CSE 341, Spring 2011, Lecture 18 Summary

Standard Disclaimer: These comments may prove useful, but certainly are not a complete summary of all the important stuff we did in class. They may make little sense if you missed class, but hopefully will help you organize and process what you have learned.

This lecture considers two topics: (1) modularity and abstraction in Scheme (particularly extensions in Racket), and (2) general notions of equivalence related to functions.

Scheme Modularity

In lecture 11, we first studied module systems. We learned how ML module systems provide namespace management, hide private bindings, and enforce invariants by using abstract types. In this lecture, we will consider how Racket's `define-struct` also lets us enforce invariants although the approach can feel a bit different. We will see how to do this "just" with local scope and then more conveniently with Racket's module system.

We will consider the same example we did in lecture 11, an interface for positive rational numbers whose ML signature is:

```
sig
    type rational
    exception BadFrac
    val make_frac : int * int -> rational
    val add : rational * rational -> rational
    val print_rat : rational -> unit
end
```

`make_frac` returns a rational only if its arguments are positive. `print_rat` always prints rationals in reduced form (i.e., 2/3 instead of 6/9). Internal implementation decisions should not be visible to clients, so we could change the implementation without clients knowing. Such decisions include (1) the representation of rationals, (2) the existence of private helper functions, and (3) whether rationals are kept in reduced form or just printed that way.

Consider first this approach, which achieves (2), but not (1) or (3):

```
(define-struct rat1 (num den))
(define pos-rat-funs1
  (letrec
      ([gcd (lambda (x y) (cond [(= x y) x]
                                [(< x y) (gcd x (- y x))]
                                [#t (gcd y x)]))]
       [reduce (lambda (r) (let ([d (gcd (rat1-num r) (rat1-den r))])
                             (make-rat1 (quotient (rat1-num r) d)
                                        (quotient (rat1-den r) d))))]
       [make-frac (lambda (x y)
                    (unless (and (integer? x) (> x 0)
                                 (integer? y) (> y 0))
                      (error "bad rat"))
                    (reduce (make-rat1 x y)))]
       [add (lambda (r1 r2)
              (check-rat r1)
              (check-rat r2)
              (let ([a (rat1-num r1)][b (rat1-den r1)]
                    [c (rat1-num r2)][d (rat1-den r2)])
                (reduce (make-rat1 (+ (* a d) (* b c))(* b d)))))]
```

```
        [print-rat (lambda (r)
                     (check-rat r)
                     (print (rat1-num r))
                     (unless (= (rat1-den r) 1)
                       (begin (print '/)
                              (print (rat1-den r)))))]
        [check-rat (lambda (r)
                     (if (not (rat1? r))
                         (error "non-rational provided"))
                     (let ([x (rat1-num r)][y (rat1-den r)])
                       (if (or (not (integer? x))
                               (not (> x 0))
                               (not (integer? y))
                               (not (> y 0))
                               (not (= 1 (gcd x y))))
                           (error "invariants violated"))))])
    (list make-frac add print-rat)))
(define make-frac (car pos-rat-funs1))
(define add (cadr pos-rat-funs1))
(define print-rat (caddr pos-rat-funs1))
```

Here we use no new features of Scheme. To encode the idea of private helper functions, we define our conceptual module's functions in a `letrec` and then have the body of the `letrec` return the conceptual public functions in a list. So `pos-rat-funs1` evaluates to a list of functions, all of which the outside world is allowed to use. The "private" functions `gcd` and `reduce` are not in the list and therefore unavailable. Since it is awkward and inconvenient to get functions out of a list before calling them, we define top-level variables `make-frac`, `add`, and `print-rat` that are bound to the appropriate functions.

The reason this does not achieve modularity goals (1) and (3) is that clients can use the functions provided by `(define-struct rat1 (num den))`. So they can simply call `make-rat1` directly to make rationals with negative numbers. Or they can use `rat1-num` and `rat1-den` to see if the result of a call to `add1` is in reduced form. Or they can use `set-rat1-num!` and `set-rat1-den!` to mutate the contents of a rational.

To try to protect our "library" from unexpected errors given clients' abilities, each "public" function needs to recheck the arguments, which we do with a call to the "private" function `check-rat`. This is annoying, potentially expensive, and in other examples it might not even be possible. If you are worried about concurrency, checking is even more difficult because a client could use `set-rat1-num!` after we check something, so it would be necessary to copy a rational before doing the check.

In pure Scheme (i.e., without Racket's additions), we can't really do better. No matter how we represent rationals, clients will know the representation, so we cannot change the representation without potentially changing client behavior.[1] If we choose to use a `cons` cell, there is nothing to prevent clients from using `cons?` to determine that and `car` and `cdr` to get the parts for example.

