Information Hiding and Modules

In this section we will discuss what modules are, what problem they solve and various ways to use them.

Information Hiding and Encapsulation

A fundamental principle in software development is that of **information hiding**:

> Every part of your code should know as little as possible about other parts of the code. When possible, language features should be used that allow information to only be available at exactly the parts of the code that need it.

A related idea is that of protecting implementation details

> Implementation details of a particular function or module should be hidden from the rest of the program. One should be able to change the implementation details of a particular function without compromising other parts of the program.

As an example, imagine that we wrote a sum function with the stateful recursion paradigm, and it needed a helper method. we could do this as follows:

```
sumHelper :: Num t => t -> [t] -> t
sumHelper acc []     = acc
sumHelper acc (x:xs) = sumHelper (acc + x) xs

sum :: Num t => [t] -> t
sum xs = sumHelper 0 xs       -- could also simply write: sum = sumHelper 0
```

This works fine. But it also exposes the sumHelper function to the rest of the file. We may for instance be tempted to use taht function directly in other parts of our file. Or some other part of our program might see that function and decide to use it.

This causes problems down the line for the maintainance of our application. We intended sumHelper to only be used by sum as an internal implementation detail: This is how we implemented sum. However, because we provided it as its own function, and now it is available to other parts of my application. And the moment those other parts use it, I can no longer change without compromising those other parts. The function sumHelper is no longer just an implementation detail, it is now part of the functions we have exposed to the rest of the application.

A solution to this is to simply define the helper function within the function sum, using a where clause. This is similar to using local variables within a function to do our work, something we do very often in other languages as well as Haskell. Except that in functional programming languages like Haskell, a local variable can in fact be a whole function in its own right.

So with this in mind, our example above might look like so:

```
sum :: Num t => [t] -> t
sum = sumHelper 0
    where sumHelper acc []     = acc
          sumHelper acc (x:xs) = sumHelper (acc + x) xs
```

This way the function sum is the only one that knows about sumHelper. And it can change it at will without worrying about any repercussions.

Some times however we cannot afford to do that. We may have a helper function that is needed by more than one part of our module. For example perhaps there are 3-4 functions all sharing a common helper function. It would be foolish to type that helper function 3-4 times inside each of these functions.

This is where modules will come in handy. Modules allow us to specify exactly which parts of our file would be available to other modules. And any other parts are isolated to the particular file, and can be shared by all functions within that file but with noone else.

Modules

Modules are files that contain related data types and functions that operate on those types. Each module must have a well defined interface with the rest of the world: It must specify which functions and data types are to be shared with the world.

In particular a module allows us to provide the concept of an abstract data type. We can define a data type but not actually reveal its internal implementation. Then users of our application cannot just directly create elements of our type, they will have to call our explicit constructor functions. We will provide such an example in a moment.

The overall structure of a model definition looks as follows:

```
module Foobar (
    -- specify what you export here

) where
... -- Definitions follow here
```

Let us become familiar with the module format, by creating a module to handle integer fractions. Here is what we should probably share with the world:

- There should be a new “fraction” type that people should be able to work with. We should probably hide how the type is internally represented (so people cannot just create a “divide by zero” fraction directly, without going through our constructor).
- There should be a way for people to create a fraction from an integer, and from a pair of a numerator and a denominator.
- There should be ways to add, divide, multiply, subtract fractions.

- We should also simplify the fractions where appropriate. But users will not need to do that, so that would be an internal function to our module, and would not be exported.
- There should be a way to convert a fraction into a Double.
- There should be a way to turn a fraction into a string, for printing purposes.
- At a later stage, we will see how to make our fractions part of the various type classes, e.g. Num, Show etc.

So with that in mind, our module preamble might looks something like:

```
module Fraction (
    Fraction,       -- Exporting the type
    fraction,       -- build a fraction from a numerator and a denominator
    fractionOfInt,  -- build a fraction from an integer
    add,            -- add two fractions to get a new fraction
    sub,            -- subtract a fraction from another to get a new fraction
    mult,           -- multiply a fraction from another to get a new fraction
    divide,         -- divide a fraction from another to get a new fraction
) where

-- Now we start our definitions
data Fraction = Frac Integer Integer

fraction :: Integer -> Integer -> Fraction
fraction a b | b == 0    = error "Cannot divide by 0"
             | otherwise = simplify $ Frac a b

fractionOfInt :: Integer -> Fraction
fractionOfInt = (`fraction` 1)   -- Same as: fractionOfInt b = fraction b 1
-- Could also have made a "Frac b 1" directly. Why did we not?

-- More functions follow
```

Notice here that we exported Fraction, and not the specific constructor Frac. This means that others cannot use Frac to create fractions, they must call the function fraction instead. This is what is known as an *abstract data type*:

> An **abstract data type** is a type defined via its behavior (semantics) regarding how it can be constructed and what operations it supports, rather than its specific implementation via a specific data structure.

Technically what we have is actually better described as an **opaque data type**.

The fraction function also serves another purpose. It ensures that we never build a fraction with 0 denominator. This is what we call an **invariant**:

> **Invariants** are properties maintained by the functions in a module. The constructors *must* ensure that no initial values are constructed that don't obey these invariants. Other functions can rely on the fact that their inputs *will* obey these invariants, and *must* ensure that their outputs do so as well.

In our instance, the only way to create a fraction is via the fraction constructor. That constructor ensures that we never have a zero denominator in a fraction. The other functions will assume this to be the case.

