Experience Report: Using Hackage to Inform Language Design

J. Garrett Morris
Portland State University
jgmorris@cs.pdx.edu

Abstract

Hackage, an online repository of Haskell applications and libraries, provides a hub for programmers to both release code to and use code from the larger Haskell community. We suggest that Hackage can also serve as a valuable resource for language designers: by providing a large collection of code written by different programmers and in different styles, it allows language designers to see not just how features could be used theoretically, but how they are (and are not) used in practice.

We were able to make such a use of Hackage during the design of the class system for a new Haskell-like programming language. In this paper, we sketch our language design problem, and how we used Hackage to help answer it. We describe our methodology in some detail, including both ways that it was and was not effective, and summarize our results.

Categories and Subject Descriptors D.3.2 [*Programming Languages*]: Language Classifications—Applicative (functional) languages

General Terms Experimentation, Languages

Keywords Haskell, Hackage

1. Introduction

As part of the High-Assurance Systems Programming[1] project at Portland State University, we are designing Habit, a dialect of Haskell intended to support systems-level programming tasks with a high level of assurance. While Habit diverges from Haskell in several significant ways, such as being strict by default and attempting to infer the pointedness of expressions, it also shares many Haskell features, like the type class system. In deciding on the features of the Habit type class system, we were eager to learn as much as possible from the Haskell community's experience, both with the core class system and with its more experimental aspects.

One such aspect is overlapping instances, a feature of the Haskell class system implemented by GHC [12] and Hugs [3]. Notwithstanding the long history of overlapping instances (Gofer, for example, first implemented overlapping instances in version 2.28, released in February 1993), there is little consensus within the Haskell community about whether, or how, they should be supported or standardized. Indeed, while some recent work depends on overlapping instances (such as Swierstra's solution to the expression problem [10]), recent extensions to the class system [6, 8] exclude overlap. This led to several questions: should Habit support overlapping instances? If not, what kinds of programs would Habit users be prevented from writing? Are there viable alternatives to the use of overlapping instances?

To help answer these questions, we surveyed the frequency and uses of overlapping instances in Hackage[2], an online repository of Haskell libraries and applications. Our survey is distinguished from the folklore and informal input that inform any language design both by being based on a large code library and by having an infrastructure to automate data collection. As much as possible, we reused the Hackage infrastructure to simplify the mechanics of the survey. In particular, we used and extended GHC and `cabal-install` [4], a tool to download and install packages (and their dependencies) automatically from Hackage. We hoped to answer the following questions:

- What proportion of the total code on Hackage uses overlapping instances?
- In code that uses overlapping instances, how many instances overlap each other?
- Are there common patterns among the uses of overlapping instances?

In turn, the answers to these questions would inform the design of the Habit class system: whether to support overlapping instances completely, not at all, or to attempt to find a new approach that supported the uses of overlapping instances without introducing their complexity.

This paper proceeds as follows: The remainder of Section 1 provides background information, including an overview of type classes, overlapping instances, and the Hackage infrastructure. As Hackage is still under active development, some aspects of Hackage will have changed since we conducted our survey in April 2009. This section attempts to describe Hackage as it was then, not as it is today; however, we will attempt to indicate those features that we know have changed in the meantime. Section 2 describes the methodology of our survey: how we modified GHC and `cabal-install` for our purposes, and how we used the modified tools. We believe that Hackage surveys can provide valuable data for other Haskell-related language design projects; therefore, as much as possible, we highlight strengths and document weaknesses in our methodology, both those that affected our survey directly and those that might be relevant for similar projects. Section 3 summarizes the results of our survey, and includes some observations on Hackage metadata. Finally, Section 4 discusses related and future work and concludes.

[1] `http://hasp.cs.pdx.edu`

[2] `http://hackage.haskell.org`

Haskell'10, September 30, 2010, Baltimore, Maryland, USA.

1.1 Background: Overlapping Instances

This section provides a summary of the overlapping instances extension; more detailed discussion is available elsewhere [5, 7].

