Visual Haskell

A full-featured Haskell development environment

Krasimir Angelov
kr.angelov@gmail.com

Simon Marlow
Microsoft Research Ltd, Cambridge, U.K
simonmar@microsoft.com

Abstract

We describe the design and implementation of a full-featured Haskell development environment, based on Microsoft's extensible Visual Studio environment.

Visual Haskell provides a number of features not found in existing Haskell development environments: interactive error-checking, displaying of inferred types in the editor, and other features based on static properties of the source code. Visual Haskell also provides full support for developing and building multi-module Haskell projects, based on the Cabal architecture. Visual Haskell supports the full GHC language, and can be used to develop real Haskell applications (including the code of the plugin itself).

Visual Haskell has driven developments in other Haskell-related projects: Cabal, the Concurrent FFI extension, and an API to allow programmatic access to GHC itself. Furthermore, development of the Visual Haskell plugin required industrial-strength foreign language interoperability; we describe all our experiences in detail.

Categories and Subject Descriptors D.2.6 [*Programming Environments*]: Integrated Environments

General Terms Languages, Design

Keywords Visual Studio, Haskell Development Environment

1. Introduction

Haskell suffers from the lack of a decent development environment. For programmers used to the elaborate environments available for more mainstream languages (eg. Microsoft Visual Studio, Borland C++ Builder/JBuilder, KDevelop, Eclipse, JCreator), Haskell's paltry offerings seem positively primitive.

There is a Haskell mode for Emacs [16], which is a fine text editor, but it falls short of providing any real help to the Haskell programmer beyond simple syntactic colouring and an attempt at automatic indentation. Even the colouring support fails to correctly colour source code in several cases, based as it is on regular expressions rather than a real lexical analyser. Support for multi-module programs and libraries is limited, relying largely on external tools with no integration in the environment. The support for automatic indentation is based on heuristics rather than a real knowledge of the syntactic structure of the code, so inevitably it often fails to work.

Emacs can provide a menu of the functions defined in a source code module, but it does this by looking for type signatures using regular expression matching, so it gets confused in certain cases: comments in the wrong place, signatures split over multiple lines, and pre-processors can all cause it to give wrong results.

There are other programming environments for Haskell available [9, 4, 3, 19, 20, 2], and some of these improve on the Emacs support in various ways, but they all stop short of providing an environment with real knowledge of the structure of the code being developed. hIDE looked the most promising, but it hasn't seen any updates for 3 years. There are also source-code browsers for Haskell [17, 5, 10, 6, 14], but these require a separate processing step to obtain the results. The programmer would be better served by having the information available immediately and interactively while developing the code.

This paper makes the following contributions. Our contributions are primarily of the tools and engineering experience variety rather than research results:

- We describe the first full-featured development environment for Haskell, implemented as a plugin for Microsoft's multilingual Visual Studio environment (Sections 3.1–4). The key points are:
  - In contrast to existing Haskell environments, the editor communicates directly with the compiler, and has full knowledge of the structure of the program, not only on a syntactic level but also including full typing information. This enables the environment to provide advanced editing features; for example, real-time error checking and interactive display of types. We have still only scratched the surface of what is possible: Section 6 gives some ideas for future enhancements. Nevertheless, in some ways, our system is more advanced than the Visual Studio environments for C++ and other languages.
  - Our environment has an advanced project and build system, which draws on the facilities provided by Cabal [11]. It fully supports multi-module libraries and applications, and doesn't require the programmer to write a single Makefile (Section 4). Projects developed in Visual Studio can be compiled and installed on platforms without Visual Studio

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. To copy otherwise, to republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee.
Haskell'05 September 30, 2005, Tallinn, Estonia.
Copyright © 2005 ACM 1-59593-071-X/05/0009. . . $5.00.

installed[1], because a Visual Studio *project* is also a Cabal *package*, and Cabal packages can be built and installed on any system that has a Haskell compiler.

  ▪ The environment works with the full set of language and compiler features supported by GHC [1]. The benefits of this should not be underestimated: tools for the Haskell language often support only plain Haskell 98 or are restricted to a few of the extensions that GHC supports; and this means that programmers who need to use some of the more advanced features are quickly locked out of using the tools. Our use of GHC itself as a source-code analysis engine means that any program that works with GHC can be developed in Visual Haskell.

