LiU-FP2010 Part II: Mini Haskell Tutorial

May 2010

Henrik Nilsson
School of Computer Science
University of Nottingham

May 12, 2010

1 Introduction

This is a small Haskell tutorial suitable for people with some background in functional programming from other statically typed functional languages like ML or OCaml. The scope is very limited: it only aims at providing some basic familiarity with on-line resources for Haskell, the interactive Haskell system GHCi, and basic Haskell syntax. The following is a short list of recommended papers and tutorials that are useful as complements or alternatives for getting started with Haskell:

- John Hughes. *Why functional programming matters.*
  `http://www.cs.chalmers.se/ rjmh/Papers/whyfp.html`
  [Classic must-read.]
- *Haskell in 5 Steps*
  `http://www.haskell.org/haskellwiki/Haskell_in_5_steps`
  [Concrete advice on how to get started]
- Paul Hudak, John Peterson, Joseph Fasel. *A Gentle Introduction to Haskell*
  `http://www.haskell.org/tutorial/`
  [Classic tutorial]
- Miran Lipovača. *Learn You a Haskell for Great Good!* `http://learnyouahaskell.com/`
  [Beautiful! Worth checking out for that reason alone.]

- *Hitchhikers guide to Haskell*
  `http://www.haskell.org/haskellwiki/Hitchhikers_guide_to_Haskell`
  [More advanced.]

This tutorial assumes you have access to a working Haskell system. If not, see *Haskell in 5 Steps* above.

2 Getting Started

Task 1 Start the interactive GHC environment by issuing the command `ghci` at the command line prompt:

```
tuck$ ghci
```

Some information about GHCi gets printed, and you'll then get a new prompt:

```
Prelude>
```

The text before `>` are the names of the modules whose definitions are in scope. Thus the above prompt means that everything in the Haskell standard Prelude is available.

Now evaluate some simple expressions. E.g.

```
Prelude> 1 + 2
Prelude> "Hello World!"
Prelude> putStrLn "Hello World!"
Prelude> "Hello\nWorld!\n"
Prelude> putStrLn "Hello\nWorld!\n"
```

Make sure you become familiar with the command-line editing facilities to save on typing. For example, try out the arrow keys and various Emacs bindings.

Beside expressions, which get evaluated when entered, GHCi accepts a number of commands. They are all prefixed by `:` to distinguish them from Haskell expressions. The command `:help` prints a list of available commands. Try it now. Note that commands may be abbreviated to save on typing. For example `:h` is enough to get help. Try it. What is the command to load a Haskell module (and recursively all modules it depends on) from a file? What is the command for leaving GHCi? Try it now, and then restart GHCi.

3 On-line Resources

Task 2 Visit `www.haskell.org` and locate:

1. The tutorial *Haskell in 5 steps*.
2. The page on learning Haskell.
3. The page on books and tutorials.
4. The GHC documentation, in particular the section on GHCi.

Another very useful on-line resource is *Hoogle*: A Haskell API search engine allowing many libraries to be searched by function name or *approximate type signature*. The latter can be extremely useful for locating functions when you know roughly what their types must look like, but haven't got a clue what such a function might be called, or in what library it might be defined.

Task 3 Visit `www.haskell.org/hoogle`.

1. Find a function that given a predicate on an arbitrary type (a function returning `Bool`) and a list of elements of the same type forms a list of those elements satisfying the predicate.
2. What other functions with the same type signature did you find?
3. Try to locate some other useful functions on lists; for example: a function for computing the length of a list, functions for turning a list of pairs into a pair of lists and vice versa, functions for reducing a list by replacing each cons by a binary operation and the empty list by an element of an appropriate type w.r.t. the binary operation.

4 Basic Programmming Exercises

Task 4 Using a text editor of your choice, define a module called `Tree`. The GHC convention is that there should be one module per file, and that the name of the file should be the same as the name of the module defined in it with an additional `.hs` extension. (This is how GHC can find the definitions of a module given just its name.) In your case the file should be called `Tree.hs`.

The module `Tree` should contain a data declaration for a tree with three data constructors representing:

- an empty tree

- a singleton tree (a leaf)
- a non-empty tree consisting of two subtrees

Both the singleton tree node and the interior tree nodes should carry a single `Int` value. Call the constructors `Empty`, `Leaf`, and `Node`, for example.

Load the module into GHCi. If there are errors, fix them and try again. Note how the prompt changes to indicate that the definitions in the module `Tree` now are in scope. Which are they? What are their types? (Hint: try the command `:type`.) As the module `Prelude` is implicitly imported into every module unless explicitly hidden, all Prelude definitions are still available. (The `*` in the prompt `*Tree>` means that the Prelude is in scope.)

Construct some `Tree` values. Can you print them? Can you compare them using the operators `==` or `<`? If not, fix the problem (hint: make use of a `deriving` declaration) and reload the module.