Type classes [14] describe relations on types and provide a general way to introduce and type overloaded functions. For example, the `Show` class includes types whose values have simple textual representations. A basic version of the `Show` class might be defined as follows:

```
class Show t where
   show :: t → String
```

Most primitive types, such as `Int` and `Char`, would naturally belong to the `Show` class. Moreover, if we can show the elements of a list, then we can show the list itself by using the Haskell convention of surrounding it with brackets and separating its elements by commas. We can write an instance of `Show` that implements this pattern, using the `intercalate` function from the `Data.List` library:

```
instance Show t ⇒ Show [t] where
    show xs = ['['] ++
              intercalate "," (map show xs) ++
              [']']
```

Unfortunately, this instance will produce unidiomatic output for strings—because strings are lists of characters, the output of an expression like `show "abc"` would not be the string constant `"abc"` but instead the list constant `['a', 'b', 'c']`. We could write an instance that would generate more idiomatic output for this case:

```
instance Show [Char] where
    show cs = ['"'] ++ cs ++ ['"']
```

However, a program that contained both the instances for `Show [t]` and `Show [Char]` would not be valid Haskell because the compiler could potentially resolve (i.e., choose an instance that implements) the predicate `Show [Char]` with either instance. As such, these instances would be considered overlapping.

We can formalize the notion of overlapping instances using substitutions. Given two instances:

```
instance P1 ⇒ C t1
instance P2 ⇒ C t2
```

These instances overlap if their conclusions unify; that is, if there are some substitutions S and T such that $S\ \mathtt{t}_1 = T\ \mathtt{t}_2$. The overlapping instances extension [7] provides a means to disambiguate some sets of overlapping instances automatically by introducing a notion of specificity among instances. Given the same examples, the first instance is more specific than the second if there is a substitution S such that $\mathtt{t}_1 = S\ \mathtt{t}_2$, but no substitution T such that $T\ \mathtt{t}_1 = \mathtt{t}_2$. When resolving a predicate, the compiler chooses the most specific applicable instance. This extension allows the two instances of `Show` given earlier, as the `Show [Char]` instance is more specific than the `Show [t]` instance. However, given two instances such as:

```
instance C (a, [b])
instance C ([a], b)
```

it does not provide a way for the compiler to resolve the predicate `C ([a], [b])` because neither instance is more specific than the other.

The overlapping instances extension is implemented differently by different compilers. For example, GHC checks that the instances that apply to a predicate can be ordered by specificity when it attempts to resolve the predicate. As a consequence, it would accept a program containing the two instances for `C` above, but would subsequently reject any attempt to resolve a predicate of the form `C ([a], [b])`. In contrast, Hugs insists that any overlapping instances must be orderable; as a result, it would reject any program containing the two instances for `C`, regardless of the remainder of the program.

1.2 Background: Hackage

Hackage is a large, online repository of Haskell libraries and applications. It organizes Haskell code into packages, each of which consists of a collection of source files along with a metadata file called a `.cabal` file. Each `.cabal` file contains: the name and version of the package; the names and version ranges of the package's dependencies; the preferred optimization and profiling settings; the language extensions used within the package; and, optionally, other compiler flags specified directly. The build and dependency information can, in turn, vary depending on the local configuration and available libraries. The `.cabal` file options include ways to activate a number of standard Haskell preprocessors; however, unlike Makefiles they cannot invoke arbitrary additional tools or further modify the build process.

In addition to the online repository of packages, there are several other tools in the Hackage infrastructure. Among those relevant to this work are Cabal (the Common Architecture for Building Applications and Libraries), which defines a library for building packages based on their `.cabal` files, and `cabal-install`, a tool for automatically downloading and installing packages and their dependencies.

While Cabal supports several Haskell compilers, including GHC, Hugs, NHC and JHC, the majority of the language extensions that Cabal recognizes are only supported by GHC. Therefore, we used GHC for our survey and will restrict our attention to it for the remainder of the paper.

2. Methodology

Our goal was to collect usage information on overlapping instances for as many of the packages on Hackage as possible. We hoped this would give us both an idea of how frequently Haskell programmers used overlapping instances, and a catalog of how they are used. In turn, these results would drive the design of the Habit class system.