    In fact, the Visual Studio plugin is self-hosting: it can be developed and built inside Visual Studio itself. As you might imagine, the Visual Studio plugin uses various exotic features of GHC, and does some heavyweight foreign language interop, so this is no mean feat.

- We chose to implement our Visual Studio plugin in Haskell itself, and doing so has not been without difficulty. However, the process has been beneficial to Haskell in a wider sense. Extensions to Haskell, GHC, and Cabal have all been driven by the Visual Studio plugin:

  ▪ The Visual Studio plugin is required to be multithreaded, which turned out to be a serious testbed for GHC's implementation of the new FFI/Concurrency extensions [15], otherwise known as the "threaded RTS"; Visual Studio helped us refine the design and implementation of the threaded RTS.

  ▪ The requirements of Visual Studio were a key factor in the design of Cabal [11].

  ▪ The Visual Studio environment communicates with its plugins via COM [18]. Our existing Haskell/COM interop tools [7, 8] were stretched to their limits and beyond, but we now have valuable experience that can be brought to bear on designing enhancements or new variants of the tools (Section 5).

  ▪ The Visual Studio plugin needs to talk directly to the compiler, and this forced us (the GHC developers) to think about what a programmatic interface to GHC should look like (Section 3.2). We have now implemented the API, and it is used not just by Visual Studio but also by GHC's existing front-ends (GHCi and the command line interface), and other projects which require a Haskell front-end are starting to look into it too [12, 17].

Basing our environment on Visual Studio means that it is limited to the Windows platform, which is unfortunate. However, much of the work we have done is not platform-specific (the GHC API and Cabal in particular), so we hope these technologies can be leveraged to develop environments for other platforms.

2. A Tour of Visual Haskell

Visual Studio is a multilingual environment with integration for C++, C#, J#, Visual Basic and many other third party languages.

Figure 2. Errors in the editor

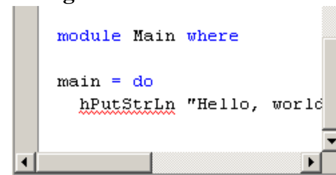
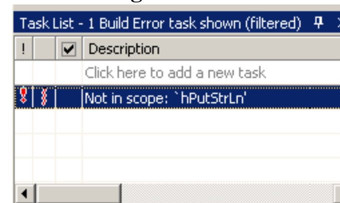


Figure 3. Tasks



The features that it provides go far beyond simple syntax colouring. There is support for projects, compilation, source browsing, source level quick info, word completion, automatic brace matching and many other language specific features. Our aim is to provide all these features for the Haskell programmer.

In this section, we illustrate the features of the Visual Haskell environment with screenshots, before going on to explain the implementation details in the following sections.

2.1 The Editor

Loading up a Haskell module into Visual Haskell presents the user with a screen similar to Figure 1. The first thing to notice is that elements of the source code are coloured according to their syntactic category (keyword, string, identifier etc.)[2]. Colouring happens as you type, and changes on the current line automatically propagate to the rest of the file as necessary (for example, opening a new multi-line comment).

The environment is also constantly checking the current source file for errors. Not just syntactic errors, but all violations of the Haskell static semantics — scoping errors, type errors, and so on — are checked for. If an error is found, the position of the error is indicated by underlining the erroneous code (Figure 2), and a task item is entered into the *task list* (Figure 3). When the user starts typing again, the underlining and the task are automatically removed. This checking happens in a background thread, so it doesn't disturb the interactive feel of the environment. Responsive interactive checking of the source code saves a great deal of time for the programmer, as the compile/edit cycle takes place within a single source code window, with no need to switch windows and issue commands. Furthermore, since we're using GHC itself for the interactive checking, and the checking takes place using exactly the same contextual information as will be available at compile time, we can guarantee that a file that checks in the editor will also compile for real.

Now, because the source code has been typechecked, we can provide the programmer with a great deal of information about the code. For example, the drop-down bar at the top of the window lists the entities defined in the module, and their types where appropriate (Figure 4).

