GP Analysis of the Hackage Database

Nikolaos Bezirgiannis
Christiaan Floor

Utrecht University
12 November, 2011

Introduction

Generic programming is a relatively new concept. A number of libraries for using GP have been published in the last ten years. Those libraries vary in the way they approach the concept of Generic programming and they all have their strengths and weaknesses.

In our project three main questions arose on which we will elaborate later on in this report. First of all: What is the number of libraries that use GP on Hackage? With this we mean does a library import and use any GP-libraries. A second question is: What do those libraries use GP for? So what is the purpose of using GP in their library. This is something we have to check manually, and is difficult to determine automatically. The last question is: What are the most “popular” (i.e. most used) GP-libraries on Hackage? There are a number of GP-libraries used in the libraries on Hackage. But since we have a deadline to meet and not all libraries are used a significant amount of times we will focus on two libraries in this project. Namely SYB and Uniplate. Later on in this report we will tell how we came to this decision.

The structure of this report is that we first discuss our approach for the analysis we have done to get our answers on the questions mentioned above. Then we tell what our results were in the form of percentages and conclusions we got from these. We will also tell about the implementation of our project and problems that arose during development. And finally we will tell possibilities for future work and then a conclusion.

Analysis

To do an analysis on the Hackage database we have splitted it up in two parts. Those are automatic analysis and the manual analysis. First we will discuss the automatic analysis and then the manual analysis.

Automatic analysis

The automatic analysis is also splitted up into two parts. Namely Deriving analysis and Function analysis. We searched for the number of deriving occurrences and the number of function-calls that, we thought, indicate the use of generic programming. Because we focused on the SYB and Uniplate libraries we first determined which functions were used in these libraries. The set of functions resulting from this were used in the function analysis.

Deriving analysis

For the deriving analysis we looked at the number of deriving-occurrences in all modules of every library in the Hackage database. We made a distinction between possible ways to derive some class. The first way is to do datatype-deriving. With this we mean that when a datatype is created a class is derived for it immediately.

```
Data D = DI Int | DC Char deriving (Show)
```

The second way is to do standalone deriving. If some datatype is created we don't derive in the definition of the datatype but the deriving is done somewhere else, possibly in another module. So if we have the datatype definition for datatype D as described in figure 1 we can also derive Eq for this by defining:

```
deriving instance Show D
```

We can also count newtype-deriving. When we have a newtype declaration like

```
newtype Foo = Foo Int
```

This is a kind of wrapper around an Int-type. An Int derives some classes automatically in Haskell.

Some of which are Eq, Ord and Show. But a property of newtypes is that they don't derive all the classes of the wrapped type. So in this case this means Foo does not derive Show, Eq etc. by default. We can say for example we want to inherit the Show instance of Int for Foo by deriving it explicitly for the new type:

newtype Foo = Foo Int deriving (Show)

Note that this has a different meaning then deriving Show for a datatype. We derive Show here for Foo. But it is the Show of type Int we inherit.

Overloading occurrences are also counted. In Haskell98 there are six derivable classes: Eq, Ord, Show, Read, Enum and Bounded. But instead of deriving one of these classes for your datatype you can also write your own instances for this datatype. So if we wanted to create a custom instance of the Ord class for the D-datatype instead of just deriving it we can do

instance Ord D where..

So the result for this analysis will be the number of datatype derivings, standalone derivings, newtype derivings and overloading occurrences.

Function analysis

For the function analysis we first created the list of function names to search for by looking at the SYB and Uniplate libraries. When we have this list of function names we simple look at every expression in all modules and record it when we come across an occurrence of a function we were looking for.

We not only count the number of occurrences of each specified function, but we also record the context of the occurrence. Every time we have an occurrence of a function-name we are looking for we mark the module-name and the source-position in the module for the current occurrence. This information can help us to easily track where an occurrence is recorded.

The result of this analysis is the number of occurrences of each function we have specified in the input of the analysis, together with the module name and the source-position for each occurrence.

Manual analysis

When we have done the automatic analysis we will get numbers as a result. From these numbers we can draw some conclusions. But this is not enough. So we also did a manual analysis to figure out for what reason GP is used in some libraries. We didn't choose libraries just looking at the name of the library or it's purpose.