We divided the survey into two stages: first, to find which packages use overlapping instances; and second, to identify the overlapping instances within each of those packages. While it would be possible to examine source code for overlapping instances by hand, this process would be vulnerable to human error and would become impractical for larger numbers of packages. Instead, we instrumented GHC to detect overlapping instances and to output information about the location of each such instance as it was encountered. We then attempted to build as many packages from Hackage as possible and collected the output of our instrumentation. Sections 2.1 through 2.3 describe this process in more detail; Section 2.4 considers the alternative of using the `.cabal` metadata to determine which packages to search for overlapping instances; Section 2.5 evaluates our methodology.

2.1 Determining package sets

The Hackage infrastructure requires that any set of packages that it installs includes at most one version of each package [1]; unfortunately, because different packages on Hackage have conflicting requirements, this means that installing all of Hackage at once is not possible. Therefore, our first task was to determine the largest set of packages to check for overlapping instances.

To find such a set, we were inspired by Duncan Coutts' description of using Hackage for regression testing [2]. First, we used `cabal-install` to generate a list of all available packages. We then attempted a dry run of installing those packages. Predictably,

`cabal-install` detected conflicting version requirements. At this point, our approach differed slightly from that described by Coutts. Rather than attempting to restrict the selection of packages to get a close to optimal choice, we moved conflicting packages to a separate package list. As a consequence, we had a number of package sets, each internally consistent but inconsistent with all of the other sets.

This approach was moderately effective. Our initial package list included 1195 packages. From this, we constructed five package lists: the first contained 992 packages, and the remaining four included 139 more. This left 64 packages (5% of the total) that we made no attempt to install, because:

- They required C libraries or version of GHC not available on our survey machine; or,
- They had internally inconsistent dependency requirements; or,
- They depended on a package we were not attempting to install.

While our approach is simple to describe, filtering incompatible packages out of packages lists can be time consuming. In particular, if a given package is incompatible with a list, not only that package but all packages dependent on it must be removed from the list. To assist with this operation, we developed rudimentary support for tracing reverse dependencies through the Hackage database. Similar functionality is now independently available online [13].

2.2 Instrumenting GHC

Our next task was to instrument the compiler to generate output about overlapping instances. By doing so, we avoided time-consuming and error-prone manual inspection of Haskell source code.

As described in Section 1.1, GHC orders instances by specificity when attempting to resolve a predicate and emits an error if the applicable instances cannot be so ordered. However, predicate resolution is an inappropriate place to add our instrumentation: the same set of overlapping instances might be detected numerous times, while other sets of overlapping instances might never be detected because no predicate required their use. We were able to find a suitable alternative place for our instrumentation. When validating instances, GHC checks that each new instance is not an exact duplicate of an instance it has already encountered. In the process, GHC also computes all the instances that unify with the new instance. This is precisely the list of overlapping instances, so we added code to the duplicate instance check to output that list.

This check detects overlaps that are otherwise irrelevant to the compilation process. For example, consider the following overlapping instances (originally presented in Section 1.1):

```
instance C (a, [b])
instance C ([a], b)
```

Our overlap detection would output this set of instances. On the other hand, GHC will not check that it can order these instances until it attempts to resolve a predicate of the form `C ([a], [b])`. In fact, as long as a program does not require GHC to resolve a predicate of that form, it would not even need to enable overlapping instance support. On the other hand, as one of the options we were considering for Habit was a strict limitations on overlap more akin to that implemented by Hugs, we were still interested in detecting this kind of unused overlap.

2.3 Collecting Results

Having identified consistent sets of packages and constructed an instrumented compiler, we were ready to generate our survey data. Following the technique described by Coutts, we compiled each set of packages independently. While we cannot avoid installing packages—a package can only be built if all of its dependencies are installed—we were able to use `cabal-install`'s existing functionality to ensure that each package set was installed to a distinct location and used a distinct local package database. As a result, the packages installed in one package set were not visible when building any other package set, and all the sets could be built without conflict.

Unlike Coutts' regression tests, we were interested in more information than whether each package compiled successfully; we also needed the overlapping instance information emitted during compilation. This meant that we had to extract the survey results from the build logs of each package by hand, instead of being able to use the build reports that `cabal-install` generates automatically. Luckily, our output strings were easily identified by regular expressions, so collecting the overlapping instances from the different package sets was relatively easy.

