Usage of Generic Programming on Hackage
— Experience report —

Nikolaos Bezirgiannis

Johan Jeuring

Sean Leather

Technical Report UU-CS-2013-014
July 2013

Department of Information and Computing Sciences
Utrecht University, Utrecht, The Netherlands
www.cs.uu.nl

ISSN: 0924-3275

Department of Information and Computing Sciences
Utrecht University
P.O. Box 80.089
3508 TB Utrecht
The Netherlands

Usage of Generic Programming on Hackage — Experience report —

Nikolaos Bezirgiannis[1] Johan Jeuring[1,2] Sean Leather[1]

[1]Department of Information and Computing Sciences, Utrecht University, P.O. Box 80.089, 3508 TB Utrecht, The Netherlands
[2]School of Computer Science, Open University The Netherlands
n.bezirgiannis@students.uu.nl {j.t.jeuring,s.p.leather}@uu.nl

Abstract

Generic programming language constructs, tools and libraries have been available in Haskell since the first report on the programming language Haskell. At the beginning of the 1990s generic programming techniques could be used via the **deriving** construct, and since then numerous generic programming libraries and tools have been developed. At the time of writing, the categories 'generic' and 'generics' on Hackage, the online repository of Haskell software, contain 53 packages. Although not all of these are generic programming libraries or tools, there are many approaches to generic programming to choose from. This brief paper discusses an analysis of the usage of generic programming language constructs, tools, and libraries. We analyse how often which language constructs, tools, and libraries are used on Hackage, how often class instances are derived generically or written manually, and for some libraries, how often the functions that appear in these libraries are used.

Categories and Subject Descriptors D.1.1 [*Programming Techniques*]: Functional Programming

General Terms Languages, Measurement, Survey

Keywords Generic Programming, Haskell, Deriving Mechanism, Hackage

1. Introduction

Datatype-generic programming, often abbreviated to generic programming, is defining functions that depend on the structure, or shape, of datatypes. Generic programming language constructs, tools and libraries have been available in Haskell since the first published report on the programming language Haskell [7] via the **deriving** construct. In the early phase of generic programming, generic programming was available via extra language constructs [5, 8, 14], or preprocessing tools [17, 20, 24]. Later, generic programming libraries [1, 3, 12, 13, 18, 23] were introduced, in many variants.

The different approaches to generic programming have been previously compared [6, 22]. Applications of generic programming have been given in quite a few papers [4, 10, 15]. However, none of these papers contains an empirical study of the actual usage of the various approaches. Information about usage of generic programming techniques might be useful to determine the interest in the different libraries, to focus updates of functionality of libraries, to test backwards compatibility of libraries, and to find out where applications of generic programming may have been missed, for example because a library contains a hand-written instance of a particular class such as $Functor$, where using a derived instance would have been preferable.

Hackage is the de-facto online repository of open-source Haskell programs and libraries. At the time we performed our latest analysis, on Sunday, June 2, 12:06:57 UTC, 2013 it contained 5207 packages. To analyse the usage of generic programming techniques, we analyse the packages in the Hackage database. Hackage is by far the most important platform to distribute Haskell software, and an analysis of code on Hackage gives a good idea about how programming techniques are used in practice.

In this paper we analyse the complete Hackage repository to gather statistics about:

- how much the original **deriving** language construct is used. The **deriving** construct has been around since the start of Haskell, and we expect it to be used frequently.
- how much the DrIFT [24] and *derive* [17] preprocessing tools are used.
- the number of packages that depend on some generic programming tool or library to compile their sources. The outcome indicates how much a generic programming tool or library is used.
- which functions from the Scrap Your Boilerplate (SYB) [13] and Uniplate [18] libraries are used in packages that import these libraries.

The rest of this paper is organised as follows. In the next section we introduce the characteristics of the deriving, derive, SYB, and uniplate tools and libraries, which we analyse in more detail later. This section partially reveals the methodology of the analysis, which is described more extensively in Section 3. This section describes the design and implementation of the analysis package. Section 4 presents the results of analysing Hackage using our package for the two deriving techniques (**deriving** and $derive$), and for the usage of the generic programming libraries. Section 5 discusses related work, and Section 6 discusses potential future research and concludes.

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for components of this work owned by others than the author(s) must be honored. Abstracting with credit is permitted. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee. Request permissions from permissions@acm.org.