To select the libraries we were going to check manually we used the result of the automatic analysis. Some libraries use more GP-functions than others. We looked at the libraries that used a significant amount of GP-functions. And we also looked at which GP-functions the libraries used. So we used the results of the automatic analysis as a guide for the manual analysis.

Approach

The approach we have used for executing the analysis was different for the Deriving analysis and the Function analysis. Because for the D analysis we had to look at all the libraries in the Hackage DB and for the F analysis we only had to look at a subset of libraries. Namely the ones that use SYB or Uniplate.

The first thing that needed to be done was downloading the Hackage DB to our system and unpack all the libraries since they are zipped when you download the DB. When everything was unpacked we used a parser for Haskell-files, HaskellSrcExts. The result of the parse is an Abstract Syntax Tree for every module that is parseable. On these ASTs we applied the D analysis. And the

result we stored in a text file, so we could use the data retrieved from executing the analysis later on.

For the F analysis the approach was slightly different. Because we only had to execute this analysis on a subset of the libraries in the Hackage DB we first had to parse the cabal file of every library to see if they indeed used GP-libraries. When we got the numbers from this, and we saw that SYB and Uniplate were the most used GP-libraries. Then we ran the F analysis over this subset of libraries and also stored the result in a text file.

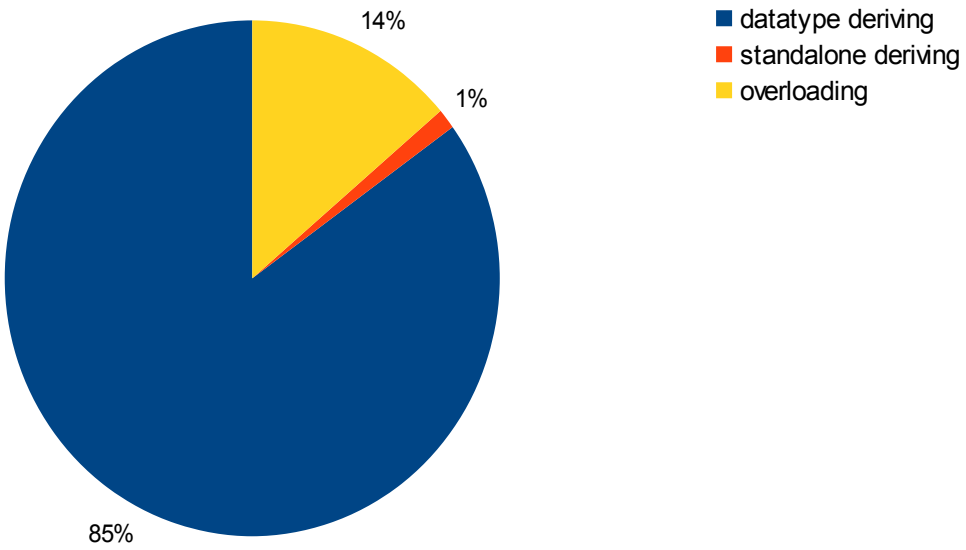
From the automatic analysis we had two text files to work with. So we got all the numbers we were interested in from these text files and draw some conclusions from this. After this we looked manually at some libraries that use SYB and some libraries that use Uniplate. This gave us more information about the way GP is used in those libraries.

Results

In this part we will present what the results were from executing our analysis together with conclusion we could draw from this. Also we give the results for the manual analysis. The D analysis was executed on approximately 3.000 libraries and the F analysis was executed over 105 libraries from which 80 libraries used SYB and 25 used Uniplate for GP.

