Functional Abstraction and Polymorphism

Corky Cartwright
Department of Computer Science
Rice University
(with thanks to John Greiner)

Abstracting Designs

- "The elimination of repetitions is the most important step in the (program) editing process"   – Textbook
- The software engineering term for revising a program to make it better or accommodate an extension: *refactoring*.
- Repeated code should be avoided at almost all costs. Why? Revisions involved repeated code are almost impossible to get right.
- *Abstractions* help us avoid this problem.

The Need for Abstractions

```
;; contains-doll? : los  ->  boolean
;; (contains-doll? alos) determines whether alos
;; contains the symbol 'doll
(define (contains-doll? alos)
  (cond
    [(empty? alos) false]
    [else (or (symbol=? (first alos) 'doll)
              (contains-doll? (rest alos)))]))
```

The Need for Abstractions

```
;; contains-car? : los  ->  boolean
;; (contains-car? Alos) determines whether
;; alos contains the symbol 'car
(define (contains-car? alos)
  (cond
    [(empty? alos) false]
    [else (or (symbol=? (first alos) 'car)
              (contains-car? (rest alos)))]))
```

Creating Abstractions

How can we write one function that replaces

- **`contains-doll?`**
- **`contains-car?`**
- **`contains-pizza?`**
- **`contains-comp210?`**
-

Creating Abstractions

```
;; contains? : symbol, los  ->  boolean
;; (contains? s alos) determines whether alos
;; contains the symbol s
(define (contains? s alos)
  (cond
    [(empty? alos) false]
    [else (or (symbol=? (first alos) s)
              (contains? s (rest alos)))]))
```

Can We Do Better?

```
;; contains? : any list-of-any -> boolean
;; (contains? v aloa) determines whether
;; aloa contains the value v
(define (contains? v aloa)
  (cond
    [(empty? aloa) false]
    [else (or (equals? (first aloa) v)
              (contains? v (rest aloa)))]))
```

Using Abstractions

- How do we use **`contains?`**

```
(contains? 'doll (list ...))
(contains? 'car  (list ...))
```

- How can we better define **`contains-doll?`**, **`contains-car?`**

```
(define (contains-doll? alos) (contains? 'doll alos))
(define (contains-car? alos) (contains? 'car alos))
```

- This idea is called **reuse**. Let’s run with it!

A more complex example

```
;; below : lon number  ->  lon
;; (below alon n) returns the list containing the
;; numbers in alon that are less than or equal to n
(define (below alon t)
  (cond [(empty? alon) empty]
        [else
          (cond [(<= (first alon) t)
                 (cons (first alon)
                       (below (rest alon) t))]
                [else (below (rest alon) t)])]))
```

A more complex example

```
;; above : lon number  ->  lon
;; (above alon n) returns the list of the numbers
;; in alon that are greater than t
(define (above alon t)
  (cond [(empty? alon) empty]
        [else
          (cond [(> (first alon) t)
                 (cons (first alon)
                       (above (rest alon) t))]
                [else (above (rest alon) t)])]))
```

Creating Abstractions

How can we write one function that replaces

- **`below`**
- **`above`**
- **`equal`**
- **`same-sign-as`**
- … ?

Creating Abstractions cont.

```
;; filter1 : relOp lon number  ->  lon
;; (filter1 test alon n) returns the list of the numbers t
;; in alon such that (test t n) is true
(define (filter1 test alon t)
  (cond [(empty? alon) empty]
        [else
          (cond [(test (first alon) t)
                 (cons (first alon)
                       (filter1 test (rest alon) t))]
                [else (filter1 test (rest alon) t)])]))
```

What did we do? Use a function as an argument!

relOp abbreviates *relational operator.* Requires the Intermediate language level.

Using Abstractions

- How do we denote (express) function values? In three different ways. We will use the simpler one for now: write the name of a defined function (primitive, library, or program-defined):

```
(filter1 < (list ...) 17))
(filter1 > (list ...) 17))
```

How can we define functions **`below`** and **`above`** without code duplication?

```
(define (below alon t) (filter1 <= alon t))
(define (above alon t) (filter1 > alon t))
```

- Both functions will work just as before!

Repetition in Types

Repetition also happens in type definitions.

A **lon** is one of:

- **empty**
- **(cons n alon)**,

where **n** is a number and **alon** is a **lon**.

A **los** is one of:

- **empty**
- **(cons s alos)**,

where **s** is a symbol and **alos** is a **los**.

Abstracting Types

A **`listOf X`** is one of:

- **`empty`**
- **`(cons x alox)`**,

  where **`x`** is an **`X`** and **`alox`** is a **`listOf X`**.

A variable at the type level.

In FP, called parametric polymorphism

In OOP, called genericity (generic types)

Abstracting Types

Type	Example(s)
`listOf number`	`(list 1 2 3)`
`listOf symbol`	`(list 'a 'b 'pizza)`

`any`	`(list 1 2 3)`
	`(list 'a 'b 'pizza)`
	`empty`
	`(list 1 'a +)`

Important! **`listOf X`** is NOT **`listOf any`**

Revisiting `filter1`

What is a more precise description of `test`'s type?

```
;; filter1 : relOp (listOf number) number →
;;    (listOf number)
;; where relOp is (number number -> boolean)
;; (filter1 r alon n) returns the list of numbers
;; t from alon such that (r t n) is true
```

Revisiting `filter1`

Can we generalize the type of filter1?

```
;; filter1 : (number number -> boolean) (listOf number) number ->
;;           (listOf number)
```

What is special about **number**? Does filter1 rely on any of the properties of **number**?

No. It could be any type **X**.

```
;; filter1 : (X X -> boolean) (listOf X) X -> listOf X
```

A better form of filtering?

Claim: `filter1` is unnecessarily complex and specialized. Compare it with the following function (which is part of the Scheme library).

```
;; filter (X -> boolean) (listOf X) -> listOf X
;; (filter p alox) returns the list of elements e
;; in alox that satisfy the predicate p.
```

Note that **p** is unary, which means that we must pass matching unary functions as arguments. This convention is inconvenient unless we add a new linguistic mechanism called lambda-notation to our language. This mechanism is available in the “Intermediate student with lambda” language. Wait until next lecture.

Final thoughts

- Function abstraction adds **expressiveness** to the programming language
- Type abstraction (polymorphism) does the same for type annotations
- They work well together, *e.g.* OCAML, Haskell.
- Programming will continue to get "easier" as we add abstraction mechanisms to our languages.

For Next Class

- Get started on HW3 (which inclues a *real* challenge problem.
- Reading:
  - Chs. 19,20: Linguistic Abstraction, Functions as values