[1] Provided the code itself is portable to the desired platform, of course.

[2] If you're reading this in monochrome, you'll have to take our word for it.

Figure 1. Visual Studio environment

Figure 4. Editor and Drop-down bar

Figure 5. Quick info

Hovering the mouse over an identifier causes the type of the identifier to pop up in a tip window (Figure 5). Pressing the key sequence for completion brings up a list of names in scope that can complete the identifier under the cursor (Figure 6). On the right-click context menu is an option to "go to definition": this jumps the cursor to the definition site for the identifier under the cursor, and it works both for top-level and locally-defined names, in the current module or another module in the project.

All of these features are supported by interactive checking of the source code, so the information on which they are based is updated constantly and never out of date.

Figure 6. Word completion

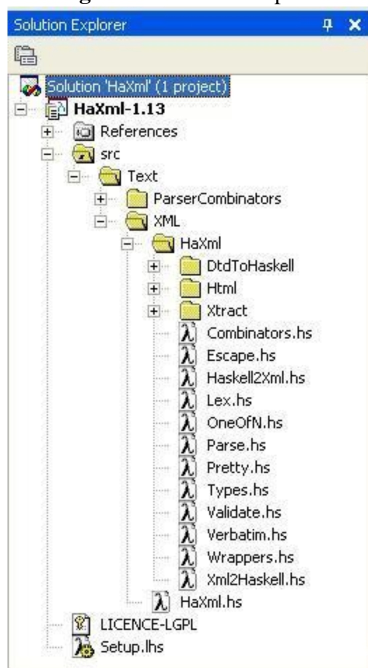
2.2 Projects

We have so far described the features available to the Haskell programmer when editing a single source file, but Visual Haskell also fully supports the construction of multi-module programs and libraries.

Figure 7 shows the "Solution Explorer" window in Visual Studio, displaying a Haskell project (HaXml in this case). A project should be thought of as a container for two kinds of information:

- Source files,
- Metadata, such as the options required to compile the sources and the dependencies on external packages.

Figure 7. Solution Explorer



The solution explorer's hierarchy reflects the filesystem: the folders are directories, and the leaves are files in the project. Some of the files are Haskell source files, and others are auxilliary, such as the `LICENSE` file. Files can be added and removed from the project, using the right-click context menu.

The *References* node in the hierarchy is special: it doesn't correspond to a real directory, and it contains an entry for every dependency of the current project on an external package. Dependencies have to be added manually; a possible future extension is to derive them automatically in some way.

A new source file is created using the "add new item" option, which yields the dialog in Figure 8. Various types of file can be selected, and the environment will then create a template source file – for example a Haskell module will have a `module` declaration based on the file name.

The meta-data associated with a project is edited via the project properties page, shown in Figure 9. The fields in the project properties are mostly descriptive meta-data, and have no semantic value. The only exception is the version number – version numbers of packages are used to resolve dependencies.

Figure 8. Add Item dialog

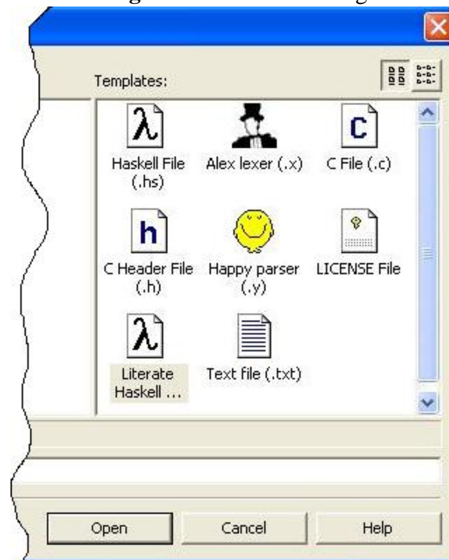
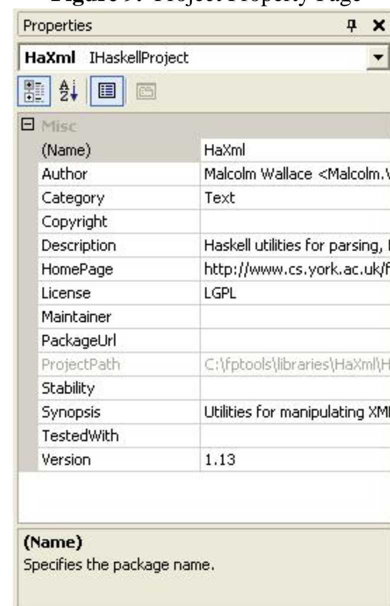


