Practical Type System Benefits

by Neil Brown ⟨neil@twistedsquare.com⟩

April 23, 2014

One of Haskell's key selling points is its type system. This article provides several practical examples of how the type system helped while writing some data analysis software. The article will touch on monads, concurrency, invariants, generic programming, and quasi-quoters.

The Project

As part of a research project, we are collecting a large volume of Java source code which we want to analyse[1]. At the time of writing, we have 50 gigabytes of source code snapshots from six months' worth of recording. To perform the analysis, we have a machine with two six-core hyperthreaded Xeon processors and 32 gigabytes of RAM. I might term this "medium data": small enough to fit on one machine, but big enough that you can make multiple cups of tea while the analysis is running.

I felt that for writing this analysis, Haskell was a good choice. This was mainly because of its easy support for generic programming and parallel processing, as will be explained in the course of this article. I thought my experiences may be of interest to others, highlighting practical uses of Haskell's powerful type system in my project. The article assumes general familiarity with Haskell, and is aimed at someone who has learned a little but wants to see what the advantages are of learning more.

[1]For more details, see "Blackbox: A Large Scale Repository of Novice Programmers' Activity" by Brown et al. (2014), available via `http://twistedsquare.com/publications.html`

Generic Programming

One of the main reasons that I felt Haskell would give me an advantage for performing source code analysis was its easy support for generic programming[2]. For example, one small analysis I wanted to perform was to look for **if** statements in Java like this:

```
if (x < 5);
{
    x = 5;
}
```

Note the crucial semi-colon after the if condition; a semi-colon is a valid statement in Java, so the semi-colon becomes the if body. Thus the if statement is pointless, and the body is executed regardless of whether x is less than five. Effectively, the compiler sees this version:

```
if (x < 5)
    ; // if has empty body

// This is now a totally separate block of code that is executed next:
{
    x = 5;
}
```

This is not a compile-time or run-time error, but without an else clause, I think it can always be termed a programmer mistake. The first thing to do in looking for this mistake was to parse the source into an Abstract Syntax Tree (AST) – for this, I used (a slightly augmented version of) the `language-java` library from Hackage. The relevant portion of the AST data type is this[3]:

```
data Stmt = Empty | IfThen Exp Stmt | IfThenElse Exp Stmt Stmt | ...
```

Exp is the data type for expressions, while Stmt is the data type for statements. You can see that Stmt has a constructor for the empty statement (i.e., the solitary semi-colon). So my empty statement will be a value that looks like IfThen something Empty. That's easy to pattern-match for, given a specific Stmt. Here's the code:

```
emptyIf :: Stmt -> Bool
```

[2]Generic programming refers to programming that works with multiple types. Although they share the same name and concept, Java's generics are a fairly restrictive form of generic programming. An example in Haskell might be a generic fold, which can fold the contents of a heterogenous container together.

[3]To be clear: the triple dots are omitted code, not Haskell syntax!

```
emptyIf (IfThen _ Empty) = True
emptyIf _ = False
```

However, a full Java AST has the type CompilationUnit and can have Stmt values in all sorts of places. For example, Java's support for anonymous inner classes means that statements can be inside method declarations that are inside a class definition in an expression which itself could be in another inner class. This is where generic programming enters the picture.

Generic programming can be used to automatically find all values of one type (e.g. Stmt) inside a complex structure containing many different types (e.g. CompilationUnit). The `listifyDepth :: (b -> Bool) -> a -> [b]` function finds a list of all values of type b (Stmt) within the given value of type a (CompilationUnit) that satisfy the given function. This particular function is implemented in the Alloy generic programming library, but the exact same function exists in Scrap Your Boilerplate (named `listify`), and similar functions are available in other generic programming libraries.

With this function, augmenting our program to look across the whole CompilationUnit structure is very easy:

```
hasEmptyIf :: CompilationUnit -> Bool
hasEmptyIf = not . null . listifyDepth emptyIf

-- OR:

findEmptyIfs :: CompilationUnit -> [Stmt]
findEmptyIfs = listifyDepth emptyIf
```

Generic programming is particularly easy in the Haskell language, and in general, dealing with tree structures like ASTs is much easier in Haskell than in other languages like Java.