WGP '13, September 28, 2013, Boston, MA, USA.
Copyright is held by the owner/author(s). Publication rights licensed to ACM.
ACM 978-1-4503-2389-5/13/09. . . $15.00.
http://dx.doi.org/10.1145/2502488.2502494

2. Characteristics of generic programming approaches

This section discusses some characteristic aspects of generic programming approaches, in particular the aspects that are relevant for performing our usage analysis. We will not attempt to give a complete overview of generic programming language constructs, tools, and libraries, which can be found elsewhere [6, 9].

2.1 deriving

The **deriving** construct has been part of Haskell since its start. The Haskell 98 language report [21] states that instances for the classes *Eq*, *Ord*, *Enum*, *Bounded*, *Show*, and *Read* can be derived automatically. **deriving** can be used on both **data** and **newtype** declarations, as in the examples below:

$$\textbf{data} \quad DT_1\ a = DT_1\ a\ \textbf{deriving}\ (Show)$$
$$\textbf{newtype}\ DT_2\ a = DT_2\ \{getDT_2 :: a\}\ \textbf{deriving}\ (Eq)$$

In the latest versions of GHC also instances of *Data*, *Typeable*, *Functor*, *Foldable*, *Traversable*, and, more recently, *Generic* classes can be derived automatically. The **deriving** construct can also be used on datatype-like GADTs, i.e. GADTs that do not make use of GADT-specific concepts. Since the implementation of the **deriving** construct looks at the structure of datatypes, it involves a form of generic programming.

A programmer can choose to construct a hand-written class instance if the standard behavior of a derived class-method is not desired. For example, we might want an equality on lists that only compares top-level constructors, as in

$$\begin{array}{l}\textbf{instance}\ Eq\ [a]\ \textbf{where}\\ \quad [\,] \equiv [\,] = True\\ \quad (x : xs) \equiv (y : ys) = True\\ \quad \_ \equiv \_ = False\end{array}$$

GHC supports an alternative way to derive a class instance, using the *StandaloneDeriving* extension. Using *StandaloneDeriving*, a programmer can derive the instance of a class for a datatype at another place in the program than the datatype itself, for example:

$$\textbf{deriving instance}\ (Read\ a) \Rightarrow Read\ (DT_1\ a)$$

StandaloneDeriving is useful if we want to use a different class context than the one derived by **deriving**, or if we want to derive an instance for a datatype in another module, which we cannot or do not want to adapt.

Another extension of GHC related to the **deriving** construct, is *GeneralizedNewtypeDeriving*. When enabled, a **newtype** declaration inherits some or all instances of the contained datatype, with the exception of *Show*, *Read*, *Data* and *Typeable* instances, which follow the normal deriving scheme.

$$\begin{array}{l}\textbf{newtype}\ DT_3\ a = DT_3\ \{getDT_3 :: Maybe\ a\}\\ \qquad\qquad \textbf{deriving}\ (Monad)\end{array}$$

GeneralizedNewtypeDeriving involves no generic programming, it only steers inheriting instances.

2.2 DrIFT

The DrIFT tool is used to automatically derive functions or class instances for Haskell data types. It is a preprocessor to Haskell 98 source files. A user annotates code with comments such as:

$$\begin{array}{l}\textbf{data}\ DT_4\ a = DT_4\ a\ \textbf{deriving}\ Show\\ \qquad\qquad \{\text{-! derive } Ord, Read\ \text{! -}\}\\ \{\text{-! for } DT_4 \text{ derive : } Eq, NFData\ \text{! -}\}\end{array}$$

for inline and standalone deriving, respectively. DrIFT supports the derivation of instances of a number of standard classes. Although not impossible, adding rules specifying how other class instances or functions are derived is hard. DrIFT has an extra "global" directive:

$$\{\text{-! global : } Enum\ \text{! -}\}$$

which is used to derive instances for every datatype defined in the module.

2.3 *derive*

The *derive* library and preprocessing tool [17] is a relatively recent successor to DrIFT. Using *derive* is very similar to using DrIFT:

$$\begin{array}{l}\textbf{data}\ DT_5\ a = DT_5\ a\ \textbf{deriving}\ (Show\ \ \{\text{-! } Read\ \text{! -}\}\ )\\ \ \{\text{-! deriving instance } Eq\ DT_5\ \text{! -}\}\end{array}$$

Using *derive* it is easier to extend the set of derivable type classes and functions using custom derivations.

When using *derive* as a library instead of a preprocessing tool, we can leverage Template Haskell and define derivations as follows:

$$\begin{array}{l}\textbf{data}\ DT_6\ a = DT_6\ a\\ \$\ (derive\ [makeEq, makeBinary]\ ''\ DT_6)\end{array}$$

The drawback of this approach is that it is less portable, since only GHC offers support for Template Haskell.

