Haskell Programming Language: Typing Things With Class

Artem Pelenitsyn

Haskell programming language is probably the only practical pure functional language at hand. "Practical" here means many things, including efficient compiler, comprehensive collection of libraries, well-developed documentation, strong community. The reason why Haskell excites people, however, lies not always in those things, but often in language's purity and the way it is ensured, namely types.

Haskell is all about the types. They are here even when not visible, thanks to solid type inference algorithm, which allows for omission of types. Indeed, we can usually skip type annotation, but often do not, and for a good reason: while types is a common tool to restrict programs, in Haskell it serve many other purposes. Let us name a few.

You frequently find type declarations for top-level functions as a mere documentation, as types can tell about a program more than many words. Plus types do not lie (words in comments at times do). `Hlint`, a Haskell code style checker and a tool of thumb for a seasoned Haskell programmer, kindly reminds you to add top-level type annotation when you forget one. Do agree with it.

The notion of type driven development has spread around recently. Once you created type annotation, you can develop against it. Types suggest you what to do and when (during declaration of function parameters or building resulting value). The whole way is clear with the algebraic data types (ADT) at the very heart of the Haskell type system. Being simple sum of products of other types, ADT allows for straightforward case analysis of input values leading to desired output; type checker tells if the latter is sensible. So develop with types as your guide.

Not only types facilitate your coding, but they can relieve you from a good portion of it. Types can direct a compiler or other tool to build routine fragments of code. Telling a compiler, when your type is a clear instance of monoid, saves you a time for defining its neutral element and associative operation. There is a small amount of fundamental type operators (parametric types) with well known properties and expressive power summarized in renowned *Typeclassopedia*; studying it can help in understanding a fair amount of code behind the curtains of compiler-generated or librarian primitives used widely. For those who did not manage to get through this functorish tao and stick to good old sums of products of ints, and bools, and others, there are still ways to benefit from generative facilities; there is a whole kit of various generic programming approaches helping you to "scrap your boilerplate" (one of them entitled just like that).

We started with the notion of purity, and now let us turn to the question of what does it mean to be pure functional programming language (and what it has to do with types). We probably cannot even print to console in a pure functional language, can we? Certainly, there are ways to produce side effects in Haskell code; the notable thing about that code is again its typing. Types will tell, when effects are involved. This means you can always tell a function, which calculates an integer based on its arguments, from a function returning an integer taken from `stdin` — just by looking at their types.

Haskell is careful about various kinds of effects; there is good assortment of them: reading a configuration, logging an execution, computing in parallel, and others. These options are clearly distinguished in types, bringing even more power to them. We can't stand noting, however, that the ways for combining effects is not that elegant as other bits of Haskell type discipline; an undergoing research targets improvement in this.

As we see, types can change many aspects of development: documentation, coding, testing, refactoring. The ultimate aim of these changes is to rise the quality of software being produced. We consider Haskell a prominent tool to achieve this.