Let us now look at the other functions in the module. They basically have to follow the standard rules for combining fractions. But we need to simplify things when possible. For instance (1/2) * (2/3) should equal (1/3), not (2/6). This is the goal of the simplify function: It takes a fraction, and simplifies it by finding the greatest common denominator of its numerator and denominator, then dividing them both by it:

```
simplify :: Fraction -> Fraction
simplify (Frac a b) = Frac a' b'
    where d = gcd a b
          a' = a `div` d
          b' = b `div` d
```

This is a function that is private to this module, and not exported to the rest of the application. This is important, as it allows us to modify the function without breaking everything else. In this occasion, we would want to ensure that our denominator is always a positive number:

```
simplify :: Fraction -> Fraction
simplify (Frac a b) = Frac (s*a') (s*b')
    where d = gcd a b
          a' = a `div` d
          b' = b `div` d
          s  = signum b'
```

Finally, let us implement the remaining functions:

```
mult :: Fraction -> Fraction -> Fraction
Frac a b `mult` Frac c d = fraction (a * c) (b * d)

divide :: Fraction -> Fraction -> Fraction
Frac a b `divide` Frac c d = fraction (a * d) (b * d)

add :: Fraction -> Fraction -> Fraction
Frac a b `add` Frac c d = fraction (a * d + b * c) (b * d)

sub :: Fraction -> Fraction -> Fraction
Frac a b `sub` Frac c d = fraction (a * d - b * c) (b * d)
```

Importing a Module

When you want to use a module within another module or your main program, you must *import* it. There are a number of different ways to achieve that:

Simple import All the values and types that the module was exporting become available to you via their names. For example we can do things like:

```
import Fraction
f1 = fraction 2 3  -- Create a new fraction
f1 `add` fraction 4 5
```

Qualified import The values that the module was exporting become available, but only if you prepend them with the module name. This is useful when the functions that the module exports would have clashed with existing names. For example we can do:

```
import qualified Fraction
f1 = Fraction.fraction 2 3  -- Create a new fraction
f2 = f1 `Fraction.add` Fraction.fraction 4 5
```

Import with alias We can do a qualified import with a specified name alias for the module. For example:

```
import qualified Fraction as F
f1 = F.fraction 2 3  -- Create a new fraction
f2 = f1 `F.add` F.fraction 4 5
```

Partial Import We can import only some functions but not others.

```
import Fraction(fraction, mult)   -- only imports these two functions
f1 = fraction 2 3  -- Create a new fraction
f2 = f1 `add` fraction 4 5  -- This will FAIL
```

Import with hide We can import all but some of the functions.

```
import Fraction hiding (add)  -- imports all exported functions except add
f1 = fraction 2 3  -- Create a new fraction
f2 = f1 `add` fraction 4 5  -- This will FAIL
```

Implementing Type Class Instances

It is often desirable to make sure our data types implement a certain type class. For example it would be nice if our fractions behaved like normal numbers, in other words that they were a **type class instance** of the Num type class. Implementing a type class instance is easy. Let us start by seeing how Num is actually defined, in the standard Prelude[1], excluding the comments:

```
class Num a where
    (+), (-), (*)           :: a -> a -> a
    negate                  :: a -> a
    abs                     :: a -> a
    signum                  :: a -> a
    fromInteger             :: Integer -> a
    -- Default definitions of (-) and negate.
    -- If one is provided, the other can be defined
    x - y                   = x + negate y
    negate x                = 0 - x
```

This is how a type class is defined, with the keyword class followed by the class name and type variable. What follows is a series of type declarations for the functions that belong to this class. If a type wants to be an instance of the class, it must implement all of these methods with a specific syntax that we are about to see.

[1] https://hackage.haskell.org/package/base/docs/Prelude.html

Some times however, a class will provide "default implementations" for some of the functions in terms of the others. In our example above, subtraction is defined in terms of addition and negation, and negation is in turn defined in terms of subtractions. What this means in practice is that a type has to only implement subtraction or negation, and then the other one will come for free.

Let us now turn our Fraction type into an instance of Num. Recall that we ensured that the denominator is always positive:

```
instance Num Fraction where
    Frac a b + Frac c d = fraction (a * d + b * c) (b * d)
    Frac a b * Frac c d = fraction (a * c) (b * d)
    negate (Frac a b)   = Frac (negate a) b
    abs    (Frac a b)   = Frac (abs a) b
    signum (Frac a _)   = signum a
    fromInteger a       = Frac a 1
```

Let us continue in the same vein, with definitions for Eq and Ord. First we look at the definitions of these type classes in the prelude (have to dig a bit for them[2]):

```
class  Eq a  where
    (==), (/=)              :: a -> a -> Bool

    x /= y                  = not (x == y)
    x == y                  = not (x /= y)

class  (Eq a) => Ord a  where
    compare                 :: a -> a -> Ordering
    (<), (<=), (>), (>=)    :: a -> a -> Bool
    max, min                :: a -> a -> a

    compare x y = if x == y then EQ
                  else if x <= y then LT
                  else GT

    x <  y = case compare x y of { LT -> True;  _ -> False }
    x <= y = case compare x y of { GT -> False; _ -> True }
    x >  y = case compare x y of { GT -> True;  _ -> False }
    x >= y = case compare x y of { LT -> False; _ -> True }

    max x y = if x <= y then y else x
    min x y = if x <= y then x else y
```

Notice how we only need to specify == or /=, as they each have default implementations in terms of each other. Similarly, compare is all that is needed for Ord a (though we can certainly define more functions if we had concerns about efficiency). We also throw in a "Show" instance.

```
instance Eq Fraction where
    Frac a b == Frac c d  = a == c && b == d

instance Ord Fraction where
    Frac a b `compare` Frac c d  = compare (a * d)  (b * c)
```

[2] https://hackage.haskell.org/package/base-4.4.1.0/docs/src/GHC-Classes.html

```
instance Show Fraction where
    show (Frac a b) = show a ++ "␣/␣" ++ show b
```