Type level programming in Haskell

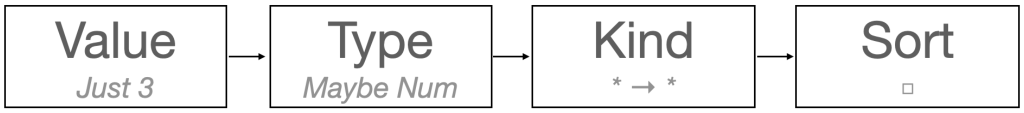
Felix Bowyer

Introduction

Haskell is a purely functional, statically typed programming language. Part of what makes it so powerful is its powerful type system.
As with most programming languages, functions in Haskell have type signatures, which tell us what values a function takes, and what type it returns. For example, the + operator is of type Num $\rightarrow$ Num $\rightarrow$ Num,[*] which tells us that it accepts two values of type Num, and returns a third Num. The cons operator (:) has type signature a $\rightarrow$ [a] $\rightarrow$ [a]. Here, a is a type variable used to define it polymorphically for any type within a list.

In Haskell, types can be thought of as also having types, in the form of kinds (sometimes notated as a calligraphic X). Concrete types, which have values themselves, have the kind * (pronounced "star"). Type constructors, which take types as a parameter, have more complex kind signatures. For example, the Maybe monad, which is parametrised over a type, has the kind signature * $\rightarrow$ *; it takes one concrete type as a parameter, and returns another concrete type. The type Maybe Int has kind *.

Considering these facts, it is possible to use Haskell's type system itself as a mode of computation. If we were to have some way to give types themselves value, then we could treat Haskell kinds as types, and Haskell types as values. We can call this type-level programming. Because Haskell is statically typed, all type resolution, and therefore all type-level computation, takes place at compile time.



Haskell also has one layer above kinds, the sort. Everything in Haskell has the same sort, notated as $\square$.

The first paper we will discuss is Type-level Instant Insanity, which aims to teach Haskell from a type-level perspective by example, by reimplementing a usual Haskell program entirely at the type level. After, we will discuss Giving Haskell a promotion, which proposes a language extension which aims to improve the kind system with custom, polymorphic kinds.

[*] In Haskell's instance context notation: (Num a) $\Rightarrow$ a $\rightarrow$ a $\rightarrow$ a

Type Level Instant Insanity[1]

This paper aims to introduce type-level computational concepts by example. It starts by specifying a usual Haskell solution to the problem Instant Insanity, introduced in Bird and Wadler's Introduction to Functional Programming.[2] Instant Insanity is a problem involving cubes with differently coloured sides and applying transformations to them. This solution is then translated to the type level, introducing a number of analogues between value level and type level programming to do so.

In standard Haskell, established types can be used to represent different data structures. For example, in the Instant Insanity solution, strings are used to represent the cubes, as a list of characters which represent the colours of each side. In type-level Haskell, we instead define atomic constants to represent different structures.

To ensure clarity, in this section the phrase "type-level value" will be used to refer to a standard Haskell type, but through the lens of using it as a value in a type-level function. The phrase "type-level function" refers to something which accepts one or more type-level values, and returns another type-level value, much like a usual function does with usual values at the value level.
The term "usual" will be used to describe non-type level concepts.

The Data keyword can be used to define atomic constants. This includes representations for concrete values (of type *), as well as value constructors, which take other values to instantiate another value (of kind * $\rightarrow$ *). For example, one representation of positive integers at the type level could be:

```
data Z -- concrete type: kind *, zero
data S n -- concrete type: kind * -> *, successor of n
```

When introduced in this way, these constants don't mean anything. The above definitions may represent numbers, but they don't have any semantic meaning until we give them meaning by defining functions in which they get used as numbers.

The class keyword introduces a type-level function signature. Continuing our previous example, we can define a type-level function to get the successor to a type-level number.

```
class Succ i1 i | i1 -> i
```

Type-level functions look slightly different to usual functions, and work more like Prolog's function declarations. There is no concept of the type-level return value; instead, the final parameter of a type-level function is repurposed as a return value.

At the type level, Haskell's overloaded functional dependency operator $\rightarrow$ can be read as specifying a type-level value dependency. This is our way of telling Haskell that the type-level value i is derived from i1, and i is the value we intend to calculate.

The instance keyword defines implementations of our type-level function for different values. Given that at the type level we have to define our own terms, it is often useful to define implementations inductively.

```
instance Succ Z (S Z)
instance Succ (S x) (S (S x))
```

We firstly define our base case, Succ Z. The final parameter is repurposed as a return value, so Succ Z evaluates to S Z. To cover non-zero values, we define Succ (S x) as evaluating to the same with an extra wrapped S, S(S x).