Alternatively, in the process of instrumenting GHC it would have been possible to output the information that we collected to particular files, possibly specified by a command line option; this would have eliminated the need for the regular expression pass over the build output. We did not take this step in performing our survey, as the output of our instrumentation was easy to detect and our changes to GHC were otherwise quite local.

2.4 Alternative: Using Package Metadata

The mechanism described in the previous sections may seem overly elaborate, especially given that support for overlapping instances must be enabled by specific compiler flags. As compiler flags are listed in `.cabal` files, it would seem that most packages that used overlapping instances could be detected by searching the `.cabal` files for the relevant compiler options or language extensions [9], and much of the previous work—particularly that involved in compiling large portions of Hackage—could have been avoided. There were several technical reasons that convinced us to take our more labor-intensive approach:

- While `.cabal` files are one place that language extensions may be specified, they are not the only place. Individual source files may also specify language extensions and compiler flags in compiler pragmas. Additionally, there are multiple ways that users can enable GHC's support for overlapping instances, including the `OverlappingInstances` language option, the `-XOverlappingInstances` compiler flag, or the older `-fallow-overlapping-instances` compiler flag.
- The presence of overlapping instance support only enables the definition of overlapping instances; it does not require them. This means that packages that declare overlapping instance support may not actually contain any overlapping instances.
- Most significantly, GHC only requires that overlapping instance support be enabled in the module that defines the less specific (overlapped) instances. For example, consider the example instances for `Show` from Section 1.1:

  ```
  instance Show t ⇒ Show [t] where ...
  instance Show [Char] where ...
  ```

  If these instances were in separate modules (perhaps even in separate packages), then only the module that contained the `Show [t]` instance would need overlapping instance support enabled. As a consequence, while examining those modules that had overlapping instance support would allow us to detect all instances that could potentially be overlapped, it would not indicate whether, or how often, any of those instances were actually overlapped.

We will return to this idea in Section 3.2, where we will see if the packages detected with our methodology match up to those that would have been selected based on their metadata files.

2.5 Evaluation

In this section, we consider the effectiveness of our methodology.

One advantage of our approach is that it required relatively little new code. While we had to modify the GHC type checker to emit details about overlapping instances, we were able to make use of the existing structure of the duplicate instance check. In total, we added 10 lines to GHC, not including comments. The changes to `cabal-install` to generate reverse dependences were larger—around 140 lines—but were localized to the implementation of a single additional command.

We were also able to achieve decent coverage of Hackage. We attempted to compile 1131 (95%) of 1195 packages, without making any attempt to repair broken dependencies manually or to install packages that either depended on absent C libraries or required non-Cabal build processes. Unfortunately, of these 1131 packages, only 826 packages (73%) built and installed successfully. The primary cause of build failures was our choice of which compiler to instrument. At the time that we performed the survey, the latest released version of GHC was 6.10.2, while the version in development was 6.11.20090330. One significant change from GHC 6.10 to 6.11 was that GHC's build system had been retooled and simplified. After several unexpected build failures using the 6.10 build tools, we decided to use 6.11 for the survey. While this resolved our build issues, it also had negative consequences. In addition to the compiler itself, GHC provides several packages, including the `base` package that includes the Haskell prelude as well as numerous primitive operations and basic combinators. GHC 6.11 included both versions 3 and 4 of the `base` library, whereas GHC 6.10 had provided only version 3. As `base` version 4 had not yet been released, some packages did not support the changes that it made, but still had dependencies on `base` without upper bounds. Cabal attempted to build these packages using `base` version 4, which failed during compilation.

We believe that these deficiencies would be significantly reduced if the survey were redone now. The current version of GHC, GHC 6.12.2, is based on the version of GHC that we used to perform the survey; as a result, the survey could be done using a released version of GHC instead of a development version. The incompatibilities with versions of the `base` library are also reduced by new features of Cabal and `cabal-install` [11].