D Analysis

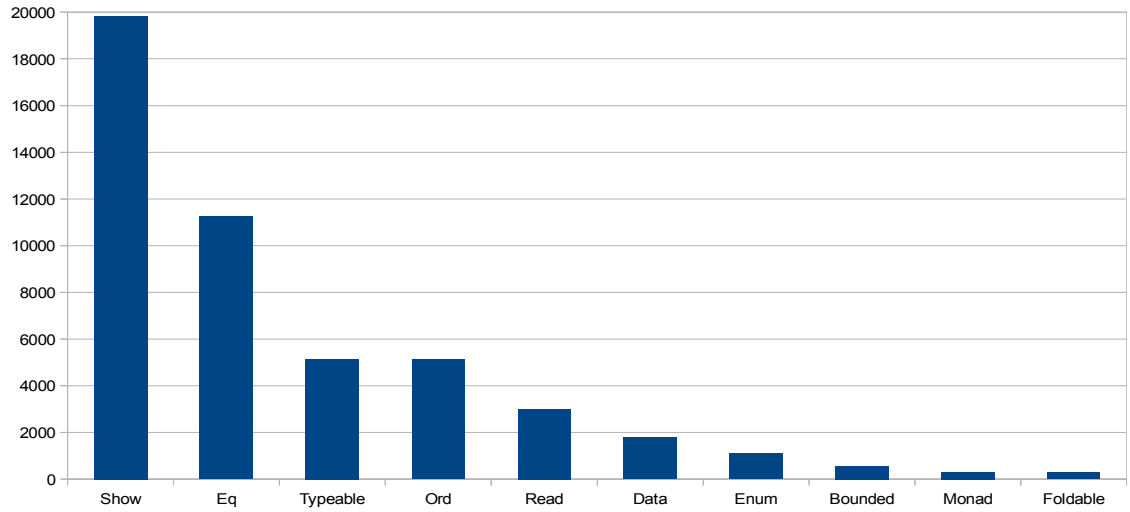
From the D analysis we got a total number of instances of 51093 and the following division:



Here you can see that the datatype deriving is by far the most popular deriving mechanism. Standalone deriving is not used very often. From this we can say that overall programmers are satisfied with the instances derived for a datatype in a library they import in most cases. Overloading of deriveable instances is used quite often. So people want to create their own instances of one of the 6 derivable classes of Haskell98 for datatypes regularly.

We also counted the number of newtype-derivings (these are not included in the number of instances (51093)) mentioned above. And we found a number of 2554. So when newtypes are used often also the deriving mechanism is used for newtypes to inherit the derived instances of the wrapped type.

The top ten of derived instances is as follows:



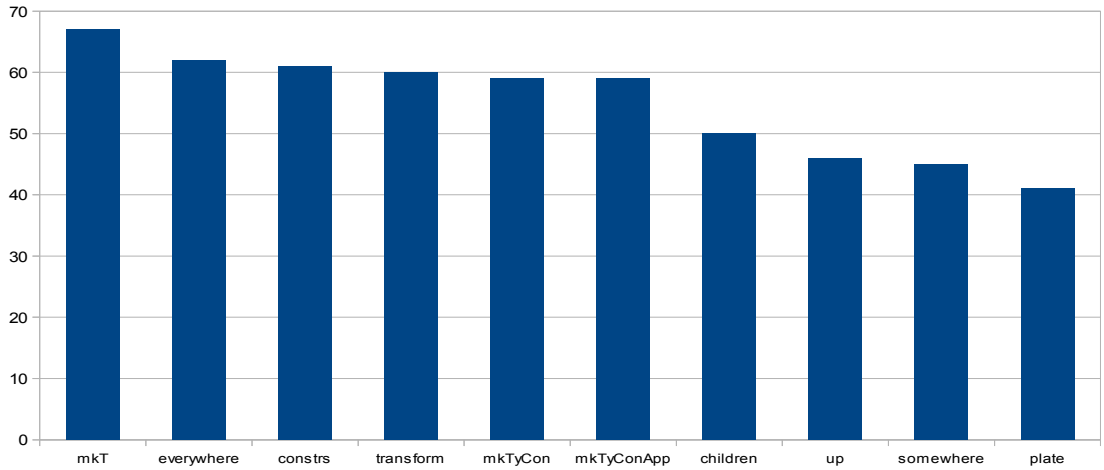
The instances that are of our interest especially from this top ten are Typeable and Data. Because they are an indication for the use of GP. Of course it is possible to derive Data and Typeable and not use GP at all in a library. But we think this is highly unlikeable. From the total number of instances detected, Typeable and Data cover 14 %. From all the libraries in Hackage Typeable instances are used more often than Data instances. The ratio between the use of the two classes is 74% / 26 %.

