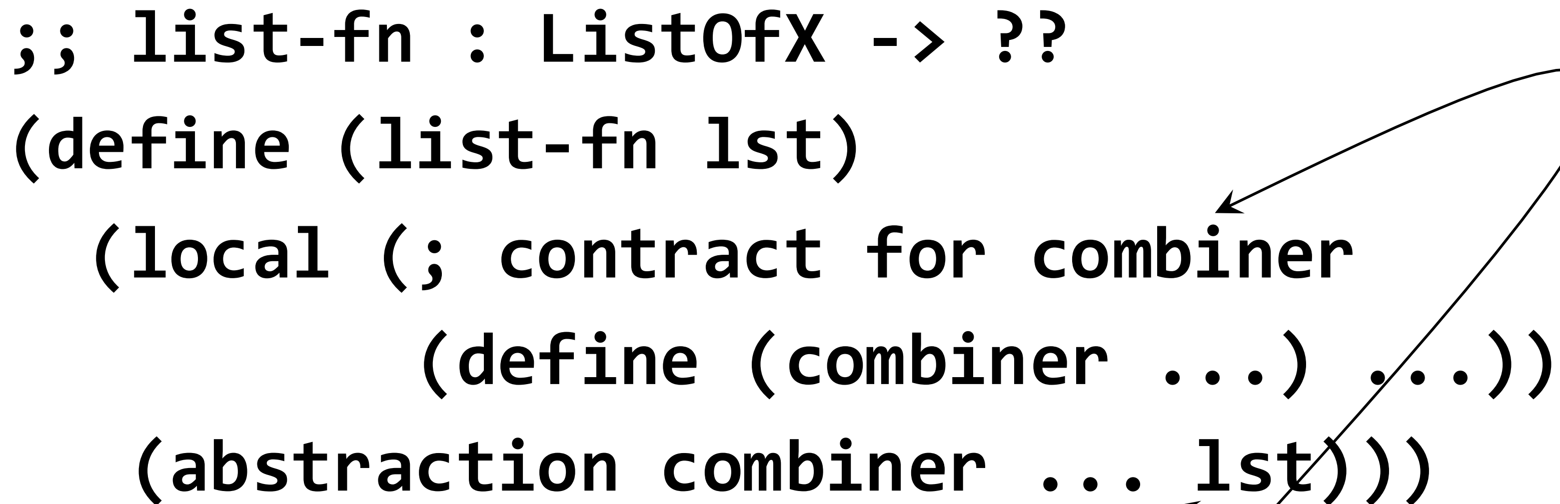
if test is true, includes the first element in the answer

recurs on the rest of the list; other arguments don't change

Patterns for using higher-order functions

- We've looked at which uses of the list template might be rewritten using higher-order functions.
- Next we'll look at what the function will look like when it's rewritten.
- The exact form of the rewritten definition will be different for each of the abstraction functions (**map**, **filter**, **foldr**, etc.). Lets look at each of those.

General pattern for using a HOF on lists



Choose your abstraction from the ones in Chapter 18.

The contract for your combiner depends on which abstraction you choose.

The arguments for the different abstractions are different. If this were foldr, the base would go here.

Pattern for using an HOF: **map**

```
;; list-fn : ListOfX -> ListOfY
(define (list-fn lst)
  (local (; operator : X -> Y
          ; purpose statement for operator
          (define (operator x) ...))
   (map operator lst)))
```

Here is the pattern for a use of **map**. It is necessary to fill in the right data definitions for **X** and **Y**. It isn't necessary to make the operator a local function or lambda, but if you do, you must write a contract and purpose statement (also examples if they are needed to clarify purpose).

Or you could use lambda

```
;; list-fn : ListOfX -> ListOfY
(define (list-fn lst)
  (map
    ; X -> Y
    ; purpose statement for operator
    (lambda (x) ...))
    lst)))
```

Example

```
;; STRATEGY: Use template for ListOfNumber on lon
(define (add-x-to-each x lon)
    (cond
      [(empty? lon) empty]
      [(else (cons
               (+ x (first lon))
               (add1-to-each x (rest lon))))]))

;; strategy: Use HOF map on lon.
(define (add-x-to-each x lon)
   (map
     ;; Number -> Number
     (lambda (n) (+ x n))
     lon))
```

This one is so simple you don't need a purpose statement for the **lambda**.