A final note is that our methodology seems to be most suited to asking positive questions, such as "how often are overlapping instances used?" or "how many packages use GADTs?" because it is possible to identify places where those expansions are implemented within the compiler and perform local instrumentation at that point. It seems harder to adapt our approach to questions such as "how many packages only use language features in Haskell 98", as answering that question would require establishing that a (large) set of extensions are all not used. Instead of instrumenting a single point in the compiler, it would be necessary to check each extension of Haskell 98 and report whether none of them are used, which would likely require non-local code changes and data collection.

3. Results

We summarize the more interesting results of our survey in two veins: first, our conclusions about the prevalence and usage patterns of overlapping instances; and second, some speculation about the usage of package-level flags and language annotations.

Set size	Frequency
1	1
2	76
3	20
4	11
5	1
6	4
7	2
8	4
9	1
10	1
22	1
72	1

Table 1. The observed sizes of overlapping instance sets and the frequency with which each size appeared

3.1 Overlapping Instances

Of the 826 packages built during our survey, 57 (7%) used at least one overlapping instance. While this may seem like a relatively small proportion of the total code base, we think this level of usage is not insignificant, as overlapping instances are an experimental and somewhat arcane feature of the Haskell type system.

In the packages that used overlapping instances, we found a total of 445 instances overlapping or overlapped by other instances. We partitioned these instances into sets, where each instance in a set overlaps at least one other instance in the set, and no instances outside the set. The 445 overlapping instances partition into 123 sets. (Intuitively, imagine a graph with a vertex for each instance, and an edge between two vertices if their corresponding instances overlap. Our overlapping sets correspond to connected components in the graph.) We can draw some additional conclusions about the use of overlapping instances by examining the sets.

Out of the 123 sets, 19 included overlapping instances from different modules, and 6 (of those 19) included overlapping instances from different packages. 104 (85%) of the sets only included instances from a single module. This suggests that, while applications exist for instances overlapping across modules, much of the use of overlapping instances is quite local.

We also analyzed the size (number of instances) of each set; the results are presented in Table 1. On average, each set had 3.6 instances. However 76 (62%) of the sets had only two instances. The average is pulled up by several outliers: for example, one set of overlapping instances contains 72 instances. This resulted from the definition of a new `Show` instance:

```
instance JSON a ⇒ Show a where ...
```

that overlapped all other instances of the `Show` class. (One could argue further that this instance is an abuse of the `Show` class, as its output is in JSON format instead of the Haskell syntax that most `Show` instances use.) As a final note, there is one set of overlapping instances that claims to contain only one instance; this resulted from an oddity in the data set in which two different modules defined exactly the same instance. The program containing these modules was rejected by the compiler as a result; however, as our data was generated simultaneously to compilation, we still detected the identical instances.

Our data suggests that while some uses require the full generality of overlapping instances, a greater proportion of uses contain a small number of locally-defined instances. To further refine this idea, we performed a manual examination of the extracted instances. We discovered two particularly common usage patterns:

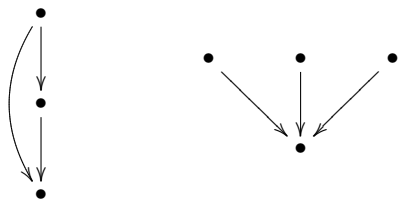

Figure 1. Usage patterns for overlapping instances: On the left, a three-instance chain of alternatives; On the right, a default instance with three more specific implementations.

Alternation. These instances express (usually simple) alternation by making later alternatives more general than earlier ones. This pattern is fragile to encode using overlapping instances: the intention of the programmer is (somewhat) obscured, the method does not easily scale to more than two or three alternatives, and users can potentially add additional alternatives unintended by the original programmer. Instances implementing alternation tend to be local to a single module, or at most a single package. Many examples of this style can be found in the `HList` package; for instance, the `hOccursMany` function returns all the elements of an `HList` with a particular type. It is implemented by the following three instances (all within a single module):

```
instance HOccursMany e HNil where
  hOccursMany HNil = []

instance ( HOccursMany e l, HList l ) ⇒
  HOccursMany e (HCons e l) where
  hOccursMany (HCons e l) = e:hOccursMany l

instance ( HOccursMany e l, HList l ) ⇒
  HOccursMany e (HCons e' l) where
  hOccursMany (HCons _ l) = hOccursMany l
```

We do not imagine that a user would have reason to add additional instances of the `HOccursMany` class.

