CIS552: Advanced Programming

Handout 11

Reminder!

Class Wednesday will be held in Berger Auditorium, where Yaron Minsky will be talking about the use of functional languages at Jane Street Capital.

Haskell Miscellany

Graduation!

You're now familiar with pretty much all of the basic concepts in Haskell.

There is more to learn, of course, but from now on I'm mostly going to let you figure things out as you need to know them.

A Few More Topics We'll Cover Together

- Module system
  - today
- Concurrency
  - after spring break

Other Features to Investigate Yourselves

- Exception handling
  - raise exceptions anywhere
  - handle them in the `IO` monad
- Advanced type class features
  - multi-parameter type classes, functional dependencies, etc.
- Many useful libraries
- Etc.

Haskell Resources

- The Haskell 98 report
- The Haskell Hierarchical Libraries documentation
- The GHC User's Manual

All are easy to find on-line.

It would be a good idea to skim these documents now, just so you have a sense of what's there in case you need it later.

Modularity

The Evils of Duplication

Every time something is duplicated in your program, you've got a bug waiting to happen.

> Principle: Every piece of knowledge should have a single, authoritative representation within a software system.

In their book, *The Pragmatic Programmer*, Hunt and Thomas talk about the DRY principle: Do not Repeat Yourself!

Abstraction

A great deal of duplication involves things that are almost but not quite the same.

Abstraction is the antidote to this kind of duplication.

> Principle: When things start to get duplicated, look for a way of separating ("abstracting out") the common part from the varying parts.

Conceptually, the common part becomes a function and the varying parts become parameters that are supplied to the function to yield particular instances.

Often, the abstracted-out common part will turn out to be interesting in itself and to have other potential uses.

Generality

The temptation to duplicate often arises from trying to program against insufficiently general abstractions.

> Principle: When building an abstraction, try to find a "natural" level of generality that leaves room for applications besides the one you've got in mind right now.

Dangers of Abstraction and Generality

Warning: Although more abstraction is generally a good thing, there are limits!

- Choosing to abstract instead of duplicate often comes with a cost in terms of the complexity / readability of the code.
- Similarly, a more general abstraction is sometimes harder to understand than a more specific one.

> Principle: Use taste when deciding how much to abstract.

It's OK to duplicate something that is really never going to change, or where the "knowledge" being duplicated is very simple.

Example

Which is better?

```
ex1 =
  do putStrLn ("The user's last name is " ++ lastname)
     putStrLn ("The user's first name is " ++ firstname)

ex1' =
  do aux "last name" lastname
     aux "first name" firstname
  where aux x y =
          putStrLn ("The user's " ++ x ++ " is " ++ y)
```

DRY documentation

Duplication occurs not only between different bits of code, but between code and documentation!

Principle: Code should be self-documenting whenever possible.

For example:

- A meaningful variable name is better than an obscure name with a meaningful comment.
- Breaking up a complex function into several bite-sized pieces is better than writing a long explanation of how it works.
- Unnecessary documentation is worse than none at all.

In general, use comments only to explain high-level features of code. Low-level details should usually be clear from the code itself.

Checked Documentation

- Type declarations are a good idea because they are checked documentation: they are useful to readers, but if they get out of date with respect to the code, the compiler will complain.
- Ditto assertions.
- Ditto unit tests.

Principle: Documentation should be checked whenever possible.

Locality of Repetition

Principle: When something must be repeated, the two copies should be located as close as possible to each other — preferably immediately adjacent.

For example, the documentation of a program's switches and preferences (for the user manual) should live in the code, near where the switches themselves are defined and used.

(Example from Unison.)

Components

Complex software systems need to be organized into bite-sized (or brain-sized) chunks, of manageable size and complexity.

These chunks are generally called components or modules.

Loose Coupling

Two components are orthogonal if one can be changed without affecting (or even knowing about!) the other.

This is also known as loose coupling.

Loose coupling is the quality that keeps large systems “light on their feet” — maintainable, extensible, and resilient to change.

Cohesion

The flip side of loose coupling is ensuring that individual components have clearly defined roles.

Principle: Each component of a system should have a single, well-defined purpose.

Interfaces

Another important way of increasing orthogonality is limiting the “surface area” that each module presents to the others in a system.

The interface of a component limits which parts of its internals are exported for the use of other components.

Abstract Data Types

An abstract data type (or ADT) is a component that provides a single (usually) type and a collection of associated operations.