Also Generic instances are counted. The Generic class can be derived automatically from GHC version 7.2 and is also an indication that GP is used. We found a mere 95 occurrences of derivings for this class. The main reason for this is probably that it is a relatively new feature. GHC 7.2 has been released in August 2011. We expect that this number will grow over the coming year.

F Analysis

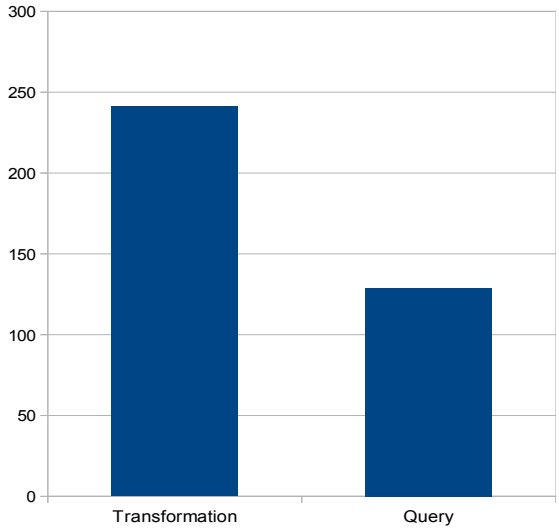
For the F Analysis we first had to determine how much libraries use GP. We found that 136 libraries use GP-libraries. From those 136, 80 use SYB and 25 use Uniplate. So, for the F Analysis we looked at those 105 libraries.

We created a list of function-names we searched for, which can be seen in the source-code for both SYB and Uniplate. The total number of GP functions we found was 1268. After executing our analysis the top ten of used GP-functions looked like this:



Some functions are used for executing queries and other functions are used for executing transformations on a datastructure. Again we created a list of functions we thought indicate the use of queries and a list of functions that indicate the use of transformations and we came up with the

following result:



Here we can see that transformations (65 %) on datastructures are used more than queries (35 %). For the lists of functions we used to get this result we advise to look at our source-code. In some libraries only queries or only transformations are used. Of course queries and transformations can also be used together in a library. More about this will be in the results of the manual analysis.

Manual Analysis

For the manual analysis we looked at five libraries. We looked at three libraries that use SYB and two that use Uniplate. Based on the numbers we got from the automatic analysis and the purposes of the libraries we decided at which we should look.

For SYB we looked at the prolog, preprocessor-tools and JSContracts packages. For Uniplate we looked at the Hoogle and Derive packages.

Prolog

The Prolog library is an intepreter for Prolog written in Haskell. In this library Prolog terms need to be parsed. It uses GP for checking if a Prolog term is present in another Prolog term. The reason for this is checking if a substitution in a term is valid.

All the Prolog terms are parsed to an Abstract Syntax Tree. So we have an AST for the term for which we want to check if it contains a given sub-term. This sub-term is also parsed to an AST. A query is executed that checks if a given sub-AST is present in the AST.

Another thing GP is used for is transforming variable-nodes in an AST. Prolog terms contain variables. When the term is parsed we get the AST of the term. In this AST we have nodes that represent the variables in a term. Each variable node contains a depth-attribute that represents the depth of the node in the tree. They use GP for annotating each variable node with a depth in the AST.

Pre-processor tools

The pre-processor tools package extends the Haskell syntax with a custom pre-processor. In this library they parse Haskell modules. Again you will get an AST as a result.

The first use of transformations is replacing (transforming) certain nodes in the AST. All the parts of the AST that are marked by the pre-processor symbol are replaced by an AST which represents the new code that should come in the place of a pre-processor region of the Haskell code.

Each node in the AST has a source-position attached to it. They execute a transformation on each node by setting all the line-numbers and column-numbers of the source-positions to -1. In other words: they reset all the source-positions in the AST.

JsContracts

This is a package for working with javascript and templates.