Figure 9. Project Property Page



2.2.1 The Class View

The "Class View"[3] (Figure 10) provides an overview of the structure of the code in a project. The top-level branches in the tree are modules, and underneath each module is a node for each of the top-level entities defined in that module (functions, classes, types, instances etc.). Clicking on a node navigates the editor to the definition site for the entity.

The class view is quite basic at the moment, but in the future there are various ways in which it could be extended. For example, we could include a rendition of the class hierarchy, and attach the instances of a class to the class itself.

[3] "Modules View" might be better name for this feature but we are just reusing the existing service which is used for C++, C# and other object oriented languages.

2.2.2 Building and testing the project

Compiling the project to an executable or library is fully supported within the environment. Selecting the build option from the menu will cause all the modules in the project to be compiled in dependency order. Any compile errors are entered as tasks in the task list, where clicking on the task will navigate to the correct source file and line containing the error. The raw textual output from the compiler is also available. If the project is a program, then it can be executed from within the environment too.

Figure 10. Class View

2.2.3 Projects and Cabal

Cabal, the "Common Architecture for Building Applications and Libraries", is a Haskell library providing facilities for configuring, building, and distributing Haskell software. Using Cabal, the author of a Haskell library or application gains access to a build system which works on any platform with a supported Haskell compiler, and also facilities for packaging and distributing their code in source or binary format. The unit of distribution is called a Cabal *package*, and may consist of a single library or application[4].

The connection between Cabal and Visual Haskell projects is an intimate one: a Visual Studio project *is* a Cabal package. The Visual Haskell project support is essentially a GUI for Cabal. In fact, the design of Cabal was heavily influenced by the requirements of Visual Haskell, to facilitate this isomorphism, as we will explain later.

When a project is created in Visual Haskell, a Cabal package is created. The file describing the Visual Haskell project is the same as the file describing a Cabal package, namely the `.cabal` file (see Figure 11 for an example `.cabal` file). Visual Haskell makes it easier to maintain and modify this file, by automatically filling in certain fields like the list of modules. However, since the syntax of this file is open and documented, the user may also edit it directly.

Since a Visual Studio project is also a Cabal package, the package can be built and installed on a system that does not have Visual Haskell. Building and installing the Cabal package only requires a Haskell compiler and the Cabal library, which is now distributed with all the compilers. This flexibility is an improvement over the other supported languages, which use their own built-in build systems, or require the programmer to work directly with Makefiles.

This correspondence also works in the other direction: existing Cabal packages developed on other systems can be loaded directly

[4] Currently multiple executables are supported, although the support is patchy and is expected to be replaced by a more general way to combine multiple packages later.

Figure 11. An example `.cabal` file

```
name:           HaXml
version:        1.13
license:        LGPL
license-file:   LICENCE-LGPL
author:         Malcolm Wallace <Malcolm.Wallace@cs.york.ac.uk>
homepage:       http://www.cs.york.ac.uk/fp/HaXml/
category:       Text
synopsis:       Utilities for manipulating XML documents
description:
        Haskell utilities for parsing, filtering, transforming and
        generating XML documents.
exposed-modules:
        Text.ParserCombinators.HuttonMeijerWallace,
        Text.XML.HaXml,
        Text.XML.HaXml.Combinators,
        Text.XML.HaXml.DtdToHaskell.Convert,
        Text.XML.HaXml.DtdToHaskell.Instance,
        Text.XML.HaXml.DtdToHaskell.TypeDef,
        Text.XML.HaXml.Escape,
        Text.XML.HaXml.Haskell2Xml,
        Text.XML.HaXml.Html.Generate,
        Text.XML.HaXml.Html.Parse,
        Text.XML.HaXml.Html.Pretty,
        Text.XML.HaXml.Lex,
        Text.XML.HaXml.OneOfN,
        Text.XML.HaXml.Parse,
        Text.XML.HaXml.Pretty,
        Text.XML.HaXml.Types,
        Text.XML.HaXml.Validate,
        Text.XML.HaXml.Verbatim,
        Text.XML.HaXml.Wrappers,
        Text.XML.HaXml.Xml2Haskell,
        Text.XML.HaXml.Xtract.Combinators,
        Text.XML.HaXml.Xtract.Lex,
        Text.XML.HaXml.Xtract.Parse
hs-source-dir:  src
build-depends:  base, haskell98
extensions:     CPP
```

