Haskell Tools from the Programatica Project

Demo Abstract

Thomas Hallgren
OGI School of Science & Engineering
Oregon Health & Science University
20000 NW Walker Rd, Beaverton, Oregon, USA
http://www.cse.ogi.edu/~hallgren/

Categories and Subject Descriptors

D.3.4 [**Programming Languages**]: Processors; D.2.6 [**Software Engineering**]: Programming Environments

General Terms

Languages

Keywords

Haskell tools

1. INTRODUCTION

One of the goals of the Programatica Project is to develop tool support for *high-assurance* programming in Haskell [21]. We have extended Haskell with syntax for *property assertions*, and envision the use of various techniques to provide *evidence* for the validity of assertions. We expect our tools to assist the programmer with *evidence management*, using *certificates* to record evidence, and to provide whatever translation of Haskell code needed to enable the use of theorem provers and other tools that can serve as sources of evidence.

The Programatica Tools, while still work in progress, can manipulate Haskell programs in various ways and have some support for evidence management. In Section 2, we describe a selection of the functionality provided by the tools, starting with functionality that might be of interest to Haskell programmers in general, and ending with functionality more directly aimed at supporting the goals of the Programatica Project. Section 3 contains some notes on the implementation.

2. TOOLS

2.1 Basic command-line tools

The functionality of our tools is available through a simple command line interface. Some functionality is also available through a graphical interface (Section 2.3). The main command is called `pfe` (Programatica Front-End).

The tools operate on a *project*, which is simply a collection of files containing Haskell source code. The command `pfe new` creates a new project, while `pfe add` adds files and `pfe remove` removes files from a project. There is also a command `pfe chase` to search for source files containing needed modules. This is *the only* command that assumes a relationship between module names and file names. A wrapper script `pfesetup` uses `pfe new` and `pfe chase` to create a project in a way that mimics how you would compile a program with `ghc --make` [8] or load it in Hugs [12].

Like batch compilers, `pfe` caches various information in files between runs (e.g., type information). `pfe` also caches the module dependency graph, while other systems parse source files to rediscover it every time they are run. This allows `pfe` to detect if something has changed in a fraction of a second, even for projects that contain hundreds of modules.

Once a project has been set up, the user can use `pfe` for various types of queries. For example, `pfe iface` M displays the interface of module M, `pfe find` x lists modules that export something called x and `pfe uses` $M.x$ lists all places where the entity called x, defined in Module M, is referenced.

2.2 The HTML renderer

The command `pfe webpages` generates web pages for a project, with one page per module. Every identifier in the generated HTML code is linked to its definition. Extensive tagging is used and a reference to a style sheet is made, allowing the user to customize the look of the resulting web pages. An example is show in Figure 1.

While the syntax highlighting provided by editors such as Emacs and Vim is based on a simple lexical analysis, our tool also makes use of syntax analysis to distinguish between type constructors and data constructors. Also, for hyperlinking identifiers to their definition, a full implementation of the module system and the scoping rules of Haskell are used. Although the source code is parsed, the HTML code is *not* generated by pretty-printing the abstract syntax tree, but by decorating the output from the first pass of our lexical analyzer [10], preserving layout and comments.

We also support a simple markup language for use in comments, keeping the plain ASCII presentation of source text readable, while, at the same time, making it possible to generate nice looking LaTeX and HTML from it. This was used in the preparation of our paper about the Haskell 98 Module

Haskell'03, August 25, 2003, Uppsala, Sweden.