Default implementations. These instances provide a default implementation for some complex behavior, based on other pre-existing classes. This pattern is roughly similar to one of the functionalities of base classes in object-oriented hierarchies. These instances can be spread across multiple modules or packages. We found these examples particularly common in serialization and generic programming libraries; for example, the `hsx` package includes an instance declaration:

```
instance (XMLGen m,  XML m ∼ x) ⇒
  EmbedAsChild m x where
  asChild = return ∘ return ∘ xmlToChild
```

This provides one way for the `EmbedAsChild` class to be populated, but is far from the only way. Several other packages, such as the `HJScript` package, add their own instances to the `EmbedAsChild` class.

Earlier, we suggested that sets of overlapping instances can be viewed as graphs, with vertices for each instance and edges for each overlap. We could extend this intuition to take account of specifity by directed edges from the more specific to the less specific instances. This would allow us to describe the usage patterns graphically, as in Figure 1.

Unfortunately, we did not collect enough information to automate classifying instances into the usage patterns easily. For each overlapping instance, our survey emits the list of unifying instances, because this is already computed by GHC. However, it would have been more useful to compute and emit specificity information with each overlapping instance. This would have allowed some automatic discovery of patterns.

Even after manual examination, it is not always apparent whether an overlapping instance set belongs to either of the above patterns. For example, the following two overlapping instances appears in the `mmtl` package:

```
instance MonadState s (State s) where ...

instance (MonadTrans t, Monad (t (State s)))
         ⇒ MonadState s (t (State s)) where ...
```

There are two ways we could interpret these instances:

- Any state monad should include the `State` type at some point. This pair of instances provides a complete implementation of the `MonadState` class.
- The `State` type provides one way implement state monads, but there are many others. This pair of instances is not the complete implementation of the `MonadState` class.

It is not clear from the data which of these alternatives is preferred. While we found no implementations of the `MonadState` class outside the `mmtl` package, which supports the first interpretation, it does not seems as clear to us as the `hOccursMany` example above.

One approach we could use to resolve questions like the usage of `MonadState` would be to take the intended use of the package into account. If the package defines an application, or defines a library with many users on Hackage, then we can be relatively certain of the conclusions drawn from the overlaps we detected. However, for libraries without many users on Hackage, the conclusions of our survey would still be uncertain.

3.2 Flags and Annotations

Having completed the survey, we returned to the question raised in Section 2.4 about whether using the Cabal metadata would be a suitable substitute for building all of Hackage. Surprisingly, we found that only 13 of the 57 packages that contained overlapping instances declared the corresponding language extension or GHC flag in their Cabal metadata. However, 59 packages that did not actually contain any overlapping instances included the overlapping instances flag in their metadata. We can imagine several reasons for this:

- Packages may use overlapping instances to provide default implementations for new classes without providing any more specific implementations. In this case, the package author would need to enable overlapping instance support, but our method would only find overlapping instances if there were more specific implementations elsewhere on Hackage.
- Package authors may use standard `.cabal` file templates, or may not remove options from `.cabal` files when they are no longer applicable.
- Package authors may prefer to use source level language pragmas when particular features or options are only needed in a portion of an entire package.

4. Conclusion

In the introduction, we posed three alternatives for the design of the Habit class system:

1. Support overlapping instances as they exist in implementations of Haskell;
2. Do not support overlapping instances at all; or,

3. Define an alternative class system feature that supports many of the uses of overlapping instances without introducing as much complexity.

Our survey suggested that there were a significant number of uses of overlapping instances, including several valuable type-class programming paradigms. This rules out Option 2. However, it also suggested that many uses of overlapping instances did not require the full power of the extension implemented by Haskell compilers, leading us to investigation of Option 3. Our consideration of the alternative pattern led to the creation of instance chains, a new feature of the Habit class system described at length elsewhere [5]. Our examination of the default instance pattern is less advanced; while we have alternative coding patterns that provide default implementations without using overlapping instances, they have not yet received as much testing as instance chains.

