Specification, Implementation, and Use

Michael P. Fourman

February 2, 2010

There are many models for the software life-cycle; this course is not the place to discuss them. However, they all include phases corresponding to the specification, implementation, and use, of a software component. In this note we introduce the SML module system, which supports these activities.

The module system has three parts: *signatures, structures,* and *functors.* In Practical 1, we have already seen examples of *signatures,* which specify an interface, and *structures,* which implement an interface. You were given a signature, `A1`, and asked to provide a structure, `Answers1`, implementing that signature, you used the structure `PolyML`, using a *qualified name,* `PolyML.use`, for one of its component functions. Functors provide a tool for separating implementation and use of a software module; they will be introduced in this note.

We use the example of implementing and using a new type, of *complex numbers,* to look at these constructs in more detail[1]. In Practical 2 you will perform a couple of similar exercises, implementing two new types: rational numbers, and approximate real numbers.

Background

A complex number may be written in the form $a + ib$, as a sum of a real part, a, and an imaginary part ib. Here, a and b are real numbers. You don't need to understand what a complex number *is* to understand this note (or to use complex numbers in many applications). We will just apply a few, straightforward rules for manipulating complex numbers. In particular, we can just treat i (the, supposedly mysterious, square root of -1) as a symbol to be manipulated according to the rules below.

Addition $(a + ib)+(c + id) = ((a + c) + i(b + d))$

[1]Don't worry if you don't know anything about complex numbers; the operations we'll need are simple, and are described in detail in this note.

Subtraction $(a+ib)-(c+id) = ((a-c)+i(b-d))$

Multiplication $(a+ib)*(c+id) = ((ac-bd)+i(bc+ad))$

Division $(a+ib)/(c+id) = ((a+ib)*(c-id))*1/(c^2+d^2)$

Complex numbers may be pictured as points on the plane; $a+ib$ corresponds to the point with *cartesian* coordinates (a, b). This representation is named after Descartes. We can also use polar coordinates for the same points; this gives a different representation of the complex numbers that we will name after another French mathematician, *Argand.* For this example, we will implement the arithmetic operations on complex numbers. We do this twice, using the two different representations.

Specification

The interface we should provide is given by the following ML signature `ComplexSig`.

```
signature ComplexSig =
sig
   type complex;

   val descartes : {real    : real, imag     : real} -> complex
   and argand    : {modulus : real, argument : real} -> complex

   and ++ : complex * complex -> complex
   and -- : complex * complex -> complex
   and ** : complex * complex -> complex
   and // : complex * complex -> complex
   and X  : real    * complex -> complex
   and == : complex * complex -> bool

   and ~~      : complex -> complex
   and modulus : complex -> real
   and argument: complex -> real
   and realpart: complex -> real
   and imagpart: complex -> real
end;
```

The functions `descartes` and `argand` will construct complex numbers from explicit representations. The arithmetic operations, other than scalar

multiplication, `X`, and unary minus, `~~`, have been described earlier. The remaining functions, `modulus`, `argument`, `realpart`, and `imagpart`, allow us to construct an explicit representation of a complex number.

Implementation

The structures `Descartes` provides an implementation of this signature, based on cartesian co-ordinates. An alternative representation, using polar coordinates to implement the same signature, is given by the structure `Argand`. A structure packages together a collection of types and values. We can access these either by opening the structure, which provides access to all its components, or by using long names, such as `Argand.++`. (Long names are never infix, so we would write `Argand.++(x, y)` where we might otherwise write `x ++ y`.)

Use

When we use complex numbers, we should not be concerned to know which representation has been used to implement them. We now introduce *functors* the ML construct used to separate code that uses a given interface from the code that provides an implementation of that interface. As an example, we consider the implementation of 2-dimensional, complex vectors and matrices.

