Haskell Learning

Joseph Sumabat

Contents

1 Introduction

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

1.1 Resources

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

- Category theory for programmers by Bartosz Milewski
- Hask is not a category by Andrej Bauer
- Haskell Wiki
  - The category Hask
- Fast and loose reasoning is morally correct by Jeremy Gibbons
- The Algebra (and calculus!) of Algebraic Data types by John Burget
- Type Theory and Functional Programming by Simon Thompson

2 Types

2.1 Hask

"Objects of Hask are Haskell Types".

Haskell types can be **thought of** like sets [1]. 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 $\{True, False\}$

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.

$\perp$ is necessary because of the existance of general recursion and is witnessed by the equation $x = x$.

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.

[1] There is a distinction between set and object of category hask (or set)

2.2 Constructive vs Classical logic

Philosophically the difference is in having to prove things via a construction

In classical logic every proposition is assigned a value T or F by law of excluded middle (think truth tables). **this is not the case for constructive logic** In constructive logic we lose out on law of excluded middle and by extension proofs by contradiction. Instead in order to prove things we must show that a proposition is *witnessed* (in this context inhabited).

-

2.3 Curry Howard Isomorphism

Isomorphism between types and Proofs. Haskell types have a mapping to **Propositional** Logic Note that more powerful type systems can map to more powerful systems of logic (Dependent types can correspond to predicate logic for example)

- Think of a Haskell type as a set and a program as a proof that such a set is inhabited
- From my understanding: Your compiler also functions as a proof checker
- From a practical standpoint this means that
- Note: Haskell

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 $True \to False$ 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.

2.4 More powerful type systems (dependent types)

3 Language and Examples

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 Applicatives