Related work. This paper describes a use of the Hackage repository for language design; we believe it is one of the first such descriptions. However, there have been several similar projects. We were strongly guided by Duncan Coutts' description of using Hackage for regression testing [2]. Another inspiration came from Andrew Wright's study of the value restriction in Standard ML [15], which studied a wide variety of ML programs to determined whether a language design choice was justified.

Future work. As discussed in Sections 2 and 3, there are numerous ways that our survey could be improved, and were we to perform the survey now we would have access to significantly more data. Despite this, we believe the survey as performed captured a representative sample of the use of overlapping instances on Hackage. Therefore, we are not currently intending to revisit this survey. We are, however, hoping to find other language design questions amenable to our general approach.

Should we do so, there are several aspects of the survey that would be improved by additional automation. In particular, although we did parts of the separation of Hackage into consistent package sets manually, we imagine that it would be possible to automate it entirely. That would make updating the results of future surveys relatively painless.

Another interesting problem has to do with the generation of instrumented compilers. Despite the existing GHC API, we had two reasons for modifying GHC itself: first, because the data we needed was already computed while checking for duplicate instances, instrumenting the compiler there was particularly painless. Second, while telling the Cabal build process to use a particular (instrumented) GHC is quite simple, adding additional steps to the compilation process (such as running a separate program, built using the GHC API, to collect overlap information) is more complex. However, this also leads to disadvantages: the output from our instrumentation process is intertwined with the regular output from GHC, and modifying and building GHC is a heavyweight process for relatively simple instrumentation.

Acknowledgements. Thanks to Mark Jones for his advice during the conception, execution, and description of this survey, and to the anonymous reviewers for their helpful feedback and discussion of the submitted draft of this work.

References

[1] D. Coutts. Solving the diamond dependency problem. `http://blog.well-typed.com/2008/08/solving-the-diamond-dependency-problem/`, 2008. Last accessed June 8, 2010.

[2] D. Coutts. Regression testing with hackage. `http://blog.well-typed.com/2009/03/regression-testing-with-hackage/`, 2009. Last accessed June 8, 2010.

[3] M. P. Jones. Hugs 98. `http://haskell.org/hugs`.

[4] Lemmih, P. Martini, B. Bringert, I. Potoczny-Jones, and D. Coutts. cabal-install: The command-line interface for cabal and hackage. `http://hackage.haskell.org/package/cabal-install`. Last accessed June 7, 2010.

[5] J. G. Morris and M. P. Jones. Instance chains: Type-class programming without overlapping instances. In *ICFP '10*, Baltimore, MD, 2010. ACM.

[6] D. Orchard and T. Schrijvers. Haskell type constraints unleashed. *Lecture Notes in Computer Science*, 6009:56–71, 2010.

[7] S. Peyton Jones, M. P. Jones, and E. Meijer. Type classes: an exploration of the design space. In *Haskell '97*, Amsterdam, The Netherlands, 1997.

[8] T. Schrijvers, S. Peyton Jones, M. Chakravarty, and M. Sulzmann. Type checking with open type functions. In *IFCP '08*, pages 51–62, Victoria, BC, Canada, 2008. ACM.

[9] D. Stewart. Re: [Haskell-cafe] Overlapping/Incoherent instances. `http://www.haskell.org/pipermail/haskell-cafe/2008-October/049155.html`, 2008. Last accessed June 8, 2010.

[10] W. Swierstra. Data types à la carte. *JFP*, 18(04):423–436, 2008.

[11] The Cabal Team. #435 (ban upwardly open version ranges in dependencies on base). `http://hackage.haskell.org/trac/hackage/ticket/435`, 2009. Last accessed June 8, 2010.

[12] The GHC Team. GHC. `http://haskell.org/ghc`, 2009.

[13] R. van Dijk. Ann: Reverse dependencies in hackage (demo). `http://www.haskell.org/pipermail/haskell/2009-October/021691.html`, 2009. Last accessed June 8, 2010.

[14] P. Wadler and S. Blott. How to make ad-hoc polymorphism less ad hoc. In *POPL '89*, pages 60–76, Austin, Texas, United States, 1989. ACM.

[15] A. K. Wright. Simple imperative polymorphism. *Lisp and Symbolic Computation*, 8(4):343–355, 1995.