Concurrency and Parallelism

Being a pure functional language, Haskell has great support for functional parallelism: evaluating several pure functions in parallel. However, it also has good support for concurrency: running several imperative threads of control alongside each other. In our application, the analysis is almost always in a map-reduce style:

1. Read some initial state from the database to determine our list of tasks.
2. Run a large batch of tasks that each fetch data from the database (or some files with cached intermediate state) and then process the data.
3. Collate the results together.

The middle step is amenable to being run across many processor cores, but since it requires imperative access to the database or file system, we must use imperative forms of concurrency (i.e. forkIO). The classic problem with parallel programming is synchronising and coordinating access to mutable variables. In the default pure functional style in Haskell, this is not an issue. However, there are still problem with synchronising concurrent access to *external resources*, such as the database connection or files. It is easy, at first, to accidentally write code like this:

```
do conn <- newConnection
   mapM_ forkIO [doTask i conn | i <- [0..9]]
```

This will unsafely use a single connection in parallel from many tasks, but will produce no compile-time error. In a language like C, where a thread is run from a different top-level function, the values shared between threads are more explicit, and so resource- sharing mistakes can sometimes be easier to pick up. However, Haskell makes it so easy to write inline parallelism that it can actually be less obvious in Haskell that values like our connection handle are being improperly shared between threads.

I found that the best way to avoid this problem was to build the connection into a monad, using a reader monad (called DB). (A reader monad encapsulates some read-only state, which you can read or use via a monadic action.) This both avoided passing the connection parameter around (it could just be accessed from the monad), and meant that I could build a parallel function that would automatically open a new database connection for each new thread. The parallel function had a simple API:

```
parallel :: [DB a] -> DB [a]
```

This API neatly hid the complexity of connection management. The function started up a bunch of new worker threads (one per core) and opened a new connection for each thread. Then each thread picked worker tasks from a list and executed them, before returning the results. With lists of tasks being 10,000s, opening a database connection per worker thread (of which there were 24) was much lower overhead than opening one per task. I have since generalised the functionality and released it as part of the `parallel-tasks` library on Hackage: `http://hackage.haskell.org/package/parallel-tasks`.

Models

The database that our application is reading from was originally created by a Ruby on Rails application. Rails uses a relational database as a persistence layer to store objects that it calls “models”. Each model has a column for each of its fields. The primary key, “id”, is an auto-incrementing integer, and thus foreign

keys are also integers. Our user and session models might be stored in a database table something like this:

User	
Field Name	Field Type
id	64-bit integer
uuid	string
created_at	datetime

Session	
Field Name	Field Type
id	64-bit integer
user_id	64-bit integer
created_at	datetime

A naive approach to dealing with this data from Haskell is to declare an ID type alias:

```
type Id = Int64
```

You might get these from a query from the `mysql-simple` library:

```
do (sessionId :: Id, userId :: Id) <- query conn "select s.id, u.id
    from sessions as s, users as u inner join on s.user_id = u.id"
```

You might then have utility functions like these:

```
getNewUsersSince :: Date -> DB [Id]

getUserInfo :: Id -> DB User

getSessionListForUser :: Id -> DB [Id]

getSessionInfo :: Id -> DB Session
```

With all these `Id` fields flying around, it is only a matter of time before you do something like this:

```
do us <- getNewUsersSince d
   mapM (\u -> do ss <- getSessionListForUser
                  mapM getSessionInfo us {- whoops! -}) us
```

Instead of passing a list of session IDs to `mapM getSessionInfo`, we passed user IDs. But as they all have the same type, so this type checks. To make matters worse, the database is fine with this too: it won't even produce an error, just wrong results. This small example is slightly contrived; but these things do happen, especially if the function took a list of user IDs and a list of session IDs – it's easy to get the parameter order wrong.