into Visual Studio as a project. This is a significant win, because it means that a large (and growing fast) body of Haskell software can be developed directly in Visual Haskell without the need to create separate project files or ensure that the correct build options are propagated into the project's settings – all this happens automatically.

However, we should admit that there is not a true isomorphism between Cabal packages and Visual Studio projects. Cabal is designed to be flexible in the sense that it can accomodate virtually any existing package, including packages which have their own configuration and build systems. Since there is no general way for Visual Haskell to extract information such as the compiler options from a bespoke build system, Visual Haskell cannot completely support such packages. A possible future extension is to have a degenerate mode of Visual Haskell in which basic editing features are provided in the absence of complete package metadata.

2.3 Summary

To summarise, Visual Haskell provides the following Haskell-specific features:

- Syntax colouring.
- Drop down list with all declarations in the current module.
- Pop-up tips displaying the type of the identifier under the mouse.
- Word completion for any identifier in scope.

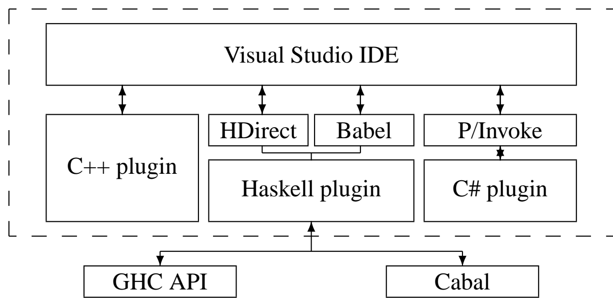


Figure 12. Overall Structure

- On the fly error checking.
- Jump to the definition of an identifier.
- Support for projects based on Cabal.
- Compilation and execution from the environment.
- Source code browser.
- Integrated documentation for GHC, all standard libraries, and other Haskell tools.

3. Implementation Walkthrough

In this section we describe the interesting aspects of the implementation of Visual Haskell, beginning with an overview of the structure, and then describing the implementation of specific features.

3.1 Overall structure

Visual Studio is a highly extensive environment. It is based around a small core, with all environment features implemented as plugins which can be installed independently. Each plugin may ask the core for specific services or provide its own services which become available to the other plugins. This simple and modular architecture is implemented on top of Microsoft's Common Object Model (COM) [18] which makes it possible to write new plugins in any COM-compatible language.

Figure 3.1 shows the structure of the complete system. The Visual Studio IDE communicates with the plugins via a published (but huge!) COM specification. Plugins may be implemented in any language, but our plugin is implemented in a combination of C++ and Haskell, for the following reasons:

- There is an existing C++ layer called Babel, which is provided with the Microsoft Visual Studio SDK. Babel consumes the large and complex COM API for language integration in Visual Studio's editor, and exposes a rather simpler COM API. Babel's purpose is to make it easier to add support for a new language to the Visual Studio editor, in this sense Babel is highly successful. However, Babel doesn't provide any support for Visual Studio features outside the editor (for example Projects), and it is lacking support for some editor features.

  Nevertheless, Babel saved us a lot of time getting started with our Visual Studio extension, and we are still using it – although we are now using a locally-modified version with support for some of the missing editor features and have modified Babel's API to more closely fit our requirements from a Haskell perspective. For implementing features not provided by Babel (such as Projects), we interact directly with the Visual Studio COM APIs.