In this package transformations are used to replace variables in templates with actual Javascript code. The Javascript template has been parsed to an AST and in the template certain variables need to be replaced with code. So the variable nodes in the AST are replaced with ASTs that represent the Javascript code. This is similar to the purpose for which transformations were used in the pre-processor tools package.

Hoogle

The Hoogle search engine is extremely popular among the Haskell community and it has become the entry point for documentation and API lookup of packages hosted at Hackage. Users can search either by function names or type signatures.

It is rather obvious that Hoogle does a lot of parsing Haskell sources and for that reason GP seems appropriate to remove boilerplate code when traversing ASTs.

Since the tool is focusing mostly on the types, there are transformations and queries specific for type manipulation. Example of this is AliasFollowing, where a type is traversed and matched with an alias of it (e.g. String ↔ [Char]). Another example is EditDistance, i.e. the edits of the user-input type necessary to match another type. Again, GP is put to use to make it easier to edit the type.

Along these lines, an internal graph representation of types is used, called TypeGraphs. Hoogle defines transformations and queries on the type graphs to create, show and search through them.

Uniplate functions are also used for rendering the results in a particular format style (mostly web stuff), prior to showing them to the user.

Derive

The derive package is a library as well as a tool to automatically derive instances for Haskell programs. It provides a list of support derivations. This list can of course be extended by the user.

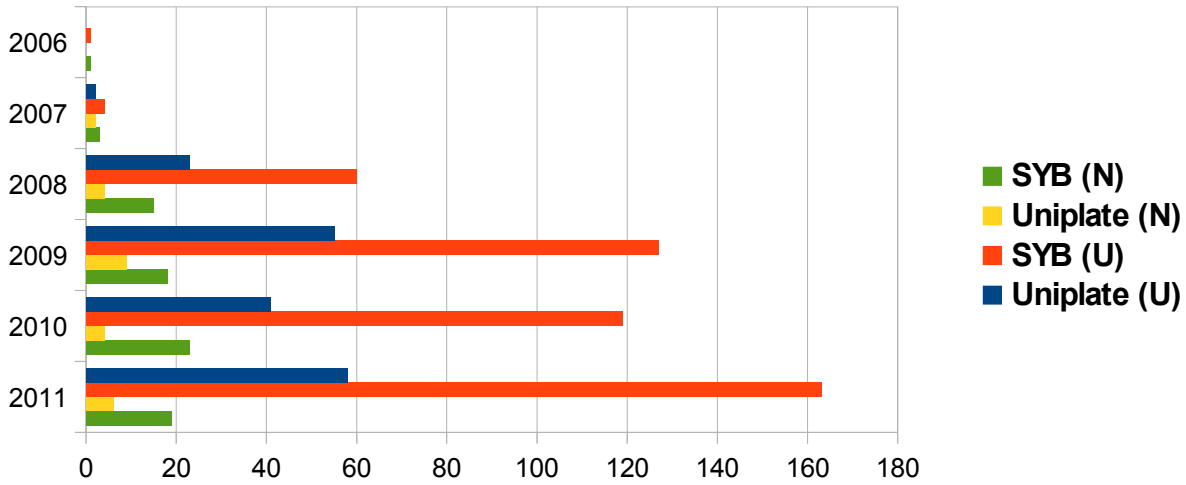
The library parses Haskell source code using HSE. The input is, then, a typed AST that has to be transformed to an untyped AST, before the deriving mechanism takes place. The conversion is done using Uniplate.

The program uses an internal DSL to represent the "logic" behind the deriving and for that transformations are used to construct, simplify and prettyprint instances of the DSL. Transformations are applied also for optimizing on the TemplateHaskell code.

On the level of types, GP is used for making type substitutions and imposing restrictions on what kind of types can derived. For instance, functions and non-monomorphic datatypes (the ones that include type variables) cannot, at all, be derived. Other applications include removing and transforming pieces of types.

History

Here we show how many new releases and updates of libraries have been done in Hackage that use SYB or Uniplate from the years 2006 till now. Notice that on the day of writing this, 2011 is not over yet.