The solution is to give each ID its own type, to prevent users from mixing them up. A first attempt might look like this:

```
newtype UserId = UserId Int64
newtype SessionId = SessionId Int64

data User = User { userId :: UserId,
                   userUuid :: String,
                   userCreatedAt :: Datetime }
data Session = Session { sessionId :: SessionId,
                         sessionCreatedAt :: Datetime }

getNewUsersSince :: Date -> DB [UserId]

getUserInfo :: UserId -> DB User

getSessionListForUser :: UserId -> DB [SessionId]

getSessionInfo :: SessionId -> DB Session
```

The behaviour of these functions is now more apparent from their type, and there is greater type-safety, with less possibility to mix up a UserId and SessionId. We now turn to look at how these types are used in the context of database queries. One way to use these types is to apply them directly after the database query:

```
do (rawSessionId, rawUserId) <- query conn "select s.id, u.id
     from sessions as s, users as u inner join on s.user_id = u.id"
   let sessionId = SessionId rawSessionId
       userId = UserId rawUserId

-- OR:

do (sessionId, userId) <- (SessionId *** UserId) <$> query conn "select
     s.id, u.id from sessions as s, users as u inner join on s.user_id = u.id"
```

A better solution is available if you know what the query function does. It has a selection of type-classes like Result which support conversion from the database wire protocol into recognisable Haskell types (Int32, String, etc). We don't actually need to know what exactly they do or how they work – using the GHC GeneralizedNewtypeDeriving language feature we can just tell the compiler to port them from Int64 to our newtype wrapper of Int64:

```
newtype UserId = UserId Int64 deriving (..., Result, ...)
newtype SessionId = SessionId Int64 deriving (..., Result, ...)
```

This allows us to extract these types directly from the database query without wrapping them ourselves:

```
do (sessionId :: SessionId, userId :: UserId) <- query conn "select
    s.id, u.id from sessions as s, users as u inner join on s.user_id = u.id"
```

(You can omit those type declarations if type inference can infer them from future statements.)

Increasing Type Safety

We have already prevented some mistakes by preventing SessionIds from being mixed up with UserIds. However, there is still potential for mistakes in our program around the MySQL queries. A MySQL query is one of those instances where you are dealing with external data, and just as when you read from a file or a socket, there is the possibility to make a mismatch with the schema or protocol. The error cannot be picked up by the compiler (unless you go to greater lengths by converting the schema/protocol into a type or structure that the compiler understands). For example, here's a possible error:

```
do (sessionId :: SessionId, userId :: UserId) <- query conn "select
    u.id, s.id from sessions as s, users as u inner join on s.user_id = u.id"
```

Do you notice the problem? The query is asking for the user ID and then a session ID (in that specific order), but the code labels the results as being a session ID and then a user ID. We are back to the previous problem: a mixup in parameters that cannot be detected by the compiler. In general, I find that several problems in Haskell can be solved by involving the type system further[4]. What I really want to do here is add a type annotation *into the MySQL queries themselves*. We can do this using quasi-quoters – I will briefly explain what quasi-quoters are, and then go on to show how I use them with MySQL queries.

Interlude: quasi-quoters

A quasi-quoter is a pre-compile step that is a bit like a pre-processor; it takes a String input and, *at compile-time*, parses it and spits out source code which is then fed into the full compilation. To illustrate this, consider the case when you want to print out some text, some of which comes from a string, some from an integer, and some from the program itself. In C, I would probably use printf for this purpose:

```
char name[256];
```

[4]The trick is to involve the type system in ways that provide a big benefit for the cost, but not going so far that the type system gets in the way of code that you know – but can't easily prove to the compiler – is safe

```
int yearOfBirth;
// ... ask user for name and year of birth
printf("Hi %s, you will reach %d years old this year",
       name, currentYear - yearOfBirth);
```

In Haskell, you might write it like this:

```
do name <- ...
   yearOfBirth <- ...
   putStrLn ("Hi " ++ name ++ ", you will reach " ++
     show (currentYear - yearOfBirth) ++ " years old this year")
```

Haskell code that is uglier than its C equivalent? Dang. A simple quasi-quoter allows you to write a string with embedded code (a la Perl, Ruby and many others). The substitution is done at compile-time (not at run-time using eval or similar), so errors in the code in the string cause compile-time errors. Using the `here` package from Hackage, you can write:

```
do name <- ...
   yearOfBirth <- ...
   putStrLn [i|Hi ${name}, you will reach ${currentYear - yearOfBirth}
     years old this year|]
```

The quasiquoting syntax consists of a square bracket, the name of the quasi-quoter function, a pipe, and then the string to be fed to the quasi-quoting function at compile-time, which is terminated by a pipe and square bracket. I'm surprised there aren't more quasi-quoters around (e.g., one for processing regexes at compile-time).

