CSE341, Fall 2011, Lecture 24 Summary

Standard Disclaimer: This lecture summary is not necessarily a complete substitute for attending class, reading the associated code, etc. It is designed to be a useful resource for students who attended class and are later reviewing the material.

This lecture discusses some basic features of Racket's module system. It is a good complement to our earlier lecture on SML's module system and, for this purpose, we use the same example as in that lecture: a small library for rational numbers. This is a less inspired choice in Racket because the Racket language already has rationals as a built-in primitive type, but the example can still serve our purposes.

A common misconception is that one needs static typing to support abstract types in a language. We show that Racket's structs are sufficient because they create a new dynamic type, distinct from all existing types, and we can hide the constructor for this type from client code. In fact, we show that we can do this even without modules (just using local scope and closures) but that modules are better style, easier to use, and support many useful features like renaming bindings that are exported or imported. Finally we give a brief demonstration of Racket's contract system, which can check arbitrary properties of arguments to and results from cross-module function calls at run-time.

Recall ML's Modules and Our Rational-Number Example

We previously studied in ML that a module system can provide *namespace management* (avoiding name conflicts for bindings in large applications), hide *private bindings* (allowing the module implementer to change/remove implementation details later), and enforce invariants by using *abstract types* (making data representation inaccessible to clients). In ML, we used a signature like this one to describe the interface to a rational-number module:

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

The implementation of our module ensured that rationals were always printed in reduced form (e.g., 2/3 instead of 6/9). Different implementations can do so in different ways by maintaining different *internal invariants*. For example, we can keep all values of type rational in reduced form and keep all denominators positive.

This approach used two techniques. First, by not including bindings for private functions like `gcd` (for computing the greatest common divisor) and `reduce` (for reducing a rational), the module does not have to specify their behavior. Second, by making `rational` an abstract type, we prevent clients from creating values of type `rational` on their own or inspecting their representation. This allows us to change the representation or our internal invariants without changing the behavior of clients.

Using Racket's `struct` Definitions for Abstract Types Without Modules

Before showing Racket's module system, we show that one does not *need* modules to hide private bindings — local scope and first-class functions are enough. This is not specific to Racket; we could have done the same thing in ML. It is not great style — this is what modules are for — but it helps teach the concepts of modularity and information hiding.

Consider this close-but-not-quite-right approach, where we omit here the full definition of the functions defined in the `letrec` binding (see the code accompanying this summary for the full definitions):

```
(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))
```

The key binding is `rat-funs`, which evaluates a `letrec` to produce a list of functions. The five functions bound to *local variables* `gcd`, `reduce`, `make-frac`, `add`, and `print-rat` can all call each other (the purpose of `letrec`) and the variables are in scope in the body of the `letrec`. The body just creates a list of three closures. So after the `letrec` evaluates, only these three closures are *directly* accessible via `rat-funs`; the other functions are "private" in the sense that they can only be called indirectly via the three functions in the list, which is what we want for modularity. Since accessing these functions via list-processing is not something clients would want to do, we bind them to top-level variables `make-frac`, `add`, and `print-rat` in the last three lines. The choice of variables names in the `letrec` and in these lines need not have anything to do with each other.

Unfortunately, the code above does *not* achieve the modularity we want. The reason is that the definition `(struct (rat num den))` is at top-level, so it is available to clients. So clients can make rationals violating our invariants directly, such as `(rat 9 6)`, `(rat 2 -3)`, or `(rat "hi" 0)`. The solution is to put the struct definition itself in a local scope so that its functions (or at least the ones we don't want to make available) are not in scope for the clients. We can this as follows, by using Racket's internal defines instead of `letrec` (the meaning is the same, but `letrec` doesn't allow `struct` definitions):

```
(define rat-funs
   (let ()
     (struct (rat num den)
     (define (gcd x y) ...)
     (define (reduce x y) ...)
     (define (make-frac x y) ...)
     (define (add r1 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))
```

Actually, it would be nice to let clients use the `rat?` predicate in their own code since in a dynamically typed language it can be useful to test at run-time, "is this a rational?" We can easily do so by adding `rat?` to the list of functions returned since it is in scope inside the definition of `rat-funs` just like all the other functions that the `struct` definition introduces.

The fundamental reason this technique works is that a `struct` definition creates a *new* dynamic type. The

only way to make a rational is to use the `rat` constructor. The `rat?` predicate answers `#t` for values created by this constructor and every other predicate, such as `cons?`, answers `#f` for values created by this constructor.

If Racket did not have a way to make new dynamic types, then we could not enforce invariants and modularity. For example, if `struct` was some sort of syntactic sugar for building data structures with cons cells, then clients could use `cons?` to learn this and use the `cons` constructor to make invariant-violating rationals.

Some languages, such as Scheme, do not have a feature like `struct` for making new types. This non-feature has advantages (it is possible to write code that works over all possible forms of data), but modularity is not one of them. It also leads to the misconception that static typing is necessary for abstract types when it is not as our example demonstrates. In a dynamically typed language, we cannot prevent clients from calling `add` or `print-rat` with non-rationals, but any rational (as defined by `rat?`) will obey our invariants.

While our example made public none of the functions that the `rat` struct introduces (or perhaps made public just `rat?`), in general we can choose to export any subset we want. For example, for a mutable struct (Racket structs are immutable by default, but you can use the `#mutable` attribute to create field-mutator functions), we might allow mutation only within a module but still make the field-accessor functions public.

