Modules and Components

sk and dbtucker

December 4 & 6, 2000

This semester we have studied three aspects of programming languages: control, data, and runtime systems. Today we'll discuss a different aspect—organization. The goal is to let software do things outside the scope it was originally intended to serve.

In order to organize large programs, we need two features. First, we need some high-level scoping construct to break the program into several smaller pieces. Second, we need a way to compile these pieces separately.

Separate compilation is the essence of program organization. It allows us to compile one fragment of the program without having the source code of another fragment, even though it uses the other fragment. C has separate compilation—it requires that you give a partial description (called a prototype) of any external functions you use. The prototype doesn't tell the implementation of the function, just its type.

Let's design a module system; we'll focus on scoping first. The keyword **module** will introduce a new module, in this instance named *M*:

(**module** *M*
  (**define** *d1* ... *d2* ...)
  (**define** *d2* ...)
  (**define** *d3* ...))

The module *M* contains definitions of *d1*, *d2*, and *d3*. The definition of *d1* refers to *d2*.

So far this module is completely useless. What features do we need to make our module system useful?

- A way to get stuff out of the module.
- A way to enforce privacy (which parts of the module should be visible to other parts of the program).
- A way to import other modules.

We'll add an **import** statement that loads the publicly visible definitions of another module *Mi* into the module *M* we're defining. We'll also add an **export** statement that declares which definitions in *M* should be publicly visible:

(**module** *M*

```
(import M1 M2 ... Mk)
(define d1 ... d2 ...)
(define d2 ...)
(define ...)
(export d1 d2 ... dm))
```

We still need some way to get stuff out of the module so we can use it in our main program. We'll introduce a **with** expression, which binds the public definitions of a module M in its body:

```
(with M
  ...)
```

Alternatively, we can access definitions of a module via dot notation; for example, $M.d$ refers to definition d in module M.

Where do we get the names of modules? We'll assume there's a global namespace of modules that is distinct from any other namespace. So, if we have a module RSA, we could write:

```
(let (RSA ...)
  (with RSA
    ...))
```

In this expression, the first RSA refers to some local variable we are binding, whereas the second RSA refers to a module bound in the global module namespace.

We would like to implement modules without adding any new rules to the interpreter. Can we transform modules into Scheme code? Yes, but we must first compromise and say we won't have separate namespaces. We can represent a module as a function which takes an argument d and returns the definition labelled by d:

```
(define M
  (λ (d)
    (if (member? d exports)
        (case d
          [(d1) ... d2 ...] ;; return the body of definition d1
          [(d2) ...])
        (error ...))))
```

There's a problem—the definition of *d1* refers to another definition in the module, *d2*. When we evaluate (*M d1*) in our program, the *d2* in the definition of *d1* won't refer to the definition of *d2* in the module. We can use **letrec** to achieve the desired scoping:

```
(define M
  (λ (d)
    (if (member? d exports)
        (letrec ([d1 ... d2 ...]
                 [d2 ...])
```

```
        (case d
          [(d1) d1]
          [(d2) d2]))
    (error ...))))
```

Now let's see how to implement the **with** construct. Our example module is:

```
(module RSA
  ...
  (export pk encrypt))
```

We want to use the module in our program:

```
(with RSA
  (encrypt pk message))
```

One option is to transform every use of *RSA*'s definitions to function application:

```
((RSA 'encrypt) (RSA 'pk) message)
```

With this approach, we need to parse the program and determine which names are bound in the module (*encrypt*, *pk*), and which ones are bound locally (*message*). We'll use a different approach—we transform **with** expressions into expressions that use **let** to bind the module definitions locally:

```
(let ([pk (RSA 'pk)]
      [encrypt (RSA 'encrypt)])
  (encrypt pk message))
```

Now back to our module definitions—we can turn all of the **import**s into **with**s. The module:

```
(module M
  (import M1 M2 ... Mk)
  ...)
```

is rewritten as:

```
(define M
  (λ (d)
    (with M1
      ...
        (with Mk
          ...))))
```

This tranlsation is inefficient: the **with**s become **let**s, which bind variables to all the public definitions of *M1* through *Mk* *every time* you access a definition in *M*. If we move the **with** expressions outside of the λ, then Scheme binds these names exactly once:

```
(define M
  (with M1
    ...
```

```
(with Mk
  (λ (d)
    ...))))
```

Good news: now we have a basic module system! Bad news: it's fairly useless. There are several desirable properties this module system does not have:

1. multiple instantiation
2. mutually recursive modules
3. separate implementation and interface
4. separate compilability

People increasingly define the collection of criteria by the term *component*. Have we seen any components so far in this course? Does a function meet these criteria?

1. We can instantiate a function multiple times:

```
(define (make-function x)
  (λ (...) ...))
```

2. We can define mutually recursive functions using **letrec**.
3. A function's type is its interface.
4. Separate compilability is not clear—for λ without free variables, yes; for λ with free variables, no.

How about an object in Java? It also has the four properties—for separate interface and implementation, consider Java's interfaces.

These four criteria (plus one more) define components. The additional property is external linking. In our module system, linking is done via the **import** statement, which is internal to the module. C relies on an external program, ld, to link modules. This model is much more powerful because it allows you to link to different libraries at different times. (We saw an example of this utility in the conservative GC lecture.) External linking is useful because it allows you to replace implementations; there may be some tradeoff in different implementations that you wish to exploit. Note that the external linking criterion subsumes the third criterion above, separation of implementation from interface. With external linking, you need to have the interface at compile time, or else you can't do separate compilation.

Classes don't meet this external linking requirement, since you have to specify the superclass when you define the class. A superclass can have many subclasses, but asymmetrically, a potential subclass have only one superclass. So, you could make the superclass a parameter:

```
(define C
  (λ (super)
```

(**class extends** *super*
  ...)))

Classes of this form all called *mixins*. Mixins can be externally linked to their superclasses, so they are classes that meet component properties.

Other systems that meet the component requirements are ML's functorial module system and the unit system in PLT Scheme.