Haskell Learning

Joseph Sumabat

Contents

1 Introduction

This document is for me to document my learning particularly with respect to Haskell.

Disclaimer: This is just my understanding of information that I've read about the language and some of the theory behind it. I am by no means a mathematician and there may be innacuracies in my understandings (please let me know if there are!) so don't use this as a source or anything like that.

1.1 Resources

Listed here are any resources I used or have found useful in my learning process

1.1.1 Books

- Category theory for programmers by Bartosz Milewski
- Type Theory and Functional Programming by Simon Thompson

1.1.2 Wikis

- Haskell Wiki
  - The category Hask
- Wikipedia
  - Intuitionistic type theory
  - Brouwer–Heyting–Kolmogorov interpretation

1.1.3 Blogs

- Hask is not a category by Andrej Bauer
- The Algebra (and calculus!) of Algebraic Data types by John Burget

1.1.4 Papers

- Fast and loose reasoning is morally correct by Jeremy Gibbons

1.1.5 Other

- Classical and Constructive Logic by Jeremy Avigad

2 Types

2.1 Hask

Hask is ostensibly the category of Haskell types[1]. That is to say "Objects of Hask are Haskell Types" and should follow the category laws of associativity and identitycategory laws of associativity and identity

Haskell types can be **thought of** like sets [2]. e.g. `Int` could be thought of as the set of values from $\{-2^{29},..,2^{29}-1\}$ and `Boolean` as the set containing two elements $\{True, False\}$. With GADTs (sum types and product types) we can compose types together to form more complex types (e.g. the two element `Boolean` type can be thought of as the sum of two unit types.

2.1.1 Bottom Type

These types actually contain an additional value of $\perp$ which is a value **included in every type** allowing for computations which don't terminate. $\perp$ traditionally is used to represent a type with **no** values (which `Void` is meant to represent in Haskell) and is thus unprovable constructively.

$\perp$ in Haskell as a value is necessary because of the existance of general recursion and is witnessed by the equation $x = x$. The inclusion of it within every type is necessary for the language to be turing complete.

2.1.2 Hask is not a category

Because of the existance of both $\perp$ and the fact that Haskell is a non-strict language with the function `seq` defined, Hask actually violates the category laws of category theory.

We usually think of working within a limited subset without either bottom values or `seq` so that we can apply category theorestic principles.

2.2 GADTs and type algebras

2.3 Type Theory and logic

2.3.1 Constructive vs Classical logic

In classical logic every proposition is assigned a value T or F **this is not the case for constructive logic** In constructive logic contradiction does not suffice for proof. we lose out on this property. Particularly this means we lose the law

[1]See Hask is not a categroy

[2]There is a distinction between set and object of category hask (or set), there's also a distinction between sets and types but I like the comparison and Bartosz Milewski uses it in Category theory for programmers

of excluded middle and by extension proofs by contradiction. Instead in order to prove things we must show that a proposition is *witnessed*

Philosophically the difference is in a focus not on the truth or falsehood of a statement ("truth in the abstract") but on the verification and its properties. .Since we can no longer prove things indirectly through methods such as contradiction but must actually provide a witness *which must be computable.*

Constructive logic is a subset of

- No law of excluded middle
- Negation is different $\neg A$ in classical logic corresponds to $A \rightarrow \bot$
- BHK interpretation o

2.3.2 Constructive Type Theory

Martin-Löf type theory named after its inventor is based on the principles of constructive logic. Wikipedia gives the example of proving that there is a prime greater than

2.4 Curry Howard Isomorphism

Isomorphism between types and Proofs. Haskell types have a mapping to **Propositional** Logic. More powerful type systems can map to more powerful systems of logic (Dependent types can correspond to predicate logic for example) , Martin-Löf was the first to make the extension to predicate logic through Martin-Löf type theory.

In the propositional sense we say that a propositional formula is true if it is *inhabited*. Consider the type `Integer`, a function of this type such as

```
one::Integer
one = 1
```

proves the that the type is inhabited by providing a value which witnesses the type since 1 is an inhabitant of the type `Integer`. This method of looking at programs as proofs makes it clear why constructive rather than classical logic is employed: since programs are what we are concerned with and programs are constructions, constructive logic forces us to prove our propositional formulas in terms of programs.

- Think of a Haskell type as a set and a program as a proof that such a set is inhabited
- From my understanding: This means the compiler also functions as a proof checker

- From a practical standpoint this makes verification of programs based on the type system more concrete

Note that the propositional logic system that Haskell types map to is **unsound** due to the bottom type ($\perp$) which inhabits every type including void. Consider the following example:

```
absurd::() -> Void
absurd a = undefined
```

Because the void type corresponds to $False$ and the unit type to $True$ we have $True \rightarrow False$ which is clearly unsound. However we note that the bottom type terminates the program so I'm not entirely sure whether you can consider it a value.

2.5 More powerful type systems (dependent types)

3 Language and Examples

This section is largely to use as a reference for specific syntax structures that are common throughout the language

3.1 Making types

- Haskell has generalized algebraic datatypes (GADTs)
  - sum type using `|` and product type as multiple parameters to a type constructor
- Types can be recursively defined as below:

Sum type example:

```
data Boolean = True | False
```

Product Type example:

```
data Pair a = Pair a a
```

Recursive Type example:

```
data Tree a = Node a Tree Tree | Leaf a
```

3.2 Typeclasses

Paremetric polymorphism is done through typeclasses. Typeclasses define implmentation of a function over members of the typeclass

Example functor typeclass with fmap

```
class  Functor f  where
    fmap :: (a -> b) -> f a -> f b
```

Example functor instance for a `MyList` type

```
instance Functor MyList where
    fmap f (Cons n rest) = Cons (f n) (fmap f rest)
    fmap f None = None
```

3.3 Modules

Module in root src dir (everything exported)

```
module MyModule where
```

Module in root src dir (construcotr of type and functions exported)

```
module MyModule (MyType(MyConstructor), func1, func2) where
```

Module in root src dir (all construcotrs of type and functions exported)

```
module MyModule (MyType(..), func1, func2) where
```

Module in directory MyDir

```
module MyDir.MyModule (func1, func2) where
```

3.4 Imports

Importing functions Module

3.5 Applicative Syntax