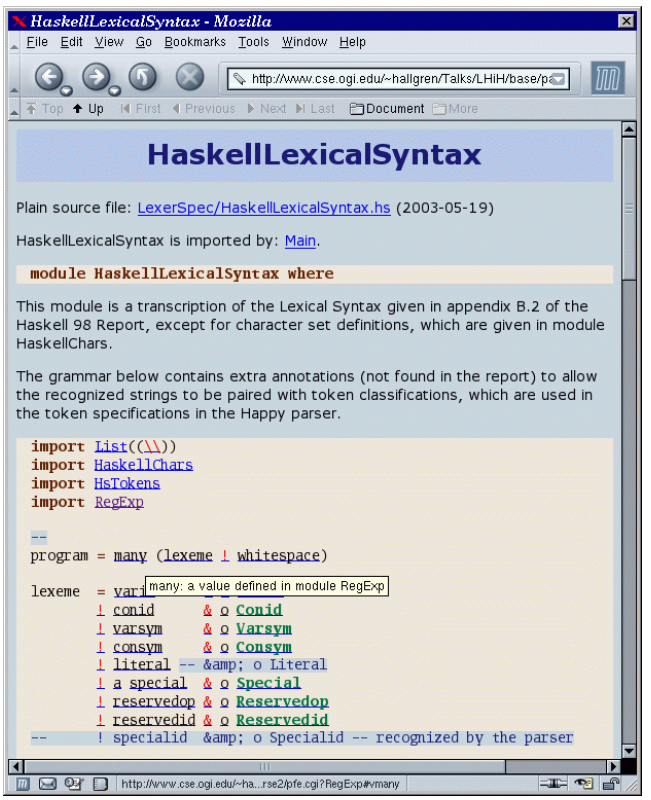

Figure 1: A sample Haskell module rendered as HTML by the Programatica Tools.

System [7].

Our HTML renderer is a tool for documenting and presenting *source code*, in contrast to Haddock [15], which is a tool for producing user documentation for *libraries*.

2.3 The Haskell source code Browser

While HTML rendering makes it possible to use a standard web browser to browse Haskell programs, we have also implemented a dedicated Haskell browser (Figure 2). It assumes that a project has already been set up with `pfe`, and is started with the command `pfebrowser`. It provides a syntax highlighted and hyperlinked source view similar to the one provided in the HTML rendering, but it also has some additional functionality. For example, the user can click on an identifier to find out what its type is. The browser can reload and retypecheck a module after it has been edited. Information is cached internally to make this quick. Reloading and retypechecking a moderately sized (a few hundred lines) module, takes just a second or two.[1]

At present, the tools type check complete modules, and do not provide any type information when type checking fails. In the future, we might make the type checker more incremental, providing partial information in the presence of type errors.

In the future, we might also turn the browser into a Haskell editor. Another possibility is creating a dedicated proof tool for Haskell, perhaps similar the proof tool Sparkle [6] for Clean.

[1]It is nowhere near the speed of Hugs, though...

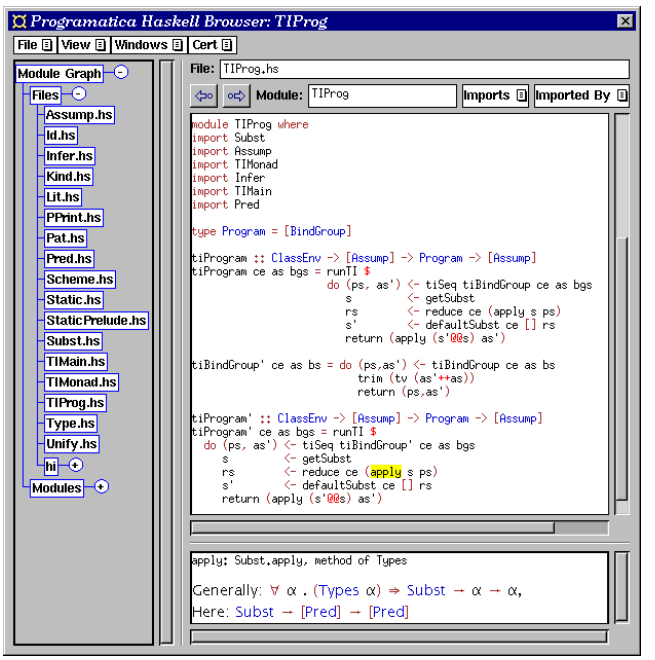

Figure 2: The Programatica Haskell Browser. The user has clicked on a use of `apply` to confirm that it applies a substitution to a list of predicates.