However, with `define-struct` we define a new kind of thing. The only way to access the parts of something made from `make-rat1` is with `rat1-num` and `rat1-den`, so if we could hide those functions, we can hide the representation of rationals. We could also hide `rat1?` so that we could even change the name of our struct later, or we could choose to expose it so that clients can tell that they have a `rat1` even though cannot access its parts. The key to all this is to put the `(define-struct rat1 (num den))` itself in some scope and only expose some of the functions it defines to clients. In our example, this is a simple change that then lets us significantly simplify the `check-rat` function and enforce all the invariants we want:

```
(define pos-rat-funs2
  (let ()
```

[1] We probably can avoid mutation problems with strange approaches using first-class functions, but this is hardly the natural way to do a simple thing like define a strong interface.

```
    (define-struct rat2 (num den)) ; not globally visible now!
    (letrec
        ([gcd ... unchanged ... ]
         [reduce ... unchanged ... ]
         [make-frac ... unchanged ... ]
         [add ... unchanged ... ]
         [print-rat ... unchanged ... ]
         [check-rat (lambda (r) (if (not (rat2? r))
                                    (error "non-rational provided")))])
      (list make-frac add print-rat rat2?))))
(define make-frac (car pos-rat-funs2))
(define add (cadr pos-rat-funs2))
(define print-rat (caddr pos-rat-funs2))
(define rat? (cadddr pos-rat-funs2))
```

As discussed above, this version chooses to expose the predicate for rationals, but not the constructor, the accessors, or the mutators.

Again, the essence of getting abstraction in a dynamically typed language is to make a new type of thing like `rat2` that clients have no way to "get at." This seems quite different than our ML approach where we could use an abstract type and rely on static type-checking to hide the representation of something that at run-time really was, for example, just `int*int`. The end result — preserving our positive-rational invariants — is essentially the same though.

While the above discussion demonstrated that the essence of abstract types in a dynamically typed language is defining a new type of thing in a local scope, this whole technique of returning a list of functions is *not* what programmers in Racket do. Instead Racket's *module system* is much more convenient. Here is our example using the module system:

```
(module pos-rat racket
  (provide make-frac add print-rat rat?)
  (define-struct rat (num den))
  (define (my-gcd x y)
    (cond [(= x y) x]
          [(< x y) (my-gcd x (- y x))]
          [#t (gcd y x)]))
  (define (reduce r)
    (let ([d (my-gcd (rat-num r) (rat-den r))])
      (make-rat (quotient (rat-num r) d)
                (quotient (rat-den r) d))))
  (define (make-frac x y)
    (unless (and (integer? x) (> x 0)
                 (integer? y) (> y 0))
      (error "bad rat"))
    (reduce (make-rat x y)))
  (define (add r1 r2)
    (check-rat r1)
    (check-rat r2)
    (let ([a (rat-num r1)][b (rat-den r1)]
          [c (rat-num r2)][d (rat-den r2)])
      (reduce (make-rat (+ (* a d) (* b c)) (* b d)))))
  (define (print-rat r)
    (check-rat r)
    (print (rat-num r))
    (unless (= (rat-den r) 1)
      (begin (print '/) (print (rat-den r)))))
  (define (check-rat r)
    (if (not (rat? r))
        (error "non-rational provided"))))
```

While the actual module system has more features, we will consider just some basic features:

- The `module` special form takes a name for the module, a language in which the module is implemented (we will ignore this, `racket` is the language variant we have been using for all our code), a `provide` list that enumerates what bindings need to be defined in the module, and a list of bindings, expressions, etc. like you would have at top-level.
- Only bindings listed in the `provide` list are available outside the module. So `my-gcd`, `reduce`, and `check-rat` are private. Notice we can expose "part" of a `define-struct` as in our example where we provide the predicate `rat?` but not the other functions.

Namespace management works a bit different than in ML. Instead of saying something like `pos-rat.make-frac`, we have to use `require` to make a module's provided bindings available. If we write,

```
(require 'pos-rat)
```

then all the provided functions in the `pos-rat` module are available until the end of the scope in which the require appears. If it appears at top-level, that means until the end of the file. Then we just use the provided functions without any extra letters or symbols, i.e., we just write things like `(make-frac 2 4)`.

So far this is inconvenient if some of the provided functions in one required module have the same name as those in another required module (or with some local bindings we have). So a variant of requiring a module lets you write something like,

```
(require (prefix pr- 'pos-rat))
```

Now instead of this adding `make-frac`, `add`, `print-rat`, and `rat?` to the environment, it adds `pr-make-frac`, `pr-add`, `pr-print-rat`, and `pr-rat?`. Notice the prefix we choose does not have to be the name of the module; it can be any sequence of characters that can be part of a Scheme identifier.

