
Module 8: Abstraction

CPSC 110

Peyton Seigo

2018-10-26

Module 8: Abstraction

Learning goals

- Be able to identify 2 or more functions that are candidates for abstraction.
- Be able to design an abstract function starting with 2 or more highly repetitive functions (or expressions).
- Be able to design an abstract fold function from a template.
- Be able to write signatures for abstract functions.
- Be able to write signatures that use type parameters.
- Be able to identify a function which would benefit from using a built-in abstract function.
- Be able to use built-in abstract functions.

Refactoring

- Remember to:
 - Pass new parameter in natural recursive calls
 - Replace point(s) of variance with variable(s)
 - * Point of variance could be a value or function call

Working through the recipe

With abstract functions, it gets **harder** as we go back towards the signature.

1. Write abstract function; replace bodies of original function(s)
2. Unit tests
 - Can be abstracted from `check-expects` of original function(s); copy all original tests and narrow down what you need
 - Insert appropriate additional parameter into tests
 - Revising existing examples may be easier than starting from scratch
3. Purpose
 - Can sometimes be abstracted from orig. purpose(s), but not always
 - Purpose statements can take A LOT of tries to get right. This is normal.
 - From now on, anytime you want a “ListOfX” type, use `(listof X)` instead of writing a data definition for it
4. Signature
 - Lists: use `(listof <Type>)`
 - Functions: use `(<Type1> -> <Type2>)`
 - Consuming a generic type: use *type parameters* for consumed/produced types

- Lists: (`listof T`), a list containing one or more `T` objects
- Functions: (`T -> U`) such that `T` is some type consumed and `U` is some type produced
- Example: (`@signature (T -> U) (listof T) -> (listof U)`).
 - * The template is (`@template (listof T) add-param`)
 - * The definition is (`define (map fn lon) ...`)

HtDF: Writing functions that call abstract functions

Writing a function that calls an abstract function like `filter`, `map`, or an abstract function you write yourself.

- Template tag: (`@template use-abstract-fn`)
- Base case test is NOT needed when using built-in abstract functions—their base case is already tested

Functions that are **abstract functions themselves** and **consume** a generic function do not have any special `template` tags. Note that if an abstract function consumes a list, the template will have (`listof <T>`).

Closures

A closure is a `local` function defined within a function body and **uses a parameter of its enclosing function**. In these cases, you MUST define the function using `local`.

Below, the helper `bigger?` is a closure. It “closes over” the value of `threshold` passed to `only-bigger`.

```
1 (define (only-bigger threshold lon)
2   (local [(define (bigger? n)
3             (> n threshold))]
4     (filter bigger? lon)))
```

Terminology

Abstraction

- **Abstraction:** generalizing repetitive code (through refactoring)
 - Make programs smaller + easier to read
 - Separates knowledge domains more clearly in code

- **Abstract function:** a helper shared between multiple functions
 - More general than the original code
- **Abstraction from examples:** abstracting/generalizing functions that have already been written
 - Backwards HtDF recipe: *Function definition* -> *Tests* -> *Purpose* -> *Signature*
- **Higher order function:** can a) consume one or more function, and b) produce a function
- **Type parameter:** a name for some generic type; often used in an abstract function's signature and template tags (*X*, *Y*, *Z*, *T*, *U*, etc.)

Built-in abstract functions template

Template for writing a function that calls a built-in abstract function:

```
1 (@template (listof t) use-abstract-fn)
2 (define (some-fn lot)
3   (<built-in-fn> ... lot))
```

Built-in abstract functions

Built-In Abstract Functions ISL and ASL have the following built-in abstract functions.

- **build-list:** make a list of elements based on their index (doesn't have to be a list of numbers!)
 - (**build-list** *n* **identity**) makes a list of naturals for [*0*, *n*)
- **filter:** list of elements that satisfy a given predicate
- **map:** apply a fn to each element
- **andmap:** analogous to Elixir's **all?**
- **ormap:** analogous to Elixir's **any?**
- **foldl**
- **foldr:** reduce a list of elements to a single element
 - similar to Elixir's **reduce**, but **foldr** does not use an [apparent] accumulator
 - HexDocs actually says Reduce is sometimes called **fold**!

foldr is the abstract function for the (**listof** *T*) template:

Signatures for each built-in function:

```
1 (define (fn-for-lot lot)
2   (cond [(empty? lot) (...)]
3         [else
```

```

4      (... (first lot)
5          (fn-for-lot (rest lot))))))
6
7 (foldr ... ... lot)

```

- The first ... in `foldr` is the **combination** in `fn-for-lot`
- The second ... in `foldr` is the **base case** in `fn-for-lot`

List of signatures **without their function arguments** for comparing to functions that have NOT yet implemented an abstract function:

1	CONSUMES	PRODUCES		ABSTRACT FUNCTION
2	-----+-----			
3	<code>Natural</code>	<code>-> (listof X)</code>		<code>build-list</code>
4	<code>(listof X)</code>	<code>-> (listof X)</code>		<code>filter</code>
5	<code>(listof X)</code>	<code>-> (listof Y)</code>		<code>map</code>
6	<code>(listof X)</code>	<code>-> Boolean</code>		<code>andmap</code>
7	<code>(listof X)</code>	<code>-> Boolean</code>		<code>ormap</code>
8	<code>Y (listof X)</code>	<code>-> Y</code>		<code>foldr</code>
9	<code>Y (listof x)</code>	<code>-> Y</code>		<code>foldl</code>

Signature + purpose for each built-in abstract function according to *Language* page.

```

1 (@signature Natural (Natural -> X) -> (listof X))
2 ;; produces (list (f 0) ... (f (- n 1)))
3 (define (build-list n f) ...)
4
5 (@signature (X -> Boolean) (listof X) -> (listof X))
6 ;; produce a list from all those items on lox for which p holds
7 (define (filter p lox) ...)
8
9 (@signature (X -> Y) (listof X) -> (listof Y))
10 ;; produce a list by applying f to each item on lox
11 ;; that is, (map f (list x-1 ... x-n)) = (list (f x-1) ... (f x-n))
12 (define (map f lox) ...)
13
14 (@signature (X -> Boolean) (listof X) -> Boolean)
15 ;; produce true if p produces true for every element of lox
16 (define (andmap p lox) ...)
17

```

```
18 (@signature (X -> Boolean) (listof X) -> Boolean)
19 ;; produce true if p produces true for some element of lox
20 (define (ormap p lox) ...)
21
22 (@signature (X Y -> Y) Y (listof X) -> Y)
23 ;; (foldr f base (list x-1 ... x-n)) = (f x-1 ... (f x-n base))
24 (define (foldr f base lox) ...)
25
26 (@signature (X Y -> Y) Y (listof X) -> Y)
27 ;; (foldl f base (list x-1 ... x-n)) = (f x-n ... (f x-1 base))
28 (define (foldl f base lox) ...)
```