- The bulk of the Haskell plugin is written in Haskell itself. We needed to use the code of GHC and Cabal, which are both Haskell libraries, so using Haskell for the rest of the plugin was the natural choice. However, this decision did lead to difficulties, because it meant that we had to do some heavyweight COM interop from within Haskell (see Section 5), for comminication both with Babel and direct to the Visual Studio APIs.

Babel is about 20,000 lines of C++. Rewriting Babel in Haskell would undoubtedly lead to a nicer end result, but would have been a lot of work. For comparison, the Haskell plugin code, not counting external libraries (GHC, Cabal, H/Direct) and not counting the IDL files with the COM API specifications, is about 8,000 lines.

3.1.1 Cabal

Since the Visual Studio environment also needs a build system — we need to be able to build and test code within the environment — the obvious solution was to build on Cabal, and make use of the build system it provides. We have already described the advantages to the user of our close coordination with Cabal, but from an implementation perspective this is a shrewd decision because it means we offload the work of developing and maintaining the build system to the Cabal maintainers, and we automatically benefit from future improvements.

The requirements of Visual Haskell influenced the design of Cabal itself in several ways. For instance, in the original Cabal design, the package specification was to be embedded in Haskell code (in the `Setup.lhs` file). However, this would have precluded editing the package specification using an external tool (eg. Visual Haskell), so the design was modified to store the package specification in a separate file with a well-defined concrete syntax.

3.1.2 GHC as a library

The Visual C++ and Visual C# plugins provide a rich set of IntelliSense® features (code completion, code browsing, go to definition, etc.). In order to implement some of these features for Haskell we decided to use the existing front-end (parser, static analysis and typechecker) from GHC. This idea was so successful that in some aspects Haskell provides much better IntelliSense® features than other languages. The GHC API and its integration with Visual Haskell is explained in the following Section.

3.2 Implementation of the editor features

The first thing that we did for Visual Haskell was to implement syntax colouring for the Visual Studio editor. Syntax colouring is implemented entirely via APIs provided by Babel, and did not require either modifications to Babel or direct interaction with Visual Studio, which enabled us to get something working quickly. We needed a small amount of infrastructure to build the Haskell plugin as a DLL and register it with Visual Studio and Babel, so that Visual Studio knew to invoke our DLL to obtain language-specific features for Haskell source code.

Colouring source code is straightforward: each time a line of source code needs to be coloured, Babel calls into our plugin passing the text of the line, a state value, and a callback function to invoke for each token. This interface allows the colouring state at the

beginning of each line to be represented by a single integer; the new state at the end of the line should be returned by the plugin after the line has been coloured.

Code can be coloured on any basis, but traditionally the lexical grammar of the language is used. For Haskell, we opted to use the lexical syntax – we could also use higher-level syntactic properties (e.g. colour types differently from code), but we are restricted by having to store the state of the colouriser in a single integer. The state for our Haskell colouriser is constructed by observing that the state at the beginning of a line can be either (a) inside a string, or (b) inside an arbitrarily-deep nesting of comments. For simplicity and speed, we built our colouring lexer using Alex [13], using a stripped-down Haskell lexical specification: it isn't necessary for the colouriser to handle layout, for example.

3.3 Editor features requiring GHC

The rest of the editor features require a more complete knowledge of the Haskell static semantics, up to and including typechecking.

The basic interface that we need to implement in our plugin is this call made by Babel into the plugin:

```
HRESULT ParseSource(
   [in] void *text,
   [in,unique] IParseSink* sink,
   [in] enum ParseReason reason,
   [out,unique,retval] IScope** scope
  );
```

`ParseSource` is called by Babel from a background thread; there is no restriction on the running time of `ParseSource`, because it doesn't interrupt the user interface thread(s).

`ParseSource` is intended to parse (and in our case, typecheck) the source code delivered in the `text` parameter, and report errors and warnings via the `sink` callback object. Additionally, `ParseSource` may construct and return an `IScope` object, which is used by Babel to further interrogate the plugin about aspects of the source code.