Function Equivalences

When we studied equivalence in lecture 12, we learned that in programming languages we generally consider two pieces of code equivalent if at any place in any program the produce the same results and have the same side-effects (exceptions, printing, infinite loops, etc.). What we did not have time for was investigating three ways that we can take certain functions or function calls in a program and produce an equivalent program. Recognizing these situations can make you a better programmer and appreciate some design goals and subtleties for programming languages.

Before considering the three situations and the important caveats about them, we need to know the definition of *free variables*. For every expression (in Scheme or ML or really any language), we can compute its free variables, which is just a set of variables. They are the variables that are *used* in the expression somewhere where the binding for that use is *not* in the expressions. (So "free" here is in the sense of "not bound".) Consider this example:

```
(let ([x 2]
      [y x])
   (+ y z))
```

For the addition expression on the last line, the free variables are `y` and `z` because they are used but not bound *in the addition expression*. However, for the whole let-expression, the free variables are `x` and `z` because the use of `x` on the second line is not bound in the let-expression nor is the `z` on the last line, but the `y` on the last line is bound on the second line. If we had `let*` instead of `let`, then the free variables of the whole expression would be only `z`.

Notice that with functions, the free variables of the body is different than the free variables for the function. In `(lambda (x) (+ x y))`, the function body `(+ x y)` has two free variables, but the function itself has only one.

We now consider the three kinds of equivalence:

Systematic argument renaming: Typically callers are not affected by the names of function arguments. So given `(lambda (x) e)`, intuitively I should be able to change it to `(lambda (y) e2)` provided `e2` is, "like `e` except every `x` is changed to `y`." We often make this sort of change to improve code readability, and it's important to know we are not affecting callers.

However, we need to be careful. Suppose we start with `(lambda (x) (+ x y)`, which makes perfect sense — we add to the argment whatever `y` is bound to in the environment. We cannot replace this function with `(lambda (y) (+ y y))`, which has a different meaning — it doubles its argument. In general, the rule is the new function-argument name must not be a free variable in the function body. Otherwise, our new choice "captures" (this is actually the technical term for it) some uses of a "different" `y` in the function body and that can change the function's meaning.

Inlining: If we have a function call where we know what function we are calling, sometimes we can just avoid the call altogether by replacing the call with just the function body where we replace all the uses of the argument name with the actual argument. For example, given `((lambda (x) (+ x x)) (+ y 2))`, we could use this idea to simplify the expression to `(+ (+ y 2) (+ y 2))`. Compilers often do this sort of thing to improve the efficiency of programs that have lots of calls to small functions.

However, there are several caveats. First, this can be wrong if the expression we are substituting in has side-effects. For example, if we consider the example above with `(begin (print "hi") (+ y 2))` instead of `(+ y 2)`, the original program would print once (since we evaluate arguments before doing a call), but the result of inlining would print twice. We can't even do the inlining if the argument might not terminate, since maybe after the inlining the argument won't be executed at all. For example, if `e` never terminates, then `((lambda(x) (lambda (y) x)) e)` does not terminate, but `(lambda(y) e)` does. (Calling it doesn't terminate, but that's a different issue.)

Finally, consider this example: `((lambda(x) (lambda (y) x)) y)`. In an environment where `y` maps to 17, this produces a closure that ignores its argument and returns 17. But after incorrect inlining, we get `(lambda (y) y)`, which is a closure that returns its argument. This is another capture problem. In general, the expression we substitute in cannot have a free variable that ends up "underneath" a different binding of the same variable after the inlining.

Unnecessary Function Wrapping: The same way that `(if e #t #f)` is a "beginner's style mistake" since it is equivalent to `e`, which is shorter and more clearly expresses the same idea (assuming that `e` evaluates to a boolean, so perhaps the ML equivalent `if e then true else false` is a better example), it is often unnecessary to wrap a function in another function. For example, consider `(lambda () (f))`. This is a thunk that when called calls `f` with no arguments and returns the answer. Now consider `f` instead — we still have an expression that evaluates to a thunk that when called returns the result of calling `f`. As another example, the ML code `List.map (fn x => SOME x) lst` can be simplified to `List.map SOME x`, since `SOME` and `(fn x => SOME x)` are equivalent functions.

In general, what we can often do is take a function of the form `(lambda () (e))`or `(lambda (x) (e x))` or `(lambda (x y) (e x y)` etc. and replace it with just `e`. However, there are caveats. First, we need that none of the arguments of the function we are removing are free variables in the expression `e`. This was trivial in our examples above because this is never a problem with zero-argument functions and `SOME` has no free variables. Second, we need that evaluating `e` always terminates and has no side-effects. (We mean here evaluating `e`, not evaluating a *call* to the function produced by evaluating `e`.) For example, `(lambda () ((begin (print "hi") f)))` and `(begin (print "hi") f)` are not equivalent — the first one prints every time it is called and the second one prints only once when it is evaluated. These caveats usually do not arise.