The type is “abstract” in the sense that the component’s interface exposes only the name of the type, not its concrete definition. The only way other modules can do things with values of this type are via the operations exported by the defining component.

(Note the similarity to the notion of class in OO languages.)

The Haskell Module System

Modules

A Haskell program is a collection of modules, one of which, by convention, must be called `Main` and must export the value `main`. The value of the program is the value of the identifier `main` in module `Main`, which must be a computation of type `IO t` for some type `t`. When the program is executed, the computation `main` is performed, and its result (of type `t`) is discarded.

Modules may reference other modules via explicit import declarations, each giving the name of a module to be imported and specifying its entities to be imported. Modules may be mutually recursive.

Module Names

The name-space for modules themselves is flat, with each module being associated with a unique module name (which are Haskell identifiers beginning with a capital letter). There is one distinguished module, `Prelude`, which is imported into all modules by default.

Hierarchical Modules

Module names are allowed to contain the character "`.`" — e.g., `Data.Time.Calendar.Easter` is a valid module name.

Implementations may (and GHC does) choose to interpret dots in module names as instructions for where to look for modules in the file system during compilation.

E.g., the module `Data.Time.Calendar.Easter` might be found in a directory `Data` with a subdirectory `Time` containing a sub-sub-directory `Calendar` containing a file `Easter.hs`.

Module syntax

(See 5.1 in Haskell 98 Report.)

Export lists

(See 5.2 in Haskell 98 Report.)

Import Declarations

(See 5.3 in Haskell 98 Report.)

Abstract Data Types

```
module Stack( StkType, push, pop, empty ) where

data StkType a = EmptyStk | Stk a (StkType a)
push x s = Stk x s
pop (Stk _ s) = s
empty = EmptyStk
```

A nice convention for using ADTs (originally from CLU?) is always to call the main type of the module just `T`.

```
module Stack ( T, push, pop, empty ) where

data T a = EmptyStk | Stk a (T a)

push x s = Stk x s
pop (Stk _ s) = s
empty = EmptyStk
```

The “.T” Convention

Now, if other modules import the `Stack` module with the `qualified` keyword, the name of the abstract type becomes `Stack.T` — nice!

```
module M where

import qualified Stack

myStack :: Stack.T Int
myStack = Stack.pop (Stack.push 5 Stack.empty)
```

The “.T” Convention

To keep programs from getting unreasonably verbose when using this style, the names of modules need to be kept fairly short. So, if `Stack` were actually called `Mystuff.Util.Data.Stack`, we would rename it locally using `as`:

```
...
import qualified Mystuff.Util.Data.Stack as Stack
...
```

Summmary

Haskell’s module system is rather basic, as functional languages go. (For example, OCaml, Standard ML, and PLT Scheme all offer much more sophisticated features.)

However, it can be used together with some programming conventions to get the job done in a great many situations.

For example...

Interfaces

Every Haskell module has an interface, in the sense of a set of facilities (types, values, and classes) that it makes available to the rest of the world.

However, there is no particular place where its interface is specified.

- The names that it exports are listed in the `exports` clause at the top.
- But the `types` of these names are not.

In other words, to know how to use a module, we need to look at its implementation — bad!

Most module systems provide a separate syntactic construct (which often even lives in a separate file) for writing down interfaces, so that users of a module are not even tempted to look at its internals — indeed, they may not be given the source code for its internals, and the internals may not even be written yet.

Interfaces

This situation is not ideal for modular programming, but there are a variety of ways to work with it.

Two common ones:

- Provide interfaces “by convention”
- Extract interfaces from implementations

Interfaces by Convention

Haskell doesn’t provide any way to put a module’s interface in a separate file. But it can at least be collected in a separate part of the file.

```
module Stack ( T, push, pop, empty ) where

-- type T a
empty :: T a
push  :: a -> T a -> T a
pop   :: T a -> T a

-----------------------------------------------
-- Implementation:

data T a = EmptyStk | Stk a (T a)
push x s = Stk x s
pop (Stk _ s) = s
empty = EmptyStk
```

Extracting Interfaces

Another point of view is to leave the type declarations adjacent to their definitions (which avoids editing two parts of the file when things change) and instead using a tool to extract the interface from the module.

Indeed, we can go a step further, keeping not only the type signature but also the documentation for interface functions.

(Haddock demo.)