2.4 Program Slicing

Viewing a program as a set of definitions, the slicer computes, given an identifier, the subset of the definitions that are referenced, directly or indirectly, from that identifier. For example, by slicing with `Main.main` as the starting point, you can remove dead code from a program. This functionality is provided by the command `pfe slice` $M.x$. The output is a set of valid Haskell modules that, for example, can be loaded in Hugs. (It will not be a complete Haskell *program* if slicing starts from something other than `Main.main`.)

The dependency analysis is performed on type-checked code, allowing unused instance declarations to be eliminated. However, to keep the slicer simple, top-level declarations are treated as atomic units[2] and this can make the slicing coarse, for example including definitions of default methods (and everything they depend on) in class declarations, even if they aren't needed by any instance declarations.

The slicer preserves the module structure of the source program. Import and export declarations are adjusted appropriately.

Because of the annoying monomorphism restriction, eliminating a definition that *is not used* can change the meaning of a declaration that *is used*. Dependencies caused by this side effect are currently not taken into account by our slicer.

2.5 Translation to Structured Type Theory

The translation to Structured Type Theory [5] allows asserted properties to be proved formally in the proof editor Alfa [9]. The translation is based on type checked code. Polymorphic functions are turned into functions with explicit type parameters and overloading is handled by the

[2]Actually, type signatures and fixity declarations with more than one identifier are split up.

usual dictionary translation, turning class declarations into record type declarations and instance declarations into record value definitions.

The syntax of Structured Type Theory is simpler than that of Haskell, so a number of simplifying program transformations are performed as part of the translation. This affects, for example, list comprehensions, the do-notation, pattern matching, derived instances and literals. There are also various ad-hoc transformations to work around syntactic restrictions and limitations of the type checker used in Alfa. Apart from this, the translated code looks fairly similar to the original Haskell code. (An example can be seen in Figure 4.)

While Haskell has separate name spaces for types/classes, values and modules, Structured Type Theory has only one, so some name mangling is required to avoid name clashes. This is done in a context-independent way, so that when some Haskell code is modified, the translation of unmodified parts remains unchanged, allowing proofs about unmodified parts to be reused unchanged.

While type theory formally only deals with total functions over finite data structures, the translator allows any Haskell program to be translated. Alfa contains a syntactic termination checker [1] that can verify that a set of structurally recursive function definitions are total. When the translation falls outside that set, any correctness proofs constructed in Alfa entail only *partial correctness*, and we leave it to the user to judge the value of such proofs. In the future, we might use a more sophisticated termination checker and/or change the translation to use *domain predicates* [2] to make partiality explicit and to make the user responsible for providing termination proofs.

While the type system of plain Haskell 98 can be mapped in a straight-forward way to the *predicative* type system of Structured Type Theory, we foresee problems extending the translation to cover existential quantification in data types and perhaps also higher rank polymorphism.

2.6 Evidence management

At the moment, the command line tools and the browser support the creation and validation of certificates for three basic forms of evidence: informal claims ("I say so"), tests run with QuickCheck [4], and formal proofs in Alfa [9]. There is work in progress on support for other forms of evidence, and our implementation has an extensible architecture to make it easy to plug them in using *certificate servers* to act as bridges between our code and external tools.

When source code is changed, the validity of existing certificates can be affected. Revalidating certificates might require some work by the user. To help identify those changes that can actually influence the validity of a certificate, dependencies are tracked on the definition level (rather than the module level). Changes are detected by comparing hash values computed from the abstract syntax. As a result, we avoid false alarms trigged by changes to irrelevant parts of a module, and changes that only affect comments or the layout of code.

Just to give a simple example, Figure 3 shows a Haskell module containing three valid certificates, two Alfa certificates and one QuickCheck certificate. Figure 4 shows the translation of the module to Alfa and the proofs of the two Alfa certificates.

