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
- 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
;; to determine 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
;; to determine 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-comp311?`
- ...

Creating Abstractions

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

Creating Abstractions, cont.

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

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
;; to construct a list of those numbers
;; in alon that are less than or equal to t
(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
;; to construct a list of those 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

```
;; filter1 : relOp lon number  ->  lon
;; to construct a list of those numbers n
;; 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*

Using Abstractions

- How do we denote (express) function values? In three different ways. We will only use the simplest 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 `above`, `below` without code duplication?

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

- Both functions will work just as before!
- Can we do better? Example is warped by assumption that abstracted function is binary. Why? No good reason. A unary filter is generally superior.

.

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

An `X-list` is one of:

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

  where `x` is an `X` and `alox` is an `X-list`.

A variable at the type level.

In FP, called parametric polymorphism

In OOP, called genericity (generic types)

Abstracting Types

Type	Example(s)
`number-list`	`(list 1 2 3)`
`Symbol-list`	`(list 'a 'b 'pizza)`

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

Important! `X-list` is NOT `any-list`

Revisiting `filter1`

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

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

where `relOp` is

```
(number number -> boolean)
```

Revisiting `filter1`

Can we generalize the type of `filter1`?

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

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) X-list X ->  X –list
```

Comment: `filter1` is still lame. It should be unary:

```
;; filter : (X -> boolean) X-list  ->  X –list
```

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.
- Function abstraction is very lightweight in Racket and other functional languages. Rather clumsy (but still important) in Java; inheritance is not a general but better (notationally simpler) in many cases.