Overall we can seen that the number of new releases and updates for SYB are growing each year. The number of new libraries that use Uniplate is not growing. Updates for Uniplate libraries are growing slightly.

Implementation

For the implementation of the project we decided to use GP ourselves to write the Analysis code. The decision was twofold: firstly because our code involved a lot of parsing syntax trees, and hence subject to possible boilerplate removal, and secondly because we wanted to "eat our own dog food".

The structure of the program is somewhat different of what would be expected. Instead of providing one executable that would be responsible for the whole analysing process, we decided to split up to many executables, so as to keep things separate and simple. Of course this is expected to introduce duplication of code and analysis work, but in the end it is easier for the user to run part of the analysis that he/she prefers.

For most analyses results we went for Data.Map, since it provided a rich API and somehow fits the particular situation.

The kind of analysis we make falls down to 5 categories: Deriving, Function, Error, Uni, Misc. Based on that we created the library's structure.

Deriving Analysis

The parsing of the DerivingAnalysis looks at the top level of each module for "interesting" datatype, newtype or instance declarations and it accumulates them to the analysis results.

The result then has the folllowing type:

```
type Analysis = M.Map ClassName (M.Map ModuleFileName (M.Map DataName (LineNumber, DerivingType)))
```

The Deriving Type as it was mentioned before is defined as:

```
data DerivingType = Normal | StandAlone | Overload
```

Despite keeping track of the names of deriveable classes, we mark the module name and line

number, so it would be handier for us and for others to manually analyse later selected packages.

A snippet of the Parsing module that make use of GP (specifically SYB):

```
countDerivingClasses :: GenericQ (M.Map ClassName (M.Map DataName (LineNumber, DerivingType)))
countDerivingClasses x = everything (M.unionWith M.union) (M.empty `mkQ` derivingDecl) x
```

It is the main driving force of parsing and it's there to remove boilerplate code, as well as providing recursion on the AST.

The Mining module includes mostly Map-related functions to manipulate the result of the deriving analysis. Example is this function that counts the occurrences of a specific Deriving Type:

```
countDerType :: DerivingType -> Analysis -> Int
countDerType d a = M.fold
        (\ m acc -> M.fold ((+) .
                M.size .
                M.filter ((== d)
                        . snd))
            acc m)
        0 a
countNormal = countDerType Normal
countStandAlone = countDerType StandAlone
countOverload = countDerType Overload
```

Function Analysis

It should be mentioned again that although DerivingAnalysis runs on the whole Hackage, FunctionAnalysis only runs on SYB- and Uniplate- dependent libraries. That was one of the reason for splitting into many executables.

The work necessary for Function analysing a module is multiple times bigger than that of analysing usage of deriving, since the whole AST (not only its top level) must be traversed for function calls related to GP.

The type of the FunctionAnalysis results is:

```
type Analysis    = M.Map FavFunction (M.Map ModuleFileName Occs)
```

A list of functions of interest (FavFunction) was hardcoded so it can be searched for in the AST.

For the parsing of functions, SYB is used again for easily recursing and quering the AST. Example:

```
collectOccs :: Module -> FavFunction -> Int
collectOccs m n = everything (+) (0 `mkQ` findN n) m
```

The Mining module includes code for getting out the selected information. The example is self-explanatory:

```
topFunctions :: Analysis -> [(String,Int)]
topFunctions res = L.sortBy eqTups $ L.sort $ M.toList $ M.map (foldr (+) 0) $ M.map M.elems res
```

Uniplate Analysis

We created seperated modules for analyzing some aspects of code written in Uniplate. The primary reason was to track the type of Uniplate and Biplate deriving taking place. We came up with this datatype:

```
data UniStyle = ManualSpeed | Automatic | ManualPointless | Mixed
```

This made it possible to categorize and comment on how the UniplateDeriving was laid out.

ErrorAnalysis

The code was added, in an ad-hoc fashion, to figure out which Hackage modules were failing to parse using HSE. The results of this analysis are mentioned later in the Pitfalls section.

Since this code runs on the whole Hackage database, it seems appropriate to integrate it with the DerivingAnalysis, so as to avoid one extra run on Hackage. This is left for future work.

