CSE341: Programming Languages

Lecture 24
Racket Modules, Abstraction with Dynamic Types; Racket Contracts

Dan Grossman
Fall 2011
Another modules lecture

• Recall lecture 12: SML modules. Key points:
  – Namespace management for larger programs (structures)
  – Hiding bindings inside the module ($\text{gcd}$, $\text{reduce}$)
  – Using an abstract type to enforce invariants

```sml
signature RATIONAL = 
sig
type rational
exception BadFrac
val make_frac : int * int -> rational
val add : rational * rational -> rational
val toString : rational -> string
end

structure Rational :> RATIONAL = …
```
Racket is different

• More flexible *namespace management*
  – Convenient ways to rename during export/import
  – (In other languages, could write wrapper modules)

• Dynamic typing still has ways to create *abstract types*
  – Just need to be able to make a new type at run-time
  – This is what *struct* does; Scheme has nothing like it

• By default, each file is a module
  – Not necessary but convenient

• State-of-the-art *contract system*
  – Arbitrary dynamic checks of cross-module calls with blame assignment
But first…

Worth emphasizing that modules are not necessary for creating abstract types: local scope and closures are enough

Recall our rationals example (but note Racket has built-in rationals):

*Interface:*
  - `make-frac` rejects 0 denominator
  - `add` adds two rationals
  - `print-rat` prints a rational in reduced form

Can implement this by maintaining these *invariants*:
  - `num` and `den` fields kept in reduced form
  - `den` is always positive
Wrong approach [see lec24_non_modules.rkt]

This uses local scope to hide \texttt{gcd} and \texttt{reduce}, but it exposes the \texttt{rat} constructor, so clients can make bad rationals

- So to be "safe", \texttt{add} and \texttt{print-rat} can re-check invariants

```scheme
(struct (rat num den))
(define rat-funs
  (letrec
    ([gcd (lambda (x y) ...)]
     [reduce (lambda (x y) ...)]
     [make-frac (lambda (x y) ...)]
     [add (lambda (r1 r2) ...)]
     [print-rat (lambda (r) ...)])
    (list make-frac add print-rat)))
(define make-frac (car rat-funs))
(define add (cadr rat-funs))
(define print-rat (caddr rat-funs))
```
**Right approach** [see lec24_non_modules.rkt]

So we also need to hide the **rat** constructor!
- Also hide mutators if you create them
- Choose to hide accessors to keep representation opaque
- This code doesn't "export" **rat?**, but doing so a good idea

```
(define rat-funs
  (let ()
    (struct (rat num den)
      (define (gcd x y) …)
      (define (reduce x y) …)
      (define (make-frac x y) …)
      (define (add rl r2) …)
      (define (print-rat r) …)
      (list make-frac add print-rat)))

(define make-frac (car rat-funs))
(define add (cadr rat-funs))
(define print-rat (caddr rat-funs))
```
The key trick

- By hiding the constructor and accessors, clients cannot make rationals or access their pieces directly.

- Clients can still pass non-rationals to `add` or `print-rat`, but any rational will satisfy the invariants.

- Technique requires fundamentally on semantics of `struct`:
  - Make a new (dynamic) type of thing
  - If `struct` were sugar for cons cells, then clients could use `cons` to make bad rationals

- So… to support abstract datatypes, dynamically typed languages need ways to make "new types of things"
  - Scheme traditionally had no such support

- Again, making `rat?` public makes perfect sense
Racket modules

• The normal and convenient way puts bindings in a file and *provides* only the ones that should be public
  – Unlike SML, no separate notion of signature – module decides what to provide

• Default is private
  – (But REPL for "Run" of a file is "inside" that file's module)
  – Which is why previous lectures used
    (provide (all-provided-out))
  – Can provide some of struct's functions

• See lec24_rationals.rkt
  – (provide make-frac add print-rat rat?)
It's the same trick

- Modules take care of hiding bindings

- `struct` takes care of making a new type

- This doesn't work if rationals are implemented with an existing type like `cons`
  - Clients could use `cons?` to figure that out and then make bad rationals

- Common misconception: Dynamically typed languages can't support abstract types
  - Some may not, but they could
Using modules [see lec24_client.rkt]

• Clients get a module's bindings with the require form
  – Many variations, using a file-name string is the simplest
    
    (require "rationals.rkt")

  – Can also get only the bindings you want, either by listing them with the only-in syntax or listing what you don't want with the except-in syntax
    • Convenient for avoiding name conflicts
    • See the manual for details
  – Can also rename bindings: rename-in and prefix-in
    • The provider can also rename when exporting

• Overall: convenient namespace management is a nice thing
Contracts

• A contract is a pre- and post-condition for a function
  – Software methodology of "design-by-contract"
  – If a function fails, blame either the caller or callee

• Old idea; Racket's modules on the cutting edge

• Can provide functions with a contract
  – Any predicate (boolean-returning function) on arguments and result
  – Any cross-module call will have its arguments and result checked at run-time (could be expensive) to assign blame
    • Intra-module calls (e.g., recursion) not checked

• (You're not responsible for the details, just the high-level idea)
**Example**

lec24_rationals_contracts.rkt provides another implementation of a rationals library with contracts on each export.

It maintains *different* (weaker) invariants, putting more work on clients, with contracts checking that work:

- Exports `rat` constructor, but contract requires integer arguments and positive denominator from client
  - Maintains these invariants
- Exports `rat-num`, `rat-den`, and `rat`?
- Does *not* keep rationals in reduced form
  - `add` doesn't care and doesn't reduce
  - `print-rat` does care (contract checks it); up to client to either call `reduce-rat` or "know" the rational is reduced
Example provide (Note: needs DrRacket 5.2)

- **contract-out** exports bindings with given contracts
- \( \rightarrow \) takes predicate functions for each argument/result and checks them on inter-module calls at run-time
  - Can use library functions or our own (e.g., reduced-rat)
- Client must satisfy argument contracts and can assume result contracts

```
(provide (contract-out
    (rat \( \rightarrow \) integer?
        (lambda (y) (and(integer? y) (> y 0)))
    rat?)
    (rat-num \( \rightarrow \) rat? integer?)
    (rat-den \( \rightarrow \) rat? integer?)
    (rat? \( \rightarrow \) any/c boolean?)
    (add \( \rightarrow \) rat? rat? rat?)
    (print-rat \( \rightarrow \) reduced-rat void?)
    (reduce-rat \( \rightarrow \) rat? reduced-rat))))
```
Contracts vs. invariants

• If you set up strong abstractions and maintain invariants, then you need to do less run-time contract checking
  – Example: No need for reduced-rat to check that the rational fields are integers with positive denominator

• This is more efficient: only check dynamically what could fail if "the other party in the contract" is wrong
  – Of course, "redundant" checks are less redundant if your abstractions are leaky due to poor design / bugs

• Invariants are not an argument against contracts
  – The two are for different purposes, as in our example