Racket Modules

Basic uses of the Racket module system are straightforward. By default, each file is its own module (what SML would call a structure) and we can use the file's name (as a string) for the name of the module. Unlike SML, there is no separate notion of a signature; the module itself decides what to make public. By default, all bindings are private. A module uses the `provide` special form to make bindings public. For example, a module containing an implementation of our rational numbers could include:

```
(provide make-frac add print-rat rat?)
```

For convenience, a module can have more than one use of `provide`. There are also several variations on how to use `provide`, well explained in The Racket Guide. One form, `(provide (all-provided-out))`, makes all top-level bindings in the module public. We used this feature in some earlier lectures and homeworks to avoid interacting with the module system, but clearly from a modularity perspective it is of questionable style if the module contains internal helper functions.

To use a module, another module uses the `require` form, for example:

```
(require "rationals.rkt")
```

As expected, this makes all the bindings *provided* by the module being required available in the module that does the require.

Racket also provides good support for fine-grained control of namespace management. For example, the `require` form has variations that import only explicitly listed bindings have other ones shadow things) or import all bindings except some that are listed. Both `provide` and `require` also allow *renaming* bindings so that different modules can refer to them with different names. For example, perhaps `add` inside the rationals module would be better called `rat-add` externally. Either the `provide` or the `require` could make this change. In other languages, one might have to create a "wrapper module" with bindings like `(define rat-add add)` to perform this helpful and common namespace control. All these features can help avoid name conflicts among bindings in different modules by renaming some bindings in modules that use them.

Contracts in Racket

Software contracts are an old idea in software development that emphasizes that what a callee expects about its arguments and what a caller expects about a function's result are essential for using code correctly. A key

part of specifying a module should be specifying these *preconditions* and *postconditions* for each function. The methodology of "design-by-contract" encourages making such "contracts" a central approach to building software.

Racket has cutting-edge support for *run-time contract checking* built into the language, and this support interacts well with the module system. While you are not responsible for any details of Racket's contract system, it is good to see the high-level idea of what it means to do run-time contract checking.

Here are the basic concepts:

- A Racket module can provide a function binding with a contract.
- This contract uses arbitrary Racket functions to describe what "must hold" of each argument to the function and what is "promised to hold" of any result from the function.
- When a client module calls a provided function with a contract, the contract is checked at run-time. (Notice how this is different from static type-checking.) If an argument contract fails (return `#f`), then the caller is *blamed*. If the argument contracts succeed (do not return `#f`), then the function is called. If the result contract (run on the result of the function after it completes) fails, then the callee is *blamed*, else the result is returned. As a result, both parties to the contract can assume the other party "does what they promised to do," knowing that a contract-violation error will occur otherwise.
- Intra-module calls do not check contracts, even for functions that are provided to other modules with contracts. This is important for performance and is consistent with the idea of contracts: A module internally should not need to enforce how "other code in the same file" behaves.

As an example, a file associated with this summary re-implements our rationals library with different, actually weaker, invariants that push more work onto clients of the module:

- Clients making rational numbers must provide a positive denominator (zero was always an error, but now so are negative denominators).
- Rationals are not kept in reduced form.
- In fact, `print-rat` does not even perform the convenience of reducing its argument. Instead its contract requires that the argument is *already* reduced, which clients can achieve eitehr by calling `reduce-rat` (now provided to clients) or by just "knowing" that the rational is in reduced form.

With this approach, there is no reason to have a separate `make-frac` function since we can provide the `rat` constructor directly with an appropriate contract. Without further ado, here is the `provide` form for this module:[1]

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

[1]This syntax is new to DrRacket version 5.2, which was released in late Fall 2011. If it is 2011, you likely still have version 5.1 and this code won't work. It is easy to upgrade.

This form provides 7 bindings as you expect and the rest of the file has definitions for these functions (either via a `struct` definition or via function definitions). What is new is a contract for each provided binding. The syntax `(-> e1 ... en)` evaluates each `ei` to a function and that is the contract-checking function for the corresponding argument (where the result of `en` is for the function result). Often, library functions like `integer?` are useful; for example, we use it to reguire that all numerators are integers. Notice how the contracts for results promise things to client modules. For example, we promise that `rat-num` returns an integer. More interestingly, we promise that `reduce-rat` returns a value that `reduced-rat` "allows" (does not return `#f` for).

While the syntax of `->` is reminiscent of ML's syntax for function types, the important difference is that contracts are really "wrapper functions" that perform arbitrary checks (we can put whatever we want in a contract) at run-time (when the function is called).

Contracts vs. Invariants

Contracts and invariants serve different purposes. Contracts check properties that other parties are supposed to get right but might not. These are the obligations that a specification makes on a module and on its clients. The purpose is to detect when a contract is violated and to blame the correct party. Invariants are properties of an implementaiton that the implementation relies on for correct behavior. It is up to a module both to maintain its own invariants and to use modularity to ensure that clients cannot cause invariants to be violated.

In designing a specification for a module and in implementing the module, part of the design is determining what are the external properties (that contracts can enforce) and what are the internal invariants (that modularity should hide from clients). Our example rationals module with contracts has a different specification than our earlier rationals module, requiring more of clients (no negative denominator and calling `print-rat` with a reduced argument). While this makes a nice example for defining some contracts, in this case the earlier rationals module is probably better because it requires less of clients without making the code more complicated or slowing down any operations.

Contracts and invariants complement each other nicely: A contract has no need to check a properly maintained invariant. In our example, our contracts need not check that the `num` and `den` fields of a rational value contain integers because this follows from the contract on `rat` and the invariants of the module. It would not necessarily hold if the module provided functions to mutate the fields to hold arbitrary values.