

COMPUTER SCIENCE 51

Spring 2009
cs51.seas.harvard.edu

Prof. Greg Morrisett
Prof. Ramin Zabih

Today

- How to enforce the abstraction barrier in PLT scheme.
 - modules
 - information hiding
 - contracts
- Some simple abstract data types
 - sets
 - stacks
 - queues

Recall the Setup

- We have an *abstract data type*
 - e.g., sets of integers
- We have a *concrete implementation*
 - e.g., lists of integers
 - coupled with an invariant: e.g., no duplicates
- We have a *contract* for clients:
 - operations we guarantee to provide
 - e.g., `member?`, `insert`, `union`, ...
 - promise about the semantics of operations
 - e.g., `(member? x (insert x s)) ↓ true`

The Intention

- We can change the implementation:
 - e.g., use a binary tree instead of list
 - e.g., use a different invariant (sorted, no duplicates)
 - e.g., change to a different algorithm
- As long as we respect the contract.
 - i.e., export at least the same operations
 - i.e., with the same semantics
- Client code should be unaffected.
 - except in good ways (e.g., runs faster).

What Can Go Wrong?

- Clients may violate the contract
 - may use an internal operation we didn't mean to export (hidden APIs).
 - may feed a *concrete* value to one of the operations that does not respect our invariants.
 - e.g., a list with duplicates
- We may violate the contract
 - e.g., violate the invariant
 - e.g., change the observable semantics

Example: Set Contract

Our abstraction is a *set of numbers*.

- `empty-set` represents \emptyset .
- `(insert n s)` represents the set $\{n\} \cup s$.
 - where n is a number.
- `(union s1 s2)` represents the set $s1 \cup s2$.
- `(intersect s1 s2)` represents the set $s1 \cap s2$.

Set Contract Continued

- `(member? n s) ↓ true` when $n \in s$.
- `(member? n s) ↓ false` when $n \notin s$.
- `(set? s) ↓ true` only when `s` is:
 - `empty-set`
 - `(insert n s)` for a number `n` & set `s`
 - `(union s1 s2)` or `(intersect s1 s2)` for sets `s1` and `s2`.

Implementation #1

- Represent sets as lists.
 - `(define empty-set empty)`
 - `(define insert cons)`
 - `(define union append)`
 - ...
- Problems?

Better Solution

- Always wrap implementation in a struct.
 - `(define-struct set (elements))`
 - Gives us a predicate for determining whether we have a set or a list.

Set Implementation #2

```
(define-struct set (elements))

(define empty-set (make-set empty))

(define (insert x s)
  (if (and (number? x) (set? s))
      (make-set (cons x (set-elements s)))
      (error 'insert
             "expecting a number and a set")))
```

Set Imp. #2 Continued

```
(define (union s1 s2)
  (if (and (set? s1) (set? s2))
      (make-set (append (set-elements s1)
                        (set-elements s2)))
      (error 'union "expecting two sets")))
```

Set Imp. #2 Continued

```
(define (mem? x xs)
  (cond [(empty? xs) false]
        [(= x (car xs)) true]
        [else (mem? x (cdr xs))]))

(define (member? x s)
  (if (and (number? x) (set? s))
      (mem? x (set-elements s))
      (error 'member?
             "expecting a number and set")))
```

Set Imp. #2 Continued

```
(define (inter xs ys)
  (cond [(empty? xs) empty]
        [(mem? (car xs) ys)
         (cons (car xs)
                (inter (cdr xs) ys))]
        [else (inter (cdr xs) ys)]))

(define (intersect s1 s2)
  (if (and (set? s1) (set? s2))
      (make-set (inter (set-elements s1)
                       (set-elements s2)))
      (error 'intersect "expecting two sets")))
```

We're Doing Better...

- Using a struct, we get a predicate.
 - *generativity*: a “fresh” type
- All of the constructors make sure they only build valid sets.
 - e.g., insert checks it is given a number.
- Does this mean we can assume every set object only has a list of numbers for the elements?
 - No!

Bad Clients...

A bad client could still:

- call `make-set` passing something besides a list.

In addition, a bad client can directly call internal operations:

- e.g., `mem?` and `inter` which assume they are operating over lists of numbers.

Modules in PLT

- A module is of the form:

```
(module name scheme
  (provide x1 x2 x3 ...))

(define x1 ...)
(define x2 ...)
...
)
```

Scope

- Inside a module, one has access to all of the variables defined there.
- Outside a module, a client only has access to those variables that are put in the `provide` clause.
 - Good default: clients don't have access unless you explicitly provide it.
 - Most (decent) languages have a way of limiting the scope of identifiers so that we can support information hiding.

Set Imp. #3

(see `sorted-list-set.ss`)

Discussion

- Invariants for this code:
 - sets are sorted, no duplicate, lists of numbers.
- Clients can't call make-set.
 - not provided to them.
- The only places where we call make-set, we make sure that we satisfy the invariant.
- In turn, this means that the only objects for which set? returns true are those that satisfy the invariant.

Defensive Coding

- This encapsulation saves us a *ton* of error checking in our code.
 - merge expects two sorted lists of numbers.
 - Only union calls merge.
 - (Clients cannot!)
 - But merge passes the elements of objects that pass the set? predicate.

Changing the Impl.

(see binary-tree-set.ss)

Discussion

- Here, we have a completely different representation and invariant.
 - tree built from empty and node
 - node should only contain sub-trees and numbers.
 - tree should be ordered (left-<, right->)
- And completely different internal operations.
- But clients can't tell the difference!

One more note...

- The fact that we used structs to get a predicate, and modules to hide operations we don't want to export isn't important.
 - your favorite language may not have these specific constructs.
- What is important is the ideas of *encapsulation* and *information hiding*.
 - must discover how to achieve these properties for each language you use.
 - not always possible to enforce with the language (e.g., C/C++).

More ADTs

- Stacks (see stack.ss)
- Queues (see queue.ss)

Contracts

- We still have lots of annoying checking code around the functions we do provide.
- In PLT, we can write explicit contracts:

```
(provide/contract
  [member?
   (-> number? set? boolean?)]
 [insert
  (-> number? set? set?)]
 [empty-set set?])
```
- Any predicate can be used in a contract.

Where to Find More...

See the PLT documentation:

<http://docs.plt-scheme.org/guide/index.html>

In particular, chapters 6 and 7.

Next time: higher-order functions.