A Type-Sensitive Preprocessor For Haskell

Noel Winstanley
Department of Computer Science
University of Glasgow

September 4, 1997

Abstract

This paper presents a preprocessor which generates code from type definitions. The preprocessor uses rules to describe the pattern of the produced code. In this way, instances may be derived for type classes. Utility functions for manipulating types, especially records, can also be produced. The set of rules is easily extendible; by adding derivation rules for new type classes, instance derivation is possible for classes defined outside the prelude.

1 Introduction

The lazy functional language Haskell [6, 1] features *type classes*; these allow a set of common operations to be defined over different types. A type is made an *instance* of a class by describing how the operations of the class behave upon it. The standard prelude uses classes to allow equality, conversion to and from strings, and arithmetic to use a uniform syntax over different types. When the programmer declares a new type, instances of classes may be coded by hand. However, instances for some of the prelude classes can be automatically generated by the compiler. This is possible because the behaviour of the class operations follows a well-defined pattern. When declaring a type, the programmer may specify which of these classes to *derive* instances for; this reduces the amount of repetitive code that must be written and removes a possible source of errors.

Instance derivation is only supported for some prelude classes. However, many programmer-defined classes will have the same property: their operations follow the same pattern for any type they may be instantiated over. Unfortunately, Haskell provides no way to extend the number of automatically derivable classes: this requires a way to express the pattern that instances follow. It is unfortunate that the number of derivable classes is fixed. Ideally, it should be possible to derive instances for classes defined in libraries in the same way as for prelude classes.

This problem was identified in a Glasgow Parallel Haskell paper discussing evaluation strategies [8]. Strategies use a class `NFData` to control the degree to

which a value is evaluated. When parallelising a program, instances of this class need to be produced for all the program's types. The paper commented that it was a shame that instances of `NFData` could not be automatically derived; the form of the instance is simple, and writing by hand is tiresome. Many classes' instances are equally simple. This is what makes coding instances tedious: there is so little thought involved.

A post[1] on the Haskell mailing list made another criticism of the deriving mechanism. It questioned why instance derivation was linked to type definitions. A programmer could import a type that did not have instances for some of the derivable classes. If these instances were required, they either had to be coded by hand or the module containing the type definition edited and recompiled. The first alternative is tedious, while the second is unsuitable when using large libraries of code — changes are spread over many modules rather than extending the code with additional modules.

Another observation is that whenever a type is defined, it is common to write utility functions to manipulate the new type. These perform such operations as testing for a constructor or field, projecting the value from a `newtype` and updating a record field. Such functions have a common pattern for each type they are defined over, but cannot be captured in a class. A way to automatically generate these utilities, in a similar way to instance derivation, would be useful.

2 The Preprocessor

A preprocessor, called Derive[2], was written to alleviate the three problems listed above. Derive contains a set of rules which each describe a block of code to generate for a type. There are predefined rules to produce instances for all the derivable classes in the Haskell Prelude. There are also rules to produces instances of `NFData` and to generate utility functions for types and records.

The input to Derive is a Haskell module annotated with preprocessor commands. The input is parsed for type definitions (`data` and `newtype` statements) and commands. Monadic parser combinators [3] were used to construct the parser, which was based on the example given in the paper. It was a simple task to extend the parser to accept record types and preprocessor commands. The commands cause rules to be applied to the parsed types, generating new code which is then appended to the bottom of the module. The preprocessor accepts commands which apply rules to a newly-defined type, an imported type, and set default rules to apply to all types in the module.

It is intended that the user may add new rules to the preprocessor; this allows authors of library code to provide rules for deriving instances of new classes. Derive is written entirely in Haskell. This means that the user does not have to learn a new language and can use all Haskell's features to express rules.

A rule is a tuple of a string and a function. The string identifies the rule in commands in the input. The function maps between the parsed representation

[1] from Olaf Chitil `<chitil@informatik.rwth-aachen.de>`

[2] Documentation and Code available at `http://www.dcs.gla.ac.uk/~ nww/derivehome.html`

of a type and the text to be output — an un-parser. An effort has been made to make writing rules as easy as possible. The text generated by a rule is described using pretty-printing combinators [2]. Working with these combinators is easier than manipulating strings, as they provide a more structured way to express rules and handle the indentation needed by the offside rule. The output is also well formatted and easy on the eye.

As mentioned above, Derive is able to generate code in a module from an imported type. This is achieved by searching for the module containing the type definition. In addition to understanding literate and non-literate scripts, the preprocessor can also extract type definitions from interface files generated by the GHC compiler [7]. This ability is especially useful for adding instances to a previously-compiled type. This feature can cause problems — it is possible for a program to contain duplicate instances for a type. However, when used carefully, the ability to add instances later in program development is very useful.

3 Evaluation

Derive is restricted to adding code to a module: existing code cannot be altered. This prevents it from performing tasks such as inserting class contexts in type signatures. This would be useful when a new instance has been produced for a type. To allow Derive to manipulate the existing code in a module a full Haskell parser would be needed. Additionally, the command syntax and rule structure would have to be extended. This would greatly increase the complexity of the preprocessor, make rules harder to express and the commands harder to use.