2.4 SYB

SYB [13] is a generic programming library that has been around for a decade now. To use SYBs transformations and queries on a particular datatype, the datatype needs to have instances of the classes *Data* and *Typeable*.

For example, if we want to prefix all variables with a string in an abstract syntax tree of type AST

$$\begin{array}{l}\textbf{data}\ AST = Var\ String\\ \qquad\quad | \ Lam\ Var\ AST\\ \qquad\quad | \ App\ AST\ AST\end{array}$$

by means of the function $prefixV$ defined by

$$\begin{array}{l}prefixV :: String \to AST \to AST\\ prefixV\ s\ (Var\ v) = Var\ (s +\!\!+ v)\\ prefixV\ s\ x = x\end{array}$$

we use the SYB functions $everywhere$ and mkT to define a generic transformation:

$$\begin{array}{l}prefix :: String \to AST \to AST\\ prefix\ s = everywhere\ (mkT\ (prefixV\ s))\end{array}$$

If we want to count the number of variables that occur in an AST-value using the function $nrOfVars$, which returns 1 for a variable and 0 for anything else, we use the functions $everything$ and mkQ to obtain a generic query:

$$\begin{array}{l}vars :: AST \to Int\\ vars = everything\ (+)\ (0\ `mkQ`\ nrOfVars)\end{array}$$

In our analysis we want to find out how often such functions are used in the modules on Hackage.

2.5 Uniplate

Uniplate [18] is a generic programming library similar to SYB, but simpler, often faster, and less powerful. Here are the definitions of the SYB examples in Uniplate:

$$\begin{array}{l}prefix' :: String \to AST \to AST\\ prefix'\ s = transform\ (prefixV\ s)\\ \\ vars :: AST \to Int\\ vars\ t = sum\ [1 \mid Var\ \_ \leftarrow universe\ t]\end{array}$$

where instead of $everywhere$ we use the Uniplate function *transform*, and instead of $everything$, we use the function $universe$.

3. Methodology

We analyse the Hackage library to find out how much the **deriving** language construct is used, which classes are derived, how much DrIFT and *derive* are used, how often generic programming libraries are used, and finally, how often generic functions from some generic programming libraries are used.

We have developed *gpah*[1] (generic programming analysis of Hackage) to perform the analysis. *gpah* analyses the entire Hackage database in a single run. In the first step of the run, it collects data from the package descriptions (Section 3.1), and in the second step we parse the modules of the packages and determine the properties we are interested in (Section 3.2). Section 3.3 evaluates the methodology we use.

In our analysis we only focus on determining the usage of generic programming language constructs, tools, libraries and functions. However, the same approach could be used to analyse Hackage software for other patterns.

3.1 Analysing packages

Hackage requires that all software uploaded to its database is packaged using the Cabal system [11], an automatic build tool for Haskell libraries and applications. Each package contains a Cabal package description file, containing information about how the package is built, what its dependencies are, and information about authors, license, etc.

gpah parses Cabal files using Cabal's own parser. From the parsed output it obtains the name of the package, the locations of its source modules, any executables that it creates, the options passed to the C-preprocessor (cpp), and its generic programming dependencies.

3.2 Analysing modules

Since a significant portion of the packages on Hackage makes use of preprocessor directives in its code, *gpah* runs the modules obtained from the locations described in the Cabal file through cpp using a custom-made shell script, prior to parsing.

gpah parses the resulting modules using *haskell-src-exts*, a Haskell parser that can handle Haskell syntax and many of its extensions. Apart from the Haskell parser in GHC, *haskell-src-exts* is probably the most complete Haskell parser, which cannot only deal with Haskell itself, but also with GADTs and Template Haskell, for example.

gpah analyses the parsed modules for specific occurrences of syntactic constructs such as **deriving**, multi-line comments to analyse usage of *derive* and DrIFT, and particular generic functions.

3.3 Evaluation

This section evaluates our method and the corresponding software *gpah* for analysing the usage of generic programming on Hackage.

gpah is written in Haskell. Since it needs to traverse values of the rather large abstract syntax tree datatype for Haskell, it makes extensive use of SYB, for example to find occurrences of generic functions in modules, and Uniplate, to remove a lot of boilerplate in the **deriving** analysis. The multi-line comments are further analyzed using a basic parser developed using the *uu-parsinglib* parser combinatory library. Analyzing every comment of the Hackage repository is rather costly, and a full analysis of Hackage takes several hours to complete on modern hardware. Moving the Hackage snapshot archive to a temporary file storage facility such as *tmpfs* on many Unix systems, speeds up the execution of successive runs of the analysis, since the entire codebase of Hackage is cached. Parallelizing the analysis will also help here.