To call type-level functions, the type checker must be invoked on a usual Haskell value. In GHCI, this is done by using :type. Before we can do this, we must instantiate some usual values with the type of our type-level functions (since they are equivalent to usual types). Continuing our example, for succ we do this as:

```
succ :: Succ i1 i => i1 -> i
succ = const undefined
-- :type succ (undefined:(S Z)) yields: S (S Z)
```

Since we do not have any usual type constructors which create values with the type of our type-level functions, we pass our type-level functions to the type checker via undefined, which is unique in being able to be instantiated as any type.

It would be nice to be able to translate more complex functions, such as recursive functions, to the type level. This is possible using Haskell's instance context notation with ⇒. A recursive function adding two numbers together at the type level, partially reproduced from the Haskell Wiki,[3] looks like:

```
class Add i1 i2 i | i1, i2 -> i, i1 i -> i2
instance Add Z i2 i2
instance (Add i1 i2 i) => Add (S i1) i2 (S i)
```

In usual Haskell, the ⇒ operator adds a type constraint (to the left) to an expression (on the right). Therefore, at the type-level, this can be thought of a value constraint, where the value of i on the right is derived from a type-level function call as described by the left.

The paper recognises that implementing everything at the type level is not the ideal way to implement Haskell programs. However, it acknowledges that by learning how to toy with the type system as its own language, one can gain a much better understanding of the language as a whole. The author states that: 'familiarity with the syntax of the Haskell Type System is a prerequisite for general Haskell programming.'

Overall, the paper does a good job of introducing essential type-level concepts in an understandable way. By introducing concepts by example, as has been done in this essay, it makes it much easier to see the practical use of certain aspects of the language.

In doing this, it focuses away from the theoretical workings of the Haskell type system, and skims over some useful details, such as why the functional dependency operator is necessary when defining a function signature. Including details like these would have made it easier for a reader to construct their own type level functions.

Additionally, it somewhat downplays the utility of type level programming. It can allow programmers to perform checks at compile time, rather than runtime, making it a very useful tool for ensuring runtime safety of code. This could have been mentioned.
It also does not discuss the limitations highlighted by the paper described below.

Giving Haskell a promotion[4]

This paper specifies an extension to Haskell's kind system to make it closer in expressiveness to the type system. Particularly, it focuses on two aspects – introducing a mechanism for programmers to define their own kinds, and introducing kind polymorphism. (These are achieved by promoting these features from the type system.) This paper is the basis for the modern DataKinds and PolyKinds GHC extensions.

Consider the following implementation for a length-indexed vector at the type level, reproduced from the paper.

```
data Zero
data Succ n

data Vec :: * → * → * where
Nil :: Vec a Zero
Cons :: a → Vec a n → Vec a (Succ n)
```

Haskell doesn't allow programmers to define their own kinds. All concrete types have kind *, and type constructors (type level functions which take one or more types as parameters and return a type) have kind * → * for each parameter. This is much less expressive than values and functions at the value level. For the above example, this would accept nonsensical values, such as Succ bool. Therefore, to make the kind system more expressive, we need to be able to define our own kinds.

The proposed extension allows this by automatically promoting data types to data kinds when defined.

```
data Nat = Zero | Succ Nat
```

As in standard Haskell, this defines a type Nat, which can have the value Zero or Succ (Some other nat). However, with the extension, this also defines types 'Zero and 'Succ, which have kind Nat. This allows us to write type-level functions which only accept certain types, based on the associated auto-promoted kind.
A reimplementation of the above, from the paper:

```
data Vec :: * → Nat → * where
VNil :: Vec a 'Zero
VCons :: a → Vec a n → Vec a ('Succ n)
```

Types such as Succ bool cannot be instantiated now, since Succ is constrained to being instantiated by some type variable of kind Nat.

The extension also introduces polymorphism for kinds. Polymorphic kinds are introduced by promoting parametrised types. In addition, when a declaration has no kind signature, Haskell automatically infers a polymorphic kind.
An example from the paper:

```
data HList :: [*] → * where
HNil :: HList '[ ]
HCons :: a → HList as → HList (a : as)
```

[X] is a parametrised type – it represents any list containing items with kind *. This can be promoted, and type-level functions acting on HList get a polymorphic type. HNil :: forall k. [k] The same applies for any promoted type-polymorphic declaration.

The combination of these two features brings Haskell's kind system much closer to its type system. However, it is not quite as powerful, and cannot be without overhauling the sort system.

Exactly which type constructors can be automatically promoted to kinds is defined formally in the description of the proposed intermediate representation, System $f_c \uparrow$. Principally, types are only promoted if promoting them will preserve the singular sort in Haskell. Introducing multiple sorts is far beyond the aims of this extension. To quote:

> Rule KV_LIFT states that T $\bar{\kappa}$ is a valid kind only if T is a fully applied type constructor of kind $*^n \rightarrow *$

$$\frac{\Gamma \vdash_{\mathsf{k}} \kappa_1 : \square \quad .. \quad \Gamma \vdash_{\mathsf{k}} \kappa_n : \square \qquad \varnothing \vdash_{\mathsf{ty}} T \, : \, \star^n \rightarrow \star}{\Gamma \vdash_{\mathsf{k}} T\,\overline{\kappa} : \square} \quad \text{KV\_LIFT}$$

This means that some useful automatic type promotions, such as those that would result in higher-order kinds of the form (* → *) → *, cannot be promoted. Types are also not promoted if they themselves are defined in terms of other promoted types, or are a type with a polymorphic kind. This restriction is kept to prevent complicating the sort system.