We could have implemented `ParseSource` by invoking a separate GHC process to compile the code. However, this would be slow, since the GHC process would have to re-read all the interfaces for external modules each time it is invoked. Clearly we would not achieve a responsive interactive feel this way. Moreover, the advanced editing features we intend to implement rely on having access to meta-information about the source code: types of identifiers and so on. Clearly the right approach is to hook into a compiler front-end directly.

Our initial implementation called direct into GHC to implement `ParseSource`. However, we quickly realised that a more principled interface to GHC was required, especially when we needed to extend the single-module view of the editor to a Project consisting of multiple modules. The following section describes the API we have designed for GHC.

3.3.1 The GHC API

The basic elements of the GHC API are given in Figure 13. The GHC API is a general programmatic interface to a Haskell compilation and execution engine. It supports typechecking and compilation of multi-module programs and libraries, and execution of code for use in an interactive environment such as GHCi. The GHC API is intended to support various user interfaces and tools. GHC's own two user interfaces are built over it: the command-line interface and GHCi. The API was also designed with other applications in mind, however: it has facilities designed to be used directly by a development environment supporting interactive checking of code, such as Visual Haskell, and it also provides access to the compiler's own type-decorated abstract syntax, to support tools that examine, analyse and manipulate Haskell programs.

Figure 13. The GHC API (abridged)

```
data Session -- abstract

data GhcMode
  = BatchCompile
  | Interactive
  | JustTypecheck
  | ...

newSession :: GhcMode -> IO Session

data Target
   = Target TargetId
            (Maybe (StringBuffer,ClockTime))

data TargetId
  = TargetModule Module
  | TargetFile FilePath

setTargets :: Session -> [Target] -> IO ()

data LoadHowMuch
   = LoadAllTargets
   | LoadUpTo Module
   | LoadDependenciesOf Module

load          :: Session
              -> LoadHowMuch
              -> (Messages -> IO ())
              -> IO SuccessFlag

checkModule   :: Session
              -> Module
              -> (Messages -> IO ())
              -> IO (Maybe CheckedModule)

getModuleInfo :: Session
              -> Module
              -> IO (Maybe ModuleInfo)

modInfoTyThings :: ModuleInfo -> [TyThing]
modInfoExports  :: ModuleInfo -> [Name]
...
```

An interaction with GHC is based around a `Session`. The `Session` is a mutable object containing the current state of the interaction with GHC: the modules that have been loaded, which flags are set, the contents of various caches, and so on. A `Session` is obtained by calling `newSession`, passing a `GhcMode` flag. The value `JustTypecheck` is designed specifically for development environments where only checking of the correctness of the code is required, rather than full compilation – this helps to improve the interactive response of the editor[5].

Once we have a `Session`, usually the next task is to load some code. This is achieved by first setting some `Targets`: these are the

[5] For boring engineering reasons, our current implementation of `JustTypecheck` does a little more work than it really needs to, in that it runs GHC's simplifier phase after typechecking. We plan to fix this shortly.

top-level modules that we want to compile; the rest of the module dependency graph will be discovered automatically. For example, a typical Haskell program will have just one `Target`: the `Main` module. A `Target` can be specified as a module name or a file name, and additionally it can be associated with a `StringBuffer`[6] – this is the actual text of the module, for use in cases where the text of the module does not reside on disk, which is the case in Visual Haskell when a file has not been saved since the last edit, for example. When using a `StringBuffer` to represent the file's contents, GHC also needs to know the `ClockTime` when the file was last edited, so it can decide whether re-compilation or re-checking is required.

In Visual Haskell, the set of `Targets` will typically contain all the Haskell source files in the Project. Having set our `Targets`, we can proceed to load the code. The function `load` is used to compile modules; whether the modules are compiled to object code, compiled to bytecode or just typechecked is dependent on the `GhcMode` parameter to `newSession`. The `LoadHowMuch` argument to `load` determines which portion of the module dependency graph is loaded; its options are self-explanatory.

Any loaded module can be inspected using `getModuleInfo`, which returns a `ModuleInfo`. The `ModuleInfo` can be interrogated to find all kinds of properties about the module: the entities it defines, the names it exports, the instances it provides, and so on.