After applying the cpp preprocessing, *gpah* parses 41027 of the 46580 Haskell source modules from Hackage, which corresponds to about 88% of the complete repository. *gpah* cannot parse the remaining 12% (5553 modules) for the following reasons:

- since we do not take in-module cpp options into account, preprocessing failed for 1685 modules (3.6%).
- the *haskell-src-exts* library does not parse all Haskell extensions used by the Hackage packages.
- some Hackage packages contain errors and cannot be parsed.
- some Hackage packages contain source files that are not needed to build the package, but which the developers left in the source tree. Some of these files do not parse. Only by performing import chasing we could detect that we can omit these files from our analysis. However, there are arguments for including these files in the analysis as well, after all, although some of them clearly contain garbage, these are Haskell modules on Hackage.

Instead of parsing using *haskell-src-exts* we could adapt the parser of GHC to collect specific information about the components we are interested in, such as **deriving** declarations or calls to generic functions, as in the analysis of overlapping instances in Hackage from Morris [19]. Instead of running *gpah*, we could then just run cabal-install on all packages on Hackage. This approach might be more complete and appropriate for our purposes, since the GHC parser and type checker are the most elaborate and well-tested parsing and typing utilities. We have decided to not take this approach for the following reasons:

- adapting and instrumenting the parser and output of GHC is considerably more work than using Cabal's parser and the parser from *haskell-src-exts*.
- even when using cabal-install and GHC, we would not be able to parse all Hackage packages, since different packages require different versions of the tools. These problems might be partially resolved if we would adapt all GHC versions from the last couple of years, and would install the different versions of cabal-install and the libraries used by Hackage packages, but we envisage that this would require a serious time investment.
- the information we want to collect for our analysis is mainly based on syntax, and there is no added benefit of checking semantic properties of packages.
- right now a full analysis using *gpah* takes several hours. We expect building all Hackage packages will take days, if not more, to complete.

4. Results

This section presents the results of performing our analysis: running *gpah* on the Hackage repository. Section 4.1 shows the usage of the **deriving** construct. Section 4.2 presents the usage results for the preprocessing tools DrIFT and *derive*. Section 4.3 shows how often the various generic programming libraries are used, and Section 4.4 shows how often some generic functions from the Uniplate and SYB libraries are used. The last section shows some highlights from the differences between the analyses performed in 2012 and 2013.

4.1 Usage analysis of deriving

The parsable modules on Hackage contain 49218 **data** and **newtype** definitions, of which 644 are GADTs, 80 type family instances, and 10 data family instances.

[1] See: `http://hackage.haskell.org/package/gpah`

28707 **data** and **newtype** declarations (58%) contain a **deriving** clause. Using these **deriving** statements, a total of 68149 instances of classes are derived, on average 2.37 class instances per datatype that contains a **deriving** clause.

We exclude the 7497 **newtype** declarations from our results. Due to the nature of **newtype**s, deriving a class instance on a newtype does not require a significant amount of generic programming: **newtype deriving** does use (some) generic programming techniques when deriving instances for *Read*, *Show*, *Typeable*, or *Data*, but not when deriving instances of *Eq*, *Ord*, *Enum* and *Bounded*. Using *GeneralizedNewtypeDeriving* code is inherited from the argument type, and no generic programming techniques are used. We do not include the 1185 standalone **deriving** declarations, because finding out whether a standalone derivation refers to a datatype or a newtype is rather hard, and for the reasons mentioned above, we do not want to include **newtype** deriving in our results. Thus we arrive at 41721 datatype definitions, 25012 of those with a deriving clause, and 55796 classes derived, respectively. Since we possibly excluded cases in which generic programming techniques are used, these numbers are conservative estimates.