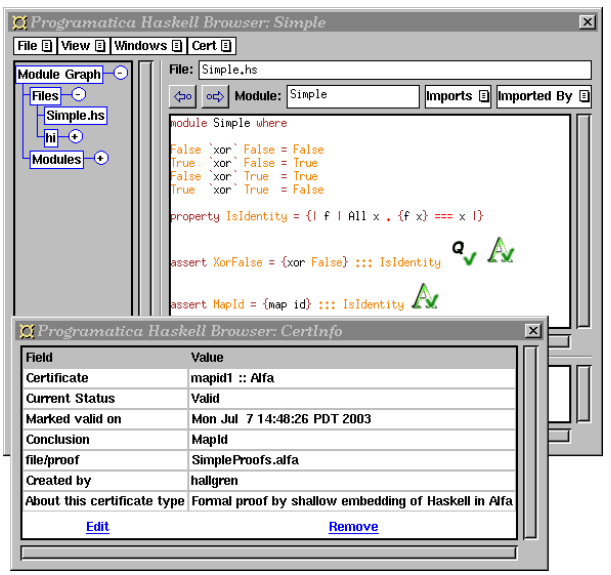

Figure 3: The module `Simple` containing three valid certificates, viewed in the Programatica Haskell Browser.

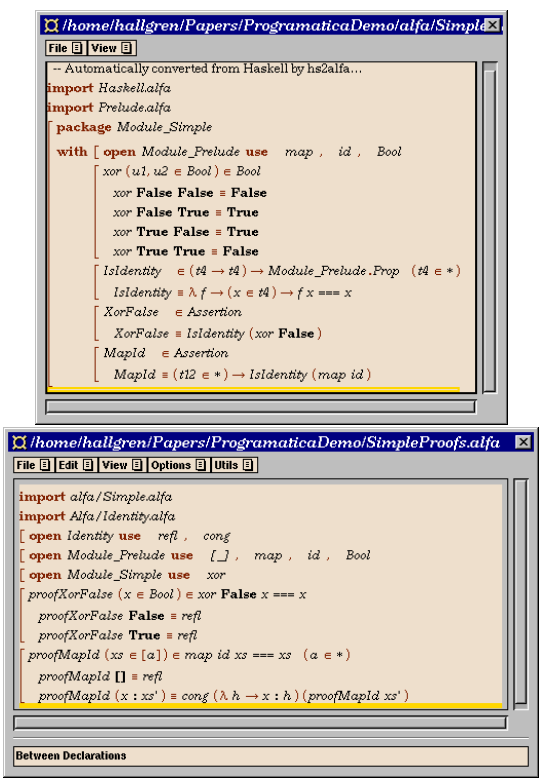

Figure 4: Translation of module `Simple` in Figure 3 to Structured Type Theory and proofs of the two Alfa certificates.

3. IMPLEMENTATION NOTES

Not surprisingly, our implementation has a lot in common with a Haskell compiler front-end, and is likely to be reusable in many contexts outside the Programatica project. For example, it has already been used in the refactoring project at the University of Kent [22].

Our tools are implemented in Haskell, including the browser, which uses Fudgets [3] for the user interface. Amongst other things, our implementation contains an abstract syntax; a lexer and a parser; a pretty printer; some static analyses, including inter-module name resolution; some program transformations; a type checker and definition-level dependency tracking.

Some parts – the abstract syntax, the parser and the pretty printer – were inherited from another source [16] and modified, while most other parts were written from scratch. Some parts – the lexer [10] and the module system [7] – are implemented in pure Haskell 98 and can serve as reference implementations, while others make use of type system extensions, in particular multi-parameter classes with functional dependencies [13]. Together with a two-level approach to the abstract syntax [19] and other design choices, this makes the code more modular and reusable, but perhaps also too complicated to serve as a reference implementation of Haskell.

The fact that the first pass of our lexer preserves white space and comments made it easy to implement the HTML renderer. The modular structure of the lexer also allowed us to quickly create a simple tool that someone asked for on the Haskell mailing list: a program that removes comments and blank lines from Haskell code [11].

While implemented from scratch, key design choices in the type checker were influenced by the simplicity of *Typing Haskell in Haskell* [14], the efficiency of the type checkers in HBC and NHC, and the constraint based approach used in one of Johan Nordlander's type checkers for O'Haskell [17]. It performs the dictionary translation and inserts enough type annotations to make the output suitable for translation to an explicitly typed language like Structured Type Theory or System F.

