# 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 – HTDP
- 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 involving 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-number?`
- …

As we design code that embodies an abstraction (such as containing a particular object), performing explicit type checks ***blocks*** abstraction. Explicit type checking expressed as executable code is a terrible software engineering idea! Why? The type information to should be confirmed in a given invocation depends on the context of the invocation!

# 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 generic (parametric typing), the correct type contract is:

```
;; contains? : (list-of T)  ->  boolean
```

Note that the “scope” of the type parameter **`T`** is the entire type expression **`(list-of T)  ->  boolean`**

# 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
;; contract: 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
;; contract: 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 the functions

- **`below`**
- **`above`**
- **`equal`**
- **`same-sign-as`**
- **`...`**

# Creating Functional Abstractions

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

# Using Abstractions

- How do we denote (express) function *values*? Two ways: names and `lambda`-abstractions. We will only use the first way for now: write the *name* of a defined function (primitive, library, or program-defined) as in:

```
(filter1 < (list ...) 0))
(filter1 > (list ...) 0))
```

- How can we define the functions `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? This example is warped by assumption that the abstracted function is a relational operator The standard `filter` operation that takes a unary predicate which is simpler, but it requires `lambda`-notation to express the function arguments in our examples!

# 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 T)`** is one of:

- **`empty`**
- **`(cons t alot)`**

  where **`t`** is a **`T`** and **`alot`** is an **`(list-of T)`**.

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)` |
| `(list-of any)` | `(list 1 2 3)`<br>`(list 'a 'b 'pizza)`<br>`empty`<br>`(list 1 'a +)` |

Important! **`(list-of X)`** is NOT the same type as **`(list-of any)`**. In some contexts, it is a sub-type.

# Improving `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 X.

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

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

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

# A Better Filter function

```
;; filter: (T -> boolean) (list-of T)  ->  (list-of T)
;; contract: (filter f alot) constructs the (list-of T) consisting of all
;; elements e in alot such that (f e) is true.  The ordering of elements
;; is preserved
(check-expect (filter number? empty) empty)
(check-expect (filter number? (list 1 false 2 true 3)) (list 1 2 3))

(define (filter f alot)
  (cond [(empty? alot) empty]
        [else
          (cond [(f (first alot))
                  (cons (first alot)
                        (filter f (rest alot)))]
                 [else (filter f (rest alot))])]))
```

# Final thoughts

- Function abstraction adds *expressiveness* to a programming language
- Type abstraction (polymorphism) does the same for type annotations.
- They work well together, *e.g.* the ML family (OCAML, Haskell).
- Core Racket does not have static type system but we still use types (homogeneous subsets of the data domain) informally in stating contracts. A type declaration is a restricted form of contract!
- Function abstraction is very lightweight in Racket and other functional languages. In contrast, it is rather clumsy and heavy-weight (both in notation and implementation cost), but still important in Java; single inheritance is not a general mechanism for expressing function abstraction but it is notationally simpler in the cases where it is applicable. Since Java 8, Java supports a clean form of multiple inheritance via interfaces with concrete methods (code!). The **`default`** method nomenclature is unfortunate.