Some more rules concerning promotion of types are reproduced below from figure 9.

$$\frac{a \mapsto \mathcal{X} \in \Theta}{\Theta \vdash a \leadsto \mathcal{X}} \quad \text{L\_VAR}$$

$$\frac{\Theta \vdash \tau_1 \leadsto \kappa_1 \quad \Theta \vdash \tau_2 \leadsto \kappa_2}{\Theta \vdash \tau_1 \rightarrow \tau_2 \leadsto \kappa_1 \rightarrow \kappa_2} \quad \text{L\_ARR}$$

$$\frac{\Theta, a \mapsto \mathcal{X} \vdash \tau \leadsto \kappa}{\Theta \vdash \forall a{:}\, \star\,.\,\tau \leadsto \forall \mathcal{X}.\,\kappa} \quad \text{L\_ABS}$$

$$\frac{\varnothing \vdash_{\mathsf{ty}} T \, : \, \star^n \rightarrow \star \quad \overline{\Theta \vdash \tau_i \leadsto \kappa_i}^{\,n}}{\Theta \vdash T\,\overline{\tau} \leadsto T\,\overline{\kappa}} \quad \text{L\_APP}$$

The paper does an excellent job at explaining problems with the lack of flexibility in Haskell's kind system. Without the proposed extension, type level programming is effectively untyped itself, allowing nonsensical type-level function calls to be valid, which return gibberish. Having the ability to add kind constraints makes type-level programming much more safe and closer to usual programming.

The paper is difficult to approach – a lot of notation is introduced without context, and a lot is presented without much explanation. It also makes no attempt to introduce type level programming concepts to the reader, therefore I found the first paper very useful to understand this paper. The paper states that their intended audience is “the community of Designers and Implementors of Typed Languages”, therefore this is more acceptable. Despite being written by the implementors and designers of the system it describes, the paper does a good job of motivating its design choices and justifying where and why the authors set certain limitations.

Thoughts

Overall, reading and understanding the papers has taught me a lot about programming in Haskell and other functional languages by re-introducing some crucial language features from a different perspective. When learning to program in a functional language having come from an imperative one, it is easy to focus on the imperative-like features and try to build an understanding of the rest from there. By taking away any hint of imperativeness, learning about type-level programming has forced me to learn how type-oriented functional languages are and how this effects the programming style.

The papers also introduced me to a few issues I hadn’t come across before. As mentioned, one use of type-level programming is to perform checks on run-time function calls at compile type, preventing run-time errors. Being able to do some form of execution at compile time can help ensure safety in programs when used correctly.

The two analysed papers were quite different, especially with respect to their target audience. The first was more focused on introducing new concepts to the reader, whereas the second was focused on those who are already familiar with the concepts and providing a natural extension to them. Having read both, I feel that my knowledge about type-level programming, and type systems overall, has been broadened.

The papers are surprisingly applicable outside of the direct domain of Haskell. More mainstream languages, such as Python and JavaScript, have been incorporating functional features for a while now, and newer languages such as Rust are gaining popularity partly due to the integration of functional features from the ground up. C++ has its own metaprogramming features, which can functionally mirror Haskell’s. The second paper could act as a blueprint for how other languages can extend these features in the future.

Overall, I have learnt a lot more about the inner workings of type systems, and look forward to how the gaining popularity of these features will change the way we program.

Notes and References

[1] Conrad Parker. Type-level instant insanity. *The Monad. Reader,(8)*, 2007.

[2] Richard Bird and Philip Wadler. Introduction to functional programming. series in computer science. *Prentice Hall International*, 20, 1988.

[3] Haskell wiki; type arithmetic. Online; `https://wiki.haskell.org/Type_arithmetic`. Accessed 01/12/22.

[4] Brent A Yorgey, Stephanie Weirich, Julien Cretin, Simon Peyton Jones, Dimitrios Vytiniotis, and José Pedro Magalhães. Giving haskell a promotion. In *Proceedings of the 8th ACM SIGPLAN Workshop on Types in Language Design and Implementation*, pages 53–66, 2012.

[5] Thomas Hallgren. Fun with functional dependencies. In *Proc Joint CS/CE Winter Meeting, Chalmers Univerity, Varberg, Sweden*, 2001.

[6] Kingsbury, kyle. typing the technical interview. Online; `https://aphyr.com/posts/342-typing-the-technical-interview`, 2017. Accessed 01/12/22.