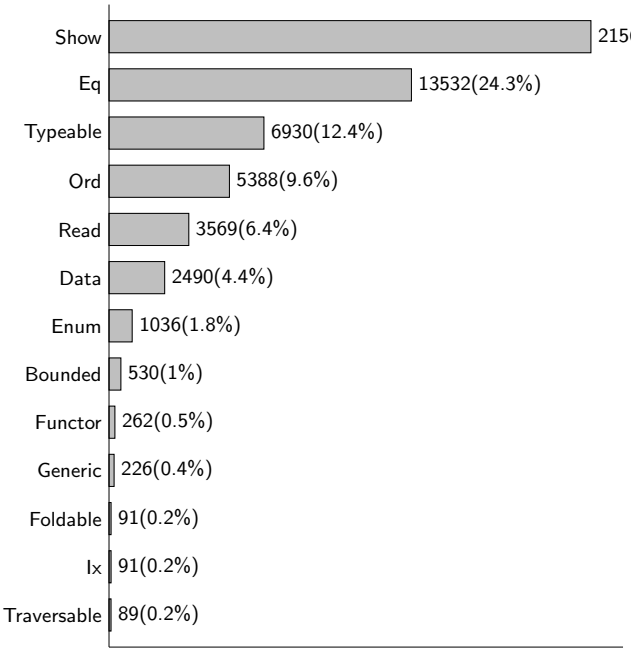

Figure 1. Derived class instances using **deriving**

Figure 1 shows which classes are derived using **deriving** clauses, with in between parentheses the percentage of total number of derived instances. *Show* and *Eq* are by far the most popular derived classes. The GHC-specific derivable classes *Typeable* and *Data* appear above the Haskell language report specified classes *Bounded* and *Enum*. Apparently quite a few Haskell programmers rely on the language extension $-XDeriveDataTypeable$, for example.

The relatively new *Generic* class [16] is derived 226 times. The classes *Data*, *Typeable*, and *Generic*, which are mainly used to define or use generic functions, are derived or hand-written 10192 times, which accounts for 15% of the total hand-written and derived instances. This shows that generic programming is used quite a lot on Hackage.

Perhaps even more interesting is Figure 2, in which we specify the number of hand-written instance declarations of classes that could have been derived automatically, with in parentheses the percentage of the total number of hand-written instances. Again, *Show* tops the list, for understandable reasons. We were surprised by the large number of hand-written instance declarations for *Functor*. An instance of *Functor* usually follows a standard, generic, pattern, and implementing an instance using generic programming techniques guarantees that the required laws hold for the functor instance. We had a look at almost 70 hand-written instances of *Functor*. We expect that more than 90% of the hand-written *Functor* instances we inspected could be automatically derived, but since about half of the modules did not compile anymore with the latest version of GHC, we could not check this. There is a wide variety of reasons for not using automatic derivation of *Functor*:

- some datatypes make use of existential quantification or GADT features.
- the code predates the automatic *Functor* derivation feature of GHC.
- *fmap* is implemented using functionality from *Traversable*.
- the *Functor* instance uses special features such as *INLINE* pragma's or lazy pattern matching.
- the code does not use the standard *Functor* class, but defines another *Functor* class.
- the programmer does not want to depend on the language extension *-XDeriveFunctor*. Indeed, few of the packages that do not automatically derive *Functor* make use of other language extensions.

We also had a brief look at the modules that derive *Data* and *Typeable*, but contain some hand-written instances for *Data* and *Typeable*. Most of the hand-written instances could be explained by the fact that the datatypes on which they are used do not expose the underlying structure of a datatype. For example, a *Vector* may contain several fields with metadata, and a single field containing an *Array* with the vector elements. When folding over this datatype, we want to fold just the array of elements.