Our code for vectors and matrices will use the interface to complex numbers provided by `ComplexSig`. The specification of our vector package is provided by the signature `VectorSig`.

```
structure Descartes (*: ComplexSig*) =
struct
    type complex = real * real;

    fun r X (a,b) : complex = (r*a, r*b)

    fun (a,b) ++ (c,d): complex = (a+c, b+d)
    and (a,b) -- (c,d): complex = (a-c, b-d)
    and (a,b) ** (c,d): complex = (a*c - b*d, b*c + a*d)
    and (a,b) // (c,d): complex =
        let val factor = 1.0/(c*c + d*d)
         in
            factor X (a,b) ** (c,~d)
        end

    and ~~(a,b) : complex = (~a,~b)

    and (a,b) == (c,d) = (a=c) andalso (b=d)

    val pi = 4.0 * arctan 1.0;

    fun descartes{real, imag} = (real,imag)
    and argand{modulus, argument} = modulus X (cos argument, sin argumen

    and realpart (real, _) = real
    and imagpart (_, imag) = imag

    and modulus (r, i) = sqrt(r*r + i*i)
    and argument(r, i) =
        if r = 0.0
             then if i < 0.0 then ~pi/2.0
                             else  pi/2.0
             else arctan(i/r)
end;
```

Figure 1: Cartesian representation of Complex numbers

```
structure Argand : ComplexSig =
struct
    type complex = real * real;

    fun r X (a, m) : complex = (a, r*m)

    fun (a, m) ** (a', m'): complex = (a + a', m * m')
    and (a, m) // (a', m'): complex = (a - a', m / m')
    and ~~(a, m) : complex = (a, ~m)

    and (a,b) == (c,d) = (a=c) andalso (b=d)

    val pi = 4.0 * arctan 1.0;

    fun descartes {real=r, imag=i} =
        let val argument =
                if r = 0.0
                     then if i < 0.0 then ~pi/2.0
                                     else  pi/2.0
                     else arctan(i/r)
            val modulus = sqrt(r*r + i*i)
         in
            (argument, modulus)
        end
    and argand {modulus, argument} = (argument, modulus)

    and realpart (a, m) = m * cos a
    and imagpart (a, m) = m * sin a

    and modulus  (a, m) = m
    and argument (a, m) = a

    fun x ++ y = let val r = realpart x + realpart y
                     and i = imagpart x + imagpart y
                 in
                     descartes{real = r, imag = i}
                 end
    fun x -- y = x ++ (~~ y)

end;
```

Figure 2: Polar representation of Complex numbers

```
infix dot;

signature VectorSig =
sig
    type scalar
    type vector
    type matrix

    val vector: scalar * scalar -> vector
    val matrix: vector * vector -> matrix

    val scale : scalar * vector -> vector
    val dot   : vector * vector -> scalar
    val apply : matrix * vector -> vector
end;
```

The implementation will use the operations specified in ComplexSig. It uses the ML `functor` declaration. This allows us to write the code that we would write to implement `VectorSig` *if* we had already implemented a structure `Complex: ComplexSig`; it allows us to write this code *before* we implement `ComplexSig` — even before we choose which representation we will use to implement `ComplexSig`.

```
functor VECTOR( structure Complex : ComplexSig) : VectorSig  =
struct
open Complex;

type scalar = complex
type vector = complex * complex
type matrix = vector * vector

fun  vector (x,y) = (x,y)
fun  matrix (a,b) = (a,b)

fun scale (s,(a,b))   = vector (s ** a, s ** b)
fun (a,b) dot (a',b') = a ** a' ++ b ** b'
fun apply ((a,b),v)   = vector (a dot v, b dot v)
end;
```

The functor `VECTOR` can be compiled before we implement complex numbers. Then, when we have an implementation, say `Argand: ComplexSig`, we

can produce a structure, implementing `VectorSig`, by applying the functor, `VECTOR`, to the structure, `Argand`.

```
structure ArgandVec = VECTOR (structure Complex = Argand);
```

We can also apply the functor to `Descartes` to produce a vector package based on our other implementation of complex numbers.

This module system will allow us to build complex software from interchangeable components. It provides the tools we will need to experiment with different implementations of various datatypes.

Pragmatics

Even for a relatively small example, such as the one presented in this note, managing the various pieces of code, compiling them in the right order, and making sure we recompile the appropriate parts when we make changes, quickly becomes tedious. PolyML provides a `make` utility to support this process. By placing the various code components in separate files, in a single directory, each with a name corresponding to the name of the signature, structure, or functor it defines, we can recompile a structure, `S`, by typing `PolyML.make "S";`. If the sources of any of the components on which `S` depends have been changed, these components will be recompiled first.

The examples in this note will be found in the directory
`/public/homepages/mfourman/web/teaching/mlCourse/doc/ml/Complex`.