GeneralAnalysis

Modules were created that don't fall exactly on the previous categories, but are rather more general. Their results are not essential to the rest of the analysis but were actually written for advising us on where we should focus our analysis on.

The Hackage module returns interesting information about the usage of GP, reading just the cabal files. The Date module parses instead the hackage.log and creates a timeline of the Syb and Uniplate Hackage usage.

Pitfalls

- Lazy IO: We were bitten early by the lazy io mechanism of Haskell, during the parsing phase of the modules. Specifically, we were reading module files using the readFile, which actually leaves the handler open so to lazily read the contents. In the end, we had thousands of opened handlers.
- Wrong encoding format of modules: There were a lot of Hackage modules that were not only using a different character encoding than UTF-8, but also mixing characters from different encodings. Then, the Prelude.readFile was failing to read the module files. The problem was solved by using Codec.Text.Iconv and discarding unreadable characters.
- Running out of memory: This was happening on the DerivingAnalysis, because we were doing a whole Hackage analysis and we were gathering a huge amount of information. The analysis results (the Data.Maps) were actually lazy evaluated and a lot of chunks were building up from that. We solved it using the DeepSeq technique, where a term is evaluated to its normal form. Thankfully the necessary "DeepSeq" instances for Map were provided by the new containers package (containers-0.4.2.0).
- Parse Failing: The error analysis has shown us that 4223 modules out of 33286 modules (accounts to 13% of hackage) were failing to parse. We are not sure why exactly this happens, but we think it is related to the fixities of HSE.

One last thing to mention is that although in an earlier implementation we made use of Data.Binary to serialize our analysis results, we switched to dumping the show representation of Maps to plain text files and reading them afterwards. The reason for this choice was that the Binary instances for Data.Map were surprisingly removed from the recent update of the containers package and we want the updated containers so as to support DeepSeq for Maps.

Conclusion

During our project we found answers to the three questions of the introduction. Not only did we do an automatic analysis, but also a manual analysis. Just collecting numbers is not enough to get an idea of why GP is used. Of course to some extend we could determine some useful information from the automatic analysis as can be seen in the results.

Overall we think GP is not used as much as we expected. From around 3000 libraries only 136 libraries use GP. The reason for this we think is the lack of knowledge and experience with generic programming. To use GP in a library is not that difficult in most cases. SYB and Uniplate are easy to use and could help a programmer to remove or avoid writing a lot of boilerplate code.

Not every GP-library is designed to use in practice. Some libraries are developed to get familiar with the concept of GP. And we think this is the reason why some GP-libraries are not used a significant amount of times on Hackage. The reason why SYB and Uniplate are the most used libraries is because they are designed to use in practice and also they are not difficult to get familiar with once the concept of GP is clear to a programmer.

From the manual analysis we got an idea of what GP is used for. We can see in the results that in some libraries it is used for transformations on datastructures and in others for queries on datastructures. Sometimes transformations and queries are used together in a library. From the automatic analysis it became clear that transformations are used more often than queries.

Future work

In our analysis we supported the libraries SYB and Uniplate. The reason for this was that those libraries were used more often then the remaining libraries all together. In the future it may be possible that one of those other libraries is going to be used more often. We think it would not take t much time to extend or implementation to support analysis for other GP-libraries. This extensibility is one of the reasons why we choose the modular setup of our code.

There are some cases in which the use of SYB can be replaced with Uniplate. It is hard (but not impossible) to check this automatically. Uniplate can be faster then SYB when you manually define the instances for Uniplate and Biplate. So it would be desirable if this could be determined.

We want to stress that the numbers retrieved from our automatic analysis may not be completely accurate. E.g. it is not guaranteed that if we search for a function x in the AST, that this function x is really the function we search for. It could be an identically named function that refers to another function than the one from one of the GP-libraries. So we think there is space for improvement in this part.

One concrete example is looking at imports of a module. When we constructed our top ten of used GP-functions, the empty function was on position one. But when we looked at the code manually we saw that most of the occurrences of empty were the Data.Map.empty or the Data.Set.empty. By looking at the imports of a module, in most cases you can resolve this issue.