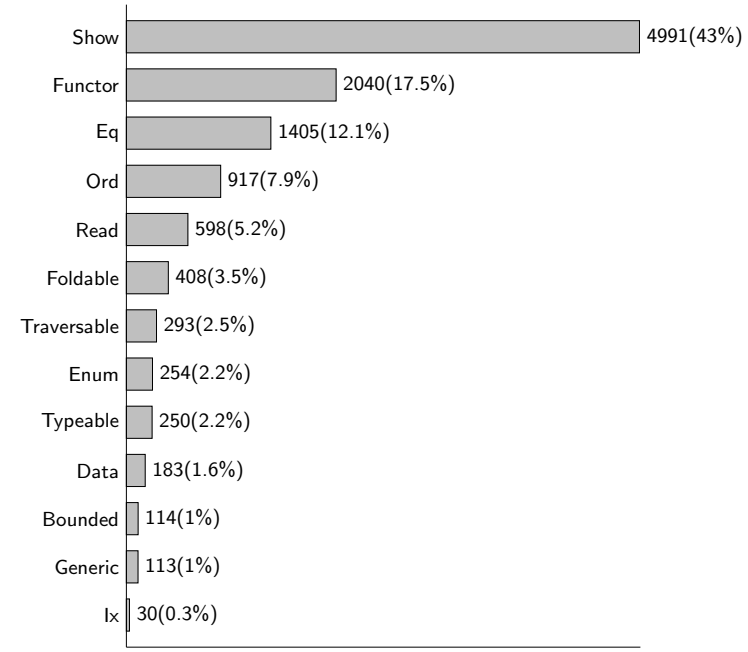

Figure 2. Hand-written class instances of derivable classes

4.2 Usage analysis of preprocessing tools

No library on Hackage specifies in its Cabal file that it depends on DrIFT. However, since DrIFT is an external preprocessing tool using annotations in comments, this does not imply that DrIFT is not used. Indeed, parsing comments to find DrIFT annotations reveals that DrIFT derives an instance of $Binary$ 47 times, and $Monoid$ 9 times. We only found a single occurrence of the global directive of DrIFT, which possibly justifies the absence of global in the more recent but similar *derive* preprocessing tool.

45 Hackage packages depend on *derive*. Figure 3 shows how many class instances are derived using *derive*. *derive* has mainly been used to derive $Binary$.

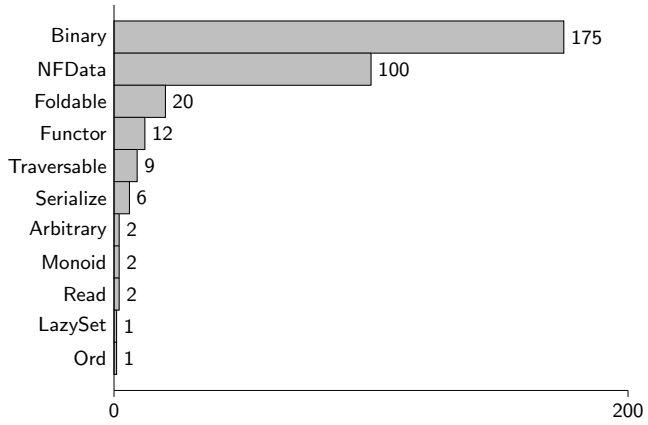

Figure 3. Derived class instances using *derive*

This figure does not show the number of user-defined derivations. We found 25 user-defined derivations, which have been used 233 times on datatypes to derive class instances or instances of generic functions.

Surprisingly, just one package uses the *derive* preprocessing syntax. The other packages use Template Haskell for the derivations. Using *derive* with Template Haskell leads to code that does not rely on external preprocessor tools, which makes compiling easier. If a developer wants to use another compiler than GHC, probably an external preprocessing tool is preferable.

4.3 Usage analysis of generic programming libraries

The Hackage categories "Generic" and "Generics" contain 53 packages. Some of these are not generic programming tools or libraries, and are omitted from our analysis. For the remaining list of libraries and tools, together with *derive*, syb-with-class, and DrIFT-cabalized, which do not appear in these categories, we calculate the reverse dependencies. A reverse dependency of a library is a package that imports the library. The results are shown in Figure 4. We do not include the packages without any reverse dependencies.

In total, there are 313 reverse dependencies, appearing in 278 different packages. 113 (41%) of these create at least one executable. This number may be a better indication of the actual generic programming usage, since in this way, we are excluding reverse dependencies introduced by generic programming libraries built on top of other generic programming libraries.

SYB (44%), *derive*, and Uniplate together account for the majority of the reverse package dependencies. Part of an explanation might be that SYB, together with the possibility to derive $Data$ and $Typeable$, has been directly available in GHC for many years. Some of the other libraries are fairly recent.