The command `:show modules` shows all loaded modules. Try it. Switch back so that only Prelude definitions are in scope again. Command `:module Prelude` (or just `:m Prelude`). Are your `Tree` type and data constructors now available?

You can still get at your definitions by using their *fully qualified names*. That is, by prefixing the name of a defined entity with the name of the module in which it is defined. For example, if one of your `Tree` data constructors is called `Node`, it is available as `Tree.Node`. Try this. This works because GHCi as an extra convenience implicitly imports the definitions of all loaded modules under their fully qualified names into all scopes. (When a module is compiled by any Haskell 98 compiler, like GHC, all used definitions from other modules must be explicitly imported in one way or another, except for those from the Prelude.) Now switch back to the `Tree` scope.

Task 5 Generalise the your `Tree` definition so that the tree can carry data of an arbitrary type. That is, make it *polymorphic*. Load the new definition into GHCi and test it. What are the types of the `Tree` data constructors now? Make sure you understand them!

Using your polymorphic data constructors, create trees of integers, characters, and strings. Check that you can print and compare them and that they have the expected type.

Task 6 Create a new module called `Main` (in the file `Main.hs`). Import the module `Tree` into this module.

Define a function `size` that returns the size of a tree. The size of the tree is the number of values carried by the tree.

Load the new module into GHCi and test it on trees of a few different sizes. (Hint: you may want to define a number of test trees under some convenient names, either in the module `Main` or in a separate third module of containing test data that you import into `Main`.)

Check which modules are loaded now. Switch between the different module scopes and figure out which definitions are available where (without giving their fully qualified names).

Define a function `insert` that inserts a value at the right place in an ordered tree. A tree is ordered if all values in the left sub tree is strictly smaller than the value in the top node, and all values in the right subtree is strictly greater than the value of the top node. A particular value occurs at most once in a tree.

Load the new version of `Main`. What is the type of `insert`. Why? Add an explicit type signature to the definition of `insert` as documentation!

5 Labelled fields

Task 7 Change your `Tree` definition so that it uses named (also called *labelled*) fields. This is Haskell's version of records. Let the name for all value fields be `value`. Let the names for the left and right subtree be `left` and `right` respectively.

Load the new definition. Verify that you can construct trees using the named field notation, and that the order among the fields does not matter when you do so. For example, assuming the constructors are called `Empty`, `Leaf`, and `Node`:

```
*Main> Node {left=Leaf {value=1}, value=2, right=Empty}
*Main> Node {right=Empty, left=Leaf {value=1}, value=2}
```

Verify that you still can construct trees using the normal way of applying the constructor functions (i.e. with positional arguments). For example

```
*Main> Node (Leaf 1) 2 Empty
```

Note that the result tree still gets printed using the named field notation.

What happens if you don't provide values for all fields? Or indeed for no field? For example, what are the results of the following? Why?

```
*Main> :type Node {left=Empty}
*Main> :type Node {}
*Main> Node {left=Empty}
*Main> Node {}
```

Verify that you automatically got selector functions `value`, `left`, and `right`, that these have the expected types, and that you can use them to pick trees apart. What happens if you apply these selector functions to trees with the wrong top-level constructor, such as:

```
*Main> value Empty
*Main> left (Leaf 2)
```

Explain.

Redefine your `size` and `insert` functions so that they make use of the field names when pattern matching. Verify that you can match the fields in any order. Also note that you easily can omit field names that are of no interest (e.g. `value` when you're defining the function `size`). It may also make sense to leave out field names that are only of interest in certain conditional branches of the code to make the pattern matching clearer and draw attention to what is most important for selecting the right conditional branch. The remaining fields can always be accessed using the field selectors as and where needed.

6 Scope Rules

Task 8 This is an exercise on understanding Haskell's scope rules. In the following code fragments, draw an arrow from each *use* of a variables (`x`, `y`, etc.) to its *defining occurrence*, if it has one in the provided fragment. For example for a code fragment like

```
let x = 3 in x + x
```

you would draw an arrow from each of the `x`'s in the expression `x + x` to the `x` in `x = 3` as that is the corresponding defining occurrence. Note that a particular variable *name*, like `x`, may be used for more than one variable even within the scope of a definition for the variable in question since inner definitions are allowed to *shadow* outer definitions in Haskell. For example, the value of the Haskell expression

```
let x = 7 in
   (let x = 3 in x + x) * x
```

is 42. Note that the following fragments are not examples of good style Haskell code! They have deliberately been made somewhat confusing to make a good exercise on Haskell's scope rules.

1.
```
f xs ys =
    let xs = x : xs in take 10 (ys ++ xs)
    where
        x = head xs
```

2.
```
f x y =
    let n = 3 in take n (g y) ++ take n (g x)
    where
        g x = take n xys
            where
                xys = x : yxs
                yxs = y : xys
        n = 10
```

3.
```
f xxs@(x:xs) =
    case xs of
        []      -> [x] : take n (repeat xs)
        (x:xs)  -> [x] : take n (repeat xs)
    where
        n = length xxs
```