4. CONCLUSION

More information on the Programatica Project is available from our web pages [18]. Preliminary versions of the tools and user documentation can be downloaded from [20].

Acknowledgments. The author would like to thank Johan Jeuring, Mark Jones and John Matthews for suggesting improvements to this abstract. Also, although this abstract has one author, many people contributed to the work it describes.

5. REFERENCES

[1] Andreas Abel. foetus – Termination Checker for Simple Functional Programs. `www.tcs.informatik.uni-muenchen.de/~abel/foetus/`, 1998. Programming Lab Report.

[2] Ana Bove. *General Recursion in Type Theory*. PhD thesis, Department of Computer Science and Engineering, Chalmers University of Technology, Gteborg, Sweden, 2002. `www.cs.chalmers.se/~bove/Papers/phd_thesis.ps.gz`.

[3] Magnus Carlsson and Thomas Hallgren. Fudgets. `www.cs.chalmers.se/Fudgets/`, 1993-2003.

[4] Koen Claessen and John Hughes. Quickcheck: a lightweight tool for random testing of haskell programs. In *International Conference on Functional Programming*, pages 268–279. ACM, 2000. `citeseer.nj.nec.com/claessen99quickcheck.html`.

[5] Thierry Coquand. Structured type theory. `www.cs.chalmers.se/~coquand/STT.ps.Z`, June 1999. Preliminary version.

[6] Maarten de Mol. Sparkle. `www.cs.kun.nl/Sparkle/`, 2003.

[7] Iavor S. Diatchki, Mark P. Jones, and Thomas Hallgren. A Formal Specification for the Haskell 98 Module System. In *Proceedings of the 2002 Haskell Workshop*, Pittsburgh, USA, October 2002. `www.cse.ogi.edu/~diatchki/hsmod/`.

[8] The Glasgow Haskell Compiler, 2002. `www.haskell.org/ghc/`.

[9] Thomas Hallgren. Home Page of the Proof Editor Alfa. www.cs.chalmers.se/ hallgren/Alfa/, 1996-2003.

[10] Thomas Hallgren. A Lexer for Haskell in Haskell. `www.cse.ogi.edu/~hallgren/Talks/LHiH`, 2002.

[11] Thomas Hallgren. stripcomments. `www.cse.ogi.edu/~hallgren/stripcomments/`, 2002.

[12] Hugs Online. `www.haskell.org/hugs/`, 2002.

[13] Mark P. Jones. Type Classes with Functional Dependencies. In *Proceedings of the 9th European Symposium on Programming, ESOP 2000*, number 1782 in LNCS, Berlin, Germany, March 2000. Springer-Verlag.

[14] M.P. Jones. Typing Haskell in Haskell. In *Proceedings of the 1999 Haskell Workshop*, Paris, France, September 1999. `www.cse.ogi.edu/~mpj/thih/`.

[15] Simon Marlow. Haddock. `www.haskell.org/haddock/`, 2003.

[16] Simon Marlow et al. The hssource library. Distributed with GHC [8].

[17] Johan Nordlander. O'haskell. `www.cs.chalmers.se/~nordland/ohaskell/`, 2001.

[18] The Programatica Project home page. `www.cse.ogi.edu/PacSoft/projects/programatica/`, 2002.

[19] Tim Sheard. Generic unification via two-level types and parameterized modules. In *International Conference on Functional Programming*, pages 86–97, 2001. `citeseer.nj.nec.com/451401.html`.

[20] The Programatica Team. Programatica Tools. `www.cse.ogi.edu/~hallgren/Programatica/download/`, August 2003.

[21] The Programatica Team. Programatica Tools for Certifiable, Auditable Development of High-assurance Systems in Haskell. In *Proceedings of the High Confidence Software and Systems Conference*. National Security Agency, April 2003. Available via [18].

[22] Simon Thompson, Claus Reinke, et al. Refactoring Functional Programs. `www.cs.kent.ac.uk/projects/refactor-fp/`, 2003.