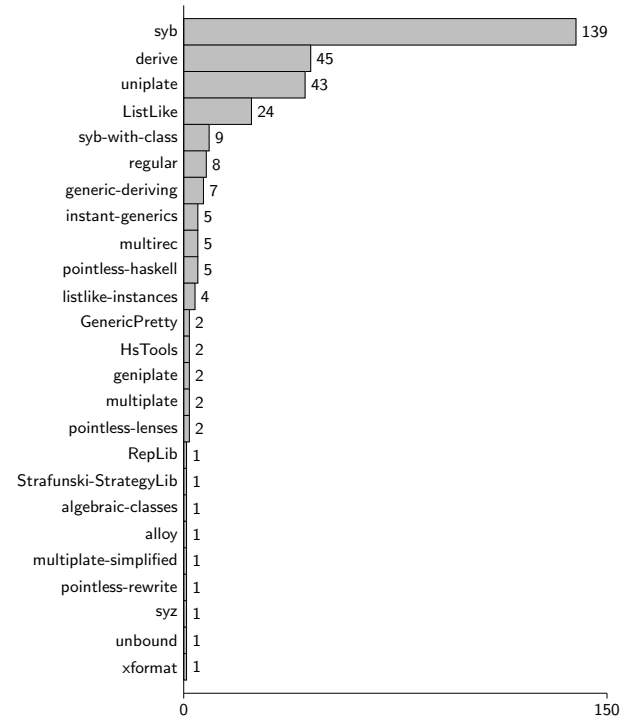

Figure 4. Reverse dependencies of generic programming libraries

4.4 Usage analysis of generic functions

We take a closer look at SYB and Uniplate, two of the three most widely-used generic programming libraries. We collect information about functions exported by SYB and Uniplate. We found 382 occurrences of generic transformations functions such as mkT (SYB, 104), $everywhere$ (SYB, 93), and $transform$ (Uniplate, 82), and 251 occurrences of generic query functions such as $gmapQ$ (SYB, 73), $everything$ (SYB, 59), and $universeBi$ (Uniplate, 58).

Finally, based on the activity log file of Hackage, we present the number of new packages uploaded to Hackage using SYB or Uniplate, and the number of updates of packages on Hackage using SYB or Uniplate in Figure 5. The increasing number of new packages and package updates using SYB and Uniplate shows that either developers are using generic programming techniques more, or that more developers are aware of generic programming techniques.

4.5 What happened last year?

The results described in this section were obtained from an analysis performed on Sunday, June 2, 12:06:57 UTC 2013. We performed the same analysis more than a year ago on Wed May 16 08:44:48 UTC 2012. This section highlights the differences between the situation then and now.

- Instances of the class $Functor$ are derived 262 times up from 151 (74% up).
- The number of derived instances of $Foldable$ and $Traversable$ increases almost 60%.
- The number of derived instances of $Data$, $Typeable$ and $Generic$ goes up from 7379 to 9646. In particular, $Generic$ goes up from 33 to 226 (585%).
- The number of packages depending on at least one generic programming library goes from 217 to 278.

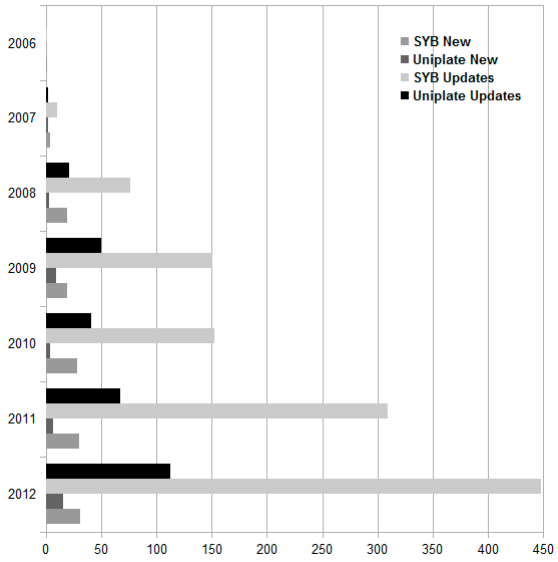

Figure 5. New and updates to libraries using SYB or Uniplate

5. Related Work

To our knowledge, our analysis is the first to study and compare usage of generic programming techniques in Haskell software in a real-world setting on Hackage. Rodriguez et al [22] compare practical aspects of generic programming libraries, but usage is not one of these.

The software on Hackage has been analysed before, for example by Morris [19], influenced by Coutts [2], for the purpose of determining the usage of overlapping instances. As explained in Section 3.3, our methodology differs significantly from the one used by Morris and Coutts.

Information about the reverse dependencies of generic programming libraries on Hackage can also be obtained via the website `http://packdeps.haskellers.com/reverse`.

6. Conclusion & Future Work

We have analysed Hackage with respect to the usage of generic programming language construct, tools, libraries and functions. The results show that generic programming techniques are widely used, in particular Haskell's **deriving** construct. The huge number of derived instances of *Data*, *Typeable* and *Generic* show that users not only derive instances of the standard Haskell classes, but also use other generic programming components. Comparing with the same analysis performed one year ago, the most striking difference is the 585% increase in usage of deriving *Generic*.

By inspecting some of the hand-written instances of the *Functor* class, we found that quite a few of the hand-written instances of *Functor* can be derived automatically. We want to manually examine the hand-written instance declarations that could have been derived automatically for other classes too. This is a substantial amount of work, but might give some insight in why developers do not use the deriving mechanism, give ideas about new generic functions based on the patterns in the hand-written instances, or give ideas to add 'warnings' to a tool like hlint, telling programmers that it might be possible to replace a hand-written instance of a class by a derived instance.

