Functional Abstraction and Polymorphism

Corky Cartwright
Stephen Wong
Department of Computer Science
Rice University

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.
- *Abstractions* affect how we think about writing software (Stephen).

The Need for Abstractions

```
;; Type contract
;;   contains-doll? : los  ->  boolean
;; Purpose: (contains-doll? alos) determines if alos
;;   contains the symbol 'doll
(define (contains-doll? alos)
  (cond
    [(empty? alos) false]
    [else (or (equal? (first alos) 'doll)
              (contains-doll? (rest alos)))]))
```

The Need for Abstractions

```
;; Type contract
;;   contains-car? : los  ->  boolean
;; Purpose: (contains-car? alos) determines if alos
;; contains the symbol 'car
(define (contains-car? alos)
  (cond
    [(empty? alos) false]
    [else (or (equal? (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?`**

How can we subsume the following functions as well

- **`contains-17: list-of-number -> boolean`**
- **`contains-true: list-of-boolean -> boolean`**

Creating Abstractions cont.

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

What do we need to change to produce a function with type

alpha list-of-alpha -> boolean

that generalizes this one? Only our documentation! (and perhaps changing the name **s** to **a** and **alos** to **aloa**).

What changes would have been necessary if we had used symbol=**?** instead of **equal?**

Abstracted Version

```
;; Type contract
;;   contains? : alpha list-of-alpha  ->  boolean
;; Purpose: (contains? a aloa) determines
;; whether aloa contains the element a of type alpha.
(define (contains? a aloa)
  (cond
    [(empty? aloa) false]
    [else (or (equal? (first aloa) a)
              (contains? a (rest aloa)))]))
```

Note: in Scheme libraries, **contains?** is called **member?**.

contains? accommodates *variant* behavior regarding which element value is searched by making that element value a parameter. Both **contains-doll?** and **contains-car?** inappropriately fix this value.

Challenge

Can we associate a more general parametric type with **`contains?`** than

`alpha list-of-alpha  ->  boolean`

Is it useful in Scheme in practice?

Using Abstractions

How do we use **contains?**

```
(contains? 'doll (list ...))
(contains? 'car  (list …))
(contains? 17 (list ...))
```

How can we better define **contains-doll?**, **contains-car?**, **contains-17?**

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

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

A more complex example

```
;; Type contract:
;;   below : lon number  ->  lon
;; Purpose: (below alon n) returns the list containing
  ;; the numbers in alon less than or equal to n
;; Code:
(define (below alon n)
  (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
;; Purpose: (above alon n) returns the list of the
;;   numbers in alon that are greater than n
(define (above alon n)
  (cond [(empty? alon) empty]
        [else
          (cond [(> (first alon) n)
                 (cons (first alon)
                       (above (rest alon) n))]
                [else (above (rest alon) n)])]))
```

Creating Abstractions II

How can we write one function that replaces

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

Creating Abstractions II cont.

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

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

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

Using Abstractions II

How do we denote (express) function values? In three different ways. We will 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 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 **`list-of-alpha`** is one of:
- **`empty`**
- **`(cons a aloa)`**,

  where **`a`** is an **`alpha`** and **`aloa`** is a **`list-of-alpha`**.

A variable at the type level.

In FP, called parametric polymorphism

In OOP, called genericity (generic types)

Abstracting Types

Type	Example(s)
`list-of-number`	**`(list 1 2 3)`**
`list-of-symbol`	**`(list 'a 'b 'pizza)`**
`any`	**`(list 1 2 3)`** **`(list 'a 'b 'pizza)`** **`empty`** **`(list 1 'a +)`**

Important! **`list-of-alpha`** is NOT **`list-of-any`**

Revisiting `filter1`

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

```
;; Type contract
;;   filter1 : relOp list-of-number number ->
;;     list-of-number
;; where relOp is (number number -> boolean)
;; Purpose: (filter1 r alon n) returns the list-of-
;;   number m from alon such that (r m n) is true
```

Revisiting `filter1`

Can we generalize the type of filter1?

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

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

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

```
;; filter1 :
;;    (alpha alpha -> boolean) list-of-alpha alpha ->
;;    list-of-alpha
```

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: (alpha -> boolean) list-of-alpha -> ]
;;   list-of-alpha
;; Purpose: (filter p aloa) returns the list of
;; elements in aloa 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 in the absence of a new linguistic mechanism called lambda-notation which is introduced in Lecture 9. This mechanism is available in the “Intermediate student with lambda” language.

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.
- In OO languages, integration is less clean in “generic” Java and C#. Opportunity for improvement in new OO languages. Scala?
- Programming will continue to get “easier” as we add abstraction mechanisms to our languages.

For Next Class

- Slides for earlier lectures have been cleaned up. Check them out.
- Review hand evaluation rule for **`local`**
- Work on HW3 (which includes a *real* challenge problem).
- Reading:

  Chs. 19-22: Linguistic Abstraction,
  Functions as values
  Chs. 21-22: Abstracting designs
  and first class functions