Another limitation is that rules only operate on a single type — code cannot be generated by combining information from two or more types. Although this does not seem to be too restrictive at the moment, when multi-parameter type classes are added to Haskell this ability will become very important. Adding this feature would involve restructuring the program so that a store of parsed types is created, which could then be accessed in a random fashion by commands.

More practical problems are limited error tolerance and reporting. Currently, if a type cannot be found the preprocessor aborts with an informative error. This needs to be changed so that Derive dies gracefully, closing files and giving a detailed explanation of the cause of the failure.

A quirk of Derive is that the source code must be edited when adding new rules. Although the modules containing user-defined rules are separated from the rest of the code, this approach is unsatisfactory. Specific rules belong logically to the project or library they are written to support, rather than the preprocessor that operates upon them — for a new project a new makefile is written; the make program remains unchanged. However, to read in scripts containing rules would require Derive to become a fully-fledged Haskell interpreter. This is much too complicated. However, an interesting development is a proposal to enable the Hugs interpreter to dynamically load in new modules during the execution of a script. This is exactly what is required for Derive. It

remains to be seen whether this feature will be implemented.

Ideally there would be a mechanism within the Haskell language or in the compiler to add new derivable classes. Failing this, a preprocessor is an acceptable alternative. Adding new rules by editing the preprocessor source code is inconvenient, but keeps the preprocessor simple while allowing the programmer to use the full Haskell language.

4 Polytypic Programming

Although Derive was not written with polytypic programming in mind, it has been found that it has the required features. Polytypic programming [5] is the use of functions that can operate upon any data structure, regardless of its shape. For example, while a conventional map applies a function to elements of a list, a polytypic map applies a function to every element of *any* datatype. A polytypic map has similar behaviour whether the datatype is a list, tree or any other structure.

A well known polytypic extension to Haskell is PolyP [4]. This is a preprocessor that adds polytypic functions to a subset of Haskell, using the new keyword `polytypic`. PolyP can generate instances of map, zip, size, catamorphisms, etc. for regular types, which are a subset of the definable datatypes.

Initially it was thought that PolyP and Derive had different uses. PolyP provides new language features, while Derive makes existing features easier to use. Thus it was thought that PolyP added expressiveness to the language, while Derive did not.

Subsequently, it was realised that polytypic programming only requires the ability to automatically derive classes — which is what Derive provides. Derive adds no language constructs; however the preprocessor itself adds a capability that is not available in the language. Each of the examples in the PolyP paper can be re-written using Derive (other than those that do not really apply; `in` and `out`, for example, just translate between types and functors). In addition, Derive can be used for applications other than polytypic programming.

To summarise the significant differences between PolyP and Derive —

- PolyP generates instances from function uses; Derive requires that the derivations be requested, either globally or explicitly. In this sense PolyP has more power.
- Derive is more flexible in generating names from the structure of a type. There is no way to do this in PolyP.
- 'Functors' are used in PolyP to represent the structure of types. This approach is more opaque and provides less information than the representation of types used by Derive.
- Derive's input is Haskell code missing definitions that are to be generated by the preprocessor. Input to PolyP is a new Haskell-based language.

This difference may be important when using other tools in conjunction with these preprocessors.

5 Conclusion

Although it has limitations, the Derive preprocessor is a useful tool to support automatic derivation for user-defined classes. The WWW server log shows a considerable amount of people have downloaded the distribution. The author has received correspondence commenting on how easy it is to write rules, and how much time has been saved by using the preprocessor.

References

[1] P. Hudak and J. H. Fasel. A gentle introduction to haskell. *SIGPLAN Notices*, 27(5), May 1992.

[2] John Hughes. The design of a pretty-printing library. In J. Jeuring and E. Meijer, editors, *Advanced Functional Programming*, volume 925 of *LNCS*. Springer Verlag, 1995.

[3] Graham Hutton and Erik Meijer. Monadic parser combinators. Technical Report NOTTCS-TR-96-4, University of Nottingham, December 1996.

[4] Patrik Jansson and Johan Jeuring. PolyP—a polytypic programming language extension. In *Conference Record of POPL '97: The 24th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pages 470–482, Paris, France, 15–17 January 1997.

[5] Johan Jeuring and Patrik Jansson. Polytypic programming. In J. Launchbury, E. Meijer, and T. Sheard, editors, *Tutorial Text 2nd Intl. School on Advanced Functional Programming, Olympia, WA, USA, 26–30 Aug 1996*, volume 1129 of *Lecture Notes in Computer Science*, pages 68–114. Springer-Verlag, Berlin, 1996.

[6] J. Peterson[editor], K. Hammond[editor], L. Augustsson, B. Boutel, W. Burton, J. Fasel, A. Gordon, J. Hughes, P. Hudak, T. Johnsson, M. Jones, S. Peyton Jones, A. Reid, and P. Wadler. Haskell 1.3, A non-strict, purely functional language. Report YALEU / DCS / RR-1106, Department of Computer Science, Yale University, May 1996.

[7] The GHC Team. *The Glorious Glasgow Haskell Compilation System, User's Guide*, v2.01 edition, jul 1996.

[8] P W Trinder, K Hammond, H W Loidl, and S Peytom Jones. Algorithm + Strategy = Parallelism. Accepted for publication in Journal of Functional Programming, December 1996.