Our methodology for analysing Hackage can also be used to analyse Hackage with respect to other aspects than generic programming.

We would like to repeat this analysis on a yearly basis.

Acknowledgements

We thank the anonymous referees for their comments on a previous version of this paper.

References

[1] J. Cheney and R. Hinze. A lightweight implementation of generics and dynamics. In *Haskell'02*, pages 90–104, 2002.

[2] D. Coutts. Solving the diamond dependency problem. `http://www.well-typed.com/blog/12`, 2008.

[3] R. Hinze. Generics for the masses. In *ICFP'04*, pages 236–243, 2004.

[4] R. Hinze and J. Jeuring. Generic haskell: Applications. In *Generic Programming*, volume 2793 of *LNCS*, pages 57–96, 2003.

[5] R. Hinze and S. Peyton Jones. Derivable type classes. In *Proceedings of the Fourth Haskell Workshop*, 2000.

[6] R. Hinze, J. Jeuring, and A. Löh. Comparing approaches to generic programming in haskell. In *Summer School on Datatype-generic programming*, LNCS, pages 72–149, 2007.

[7] P. Hudak, S. Peyton Jones, P. Wadler, B. Boutel, J. Fairbairn, J. Fasel, M. M. Guzmán, K. Hammond, J. Hughes, T. Johnsson, D. Kieburtz, R. Nikhil, W. Partain, and J. Peterson. Report on the programming language haskell: a non-strict, purely functional language version 1.2. *SIGPLAN Notices*, 27(5):1–164, 1992.

[8] P. Jansson and J. Jeuring. PolyP — a polytypic programming language extension. In *POPL'97*, pages 470–482, 1997.

[9] J. Jeuring, S. Leather, J. P. Magalhães, and A. R. Yakushev. Libraries for generic programming in Haskell. In *Advanced Functional Programming*, volume 5832 of *LNCS*, pages 165–229, 2008.

[10] J. Jeuring, J. P. Magalhães, and B. Heeren. Generic programming for domain reasoners. In *TFP '09*, pages 113–128, 2009.

[11] I. Jones, S. Peyton Jones, S. Marlow, M. Wallace, and R. Patterson. The Haskell Cabal, a common architecture for building applications and libraries, 2005.

[12] O. Kiselyov. Smash your boilerplate without class and Typeable. `http://article.gmane.org/gmane.comp.lang.haskell.general/14086`, 2006.

[13] R. Lämmel and S. Peyton Jones. Scrap your boilerplate: a practical design pattern for generic programming. In *ICFP '03*, 2003.

[14] A. Löh, J. Jeuring, T. van Noort, A. Rodriguez, D. Clarke, R. Hinze, and J. de Wit. The Generic Haskell users guide, Version 1.80 - Emerald release. Technical Report UU-CS-2008-011, Utrecht University, 2008.

[15] J. P. Magalhães and W. B. de Haas. Functional modelling of musical harmony: an experience report. In *ICFP '11*, pages 156–162, 2011.

[16] J. P. Magalhães, A. Dijkstra, J. Jeuring, and A. Löh. A generic deriving mechanism for Haskell. *Haskell '10*, pages 37–48, 2010.

[17] N. Mitchell. Deriving a relationship from a single example. In *AAIP '10*, volume 5812 of *LNCS*, 2010.

[18] N. Mitchell and C. Runciman. Uniform boilerplate and list processing. In *Haskell '07*, pages 49–60, 2007.

[19] J. G. Morris. Experience report: using Hackage to inform language design. In *Haskell '10*, pages 61–66, 2010.

[20] U. Norell and P. Jansson. Prototyping generic programming in template haskell. In *MPC '04*, volume 3125 of *LNCS*, pages 314–333, 2004.

[21] S. Peyton Jones. *Haskell 98 Language and Libraries: The Revised Report*. Cambridge University Press, 2003.

[22] A. Rodriguez, J. Jeuring, P. Jansson, A. Gerdes, O. Kiselyov, and B. C. d. S. Oliveira. Comparing libraries for generic programming in Haskell. In *Haskell '08*, pages 111–122, 2008.

[23] S. Weirich. RepLib: a library for derivable type classes. In *Haskell '06*, pages 1–12, 2006.

[24] N. Winstanley and J. Meacham. *DrIFT user guide*, 2006. `http://repetae.net/~john/computer/haskell/DrIFT/`.