Back to our code

Let's now return to our problem of adding type annotations to MySQL queries. My quasi-quoter, qquery, takes a query String like this, with type annotations inside it:

```
[qquery| select s.id{SessionId}, u.id{UserId} from sessions as s, users as u
   inner join on s.user_id = u.id|]
```

It then turns it into the same String with the annotations removed and placed into the type of the query, i.e.:

```
(mkQuery "select s.id, u.id from sessions as s, users as u inner join
  on s.user_id = u.id" :: Query (SessionId, UserId))
```

Thus, if we now re-write our existing code:

```
do (userId, sessionId) <- query conn [qquery| select s.id{SessionId},
    u.id{UserId} from sessions as s, users as u
    inner join on s.user_id = u.id |]
```

The types are now annotated *exactly where they are used* inside the query string. The above code will now give a compile-time error because of the mismatch in the ordering between user ID and session ID. I have packaged this type-annotation quasi-quoter into a library on Hackage (`http://hackage.haskell.org/package/mysql-simple-quasi`), so you can use it if you wish.

As a general observation: these benefits do not come completely for free (the quasiquoter had to be implemented), but once you have the extra capability to add more type information, I believe it pays off time and time again in program clarity and the ability to avoid mistakes as early as possible (compile-time). Since I have a growing ecosystem of a core central library and lots of quickly hacked together analysis programs to explore data, the benefits of extra safety in the core libraries keep paying off as I use them in my small other programs.

Types as Light-Weight Contracts

After all that database talk, I will end with a simpler example of using types to your advantage. Often, you will know a property or pre-requisite of a value, e.g. that a particular integer is (or should be) non-negative. In my program I pull out sorted lists from the database (or sort them myself). Later, I want to merge these lists into a single sorted list. If I am playing it safe, I have to write the merge functions like so:

```
mergeLists :: [a] -> [a] -> [a]
mergeLists xs ys = mergeSorted (sort xs) (sort ys)
```

If the list is already sorted, as is usually the case in my program (because I use "ORDER BY" in the database query), I am unnecessarily attempting to sort the lists again. (This is not just slow because it makes an extra pass of the list, but it also forces evaluation of the list, which may interfere with laziness.) Instead, I can annotate that the lists are sorted by making a new type for this:

```
newtype SortedList a = Sorted [a] deriving (Eq, Ord, ...)
```

The type is exported opaquely: outside the module, the only way to create a sorted list is via the exported `sortList` function, or a few other generators:

```
sortList :: [a] -> SortedList a
sortList = Sorted . sort

enumFromToSorted :: a -> a -> SortedList a
```

```
enumFromToSorted x y = Sorted (enumFromTo x y)
```

I can then write an $O(n)$ function to merge two sorted lists and be sure that I will never accidentally call it on an unsorted list:

```
mergeSorted :: SortedList a -> SortedList a -> SortedList a
```

This technique can also be used in other languages (e.g., defining a new collection class for sorted lists). But Haskell's newtype wrappers make it very easy to add a new type, with all the features of the old type (by deriving type-classes automatically), and some more features added or removed[5]

Conclusion

Often, when people talk about the advantages of a particular programming language, they talk in generalities. Pure functions make your code better; garbage collection lets you write less code; monads are really useful. If you know nothing or little of the language, these general benefits are often hard to understand. I hope that in this article I have given some small, understandable, concrete examples of how some of Haskell's type features can be useful in a real programming project – without turning into one of those infamously baffling monad tutorials.

[5] You can see more discussion, and a more complicated example of this for array index checking, at Oleg's site: `http://okmij.org/ftp/Haskell/types.html#branding`