Here's an original function, and what we get after we've converted it to use map. Here I've used lambda, but that isn't necessary.

Pattern for using a HOF: **filter**

```
;; list-fn : ListOfX -> ListOfX
(define (list-fn lst)
  (local (; X -> Boolean
          ; purpose statement for test
          (define (test x) ...))
   (filter test lst)))
```

Similarly, here is the pattern for a use of **filter**.

Or you could use lambda

```
;; list-fn : ListOfX -> ListOfX
(define (list-fn lst)
  (filter
    ; X -> Boolean
    ; purpose statement for the test
    (lambda (x) ...)
    lst))
```

Pattern for using an HOF: **andmap/ormap**

```
;; list-fn : ListOfX -> Boolean
(define (list-fn lst)
  (local (; X -> Boolean
          ; purpose statement for test
          (define (test x) ...))
   (andmap/ormap test lst)))
```

Either **andmap** or **ormap**.

Using an HOF with **lambda** : **andmap/ormap**

```
;; list-fn : ListOfX -> Boolean
(define (list-fn lst)
  (andmap/ormap
    ; X -> Boolean
    ; purpose statement for the test
    (lambda (x) ...)
    lst))
```

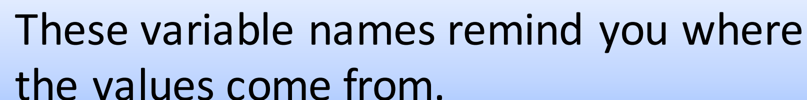
Pattern for using an HOF: **foldr**

```
;; list-fn : ListOfX -> Y
(define (list-fn lst)
  (local (; X Y -> Y
          ; purpose statement for combiner
          (define (combiner x y) ...))
   (foldr combiner base-val lst)))
```

Finally, here is the pattern for a use of **foldr**.

Pattern for using an HOF with **lambda**: **foldr**

```
;; list-fn : ListOf<X> -> Y
(define (list-fn lst)
  (foldr
    ; X Y -> Y
    ; purpose statement for combiner
    (lambda (first-elt ans-for-rest) ...)
    base-val
    lst))
```



Recognizing Opportunities for using HOF's

- Once you get the idea, you can anticipate when you can use an HOF, and apply it directly.
- If your function treats all members of the list in the same way, then your function is a candidate.
  - remove-evens --Yes, all elements are included if they are even.
  - count-trues --Yes, all elements are counted if they are **true**.
  - remove-first-even --No, elements after the first even are treated differently.

What's the strategy?

- From now on, you can use HOFs anywhere.
- So you could write

```
(filter some-fcn (robot-history r))
```

- For your strategy you could write
  - Use template for Robot on r OR
  - Use HOF filter on (robot-history r)
- Either would be OK. Use whichever one best describes how your whole function works.

Which HOF should I use?

- Key: look at the contracts
- Here's a recipe, followed by a video demonstration

Recipe for Using a Higher-Order Function to process a list

1. Write the contract, purpose statement, and examples for your function. Does your function process a list? Does it treat all members of the list in more or less the same way?
2. Choose a function from Chapter 18 whose contract matches yours. What choices for X, Y, etc. match your contract?
3. Create a copy of the pattern for the function. What is the contract for the combiner? What is its purpose?
4. Define the combiner function.
5. Test as usual.

Video Demonstration: Using an HOF (part 1)

https://www.youtube.com/watch?v=WzBt8637-uM&feature=youtu.be

Remember: We don't write **ListOf<X>** any more; we write **ListOfX** instead.

Video Demonstration: Using an HOF (part 2)

https://www.youtube.com/watch?v=g0X6OKvUB40&feature=youtu.be

Summary

- You should now be able to:
  - Use pre-built HOFs for processing lists found in ISL.
  - Recognize the pre-built HOF that's appropriate for your function.
  - Follow the recipe for converting your use of a template into a use of an HOF

Next Steps

- Study 05-4-sets.rkt in the Examples folder.
- If you have questions about this lesson, ask them on the Discussion Board.
- Do Guided Practice 5.5
- Do Problem Set 5.