An alternative to `load` is `checkModule`. The `checkModule` interface behaves almost identically to `load` with the `LoadUpTo` option, except that `checkModule` returns a `CheckedModule` structure if the compilation was successful. A `CheckedModule` contains up to three versions of the compiler's abstract syntax tree: after parsing, after renaming[7], and after typechecking. Additionally, `checkModule` returns a `ModuleInfo` structure for the checked module.

The main reason for separating `checkModule` from `load` is that keeping around the abstract syntax trees from the various front-end phases constitutes a space leak, so we don't want to do this during the normal course of a `load`.

Both `load` and `checkModule` take a function of type

```
Messages -> IO()
```

as an argument; this is a callback invoked for both error messages and warnings discovered during the compilation or checking process. In the event of compilation errors, the callback will be invoked, and `load` or `checkModule` will return a result indicating failure.

3.3.2 Using the GHC API in the Visual Haskell editor

The basis of the rest of the editor features is that the `ParseSource` entry point to our plugin invokes `checkModule` for the current module. Firstly, however, it updates the `Target` for this module in the current `Session` (more about how we keep track of the `Session` later) to contain a `StringBuffer` representing the current text of the module.

Visual display of error messages.

The error-message callback that we pass to `checkModule` is a function that in turn invokes methods on the `IParseSink` object passed to `ParseSource`: this is how error messages are reported back to Babel. Babel handles the visual underlining of the erroneous code and the addition of the error task to the task list.

Visual Studio requires an exact source *span* for an error message – the line and column number of both the start and end points of the syntactic entity containing the error, so that the error can be underlined in the editor. Previous versions of GHC, however, only kept approximate source-location information in the form of line numbers attached to selected points in the abstract syntax tree, primarily declaration sites. We had to modify GHC such that each element of the abstract syntax tree is explicitly annotated with the span of the text from which it was derived. This turned out to be a great deal of work, but ultimately worthwhile.

Pop-up type information.

The "quick info" feature, where the type of an identifier is displayed in a pop-up window when the mouse hovers over it, is implemented as follows. If typechecking is successful, the call to `checkModule` returns the abstract syntax tree for the module generated by the typechecker – this version of the abstract syntax has two important properties:

- It is decorated with types. In particular, all identifiers have types attached.
- It has been translated to include explicit type abstraction and application, and explicit dictionary passing.

The first property is the most important from our perspective. When the Visual Haskell user hovers the mouse over an identifier, Babel calls a method in our plugin passing the source location of the mouse, and we have to return the text for the pop-up window, if any. Finding the type is a matter of finding the identifier in the typed abstract syntax, and extracting its type. We do this by searching the abstract syntax by location; the search is linear in the depth of the tree, because each node is decorated with a span, so we can ignore subtrees whose span does not contain the location we are interested in.

In addition to displaying the type of an identifier, the quick-info feature will also display information about type names, class names, and module names in `import` statements. However, the typechecked abstract syntax tree contains only the function definitions from the original source code: type signatures, `class` declarations, `instance` declarations, and `import` statements have all been converted into internal representations. This is the reason that `checkModule` also returns the abstract syntax from earlier phases in the compilers front-end. To summarise the different forms of the abstract syntax:

- **Parsed**: abstract syntax translated exactly from the source code. Identifiers are strings.
- **Renamed**: export list and `import` statements removed. Identifiers are resolved to entities, and contain defining locations.
- **Typechecked**: type signatures, `class` and `instance` declarations are removed, and the abstract syntax contains type annotations and dictionary passing. Identifiers are annotated with types.

These three versions of the abstract syntax tree can't be easily combined, because they have different types – the abstract syntax type is abstracted over the type of identifiers.

[6] `StringBuffer` is a type used internally by GHC; it represents a flat array of bytes.

[7] GHC's term for the resolving of names to entities.

So our quick info feature has to additionally search the renamed and parsed versions of the abstract syntax tree. If we find a type or class name under the cursor, then we can find the definition of that type or class by interrogating the GHC API, and display its definition in the pop-up window.

Go to definition.

This feature is implemented in a similar way to quick info. However, we did have to modify Babel to add support for the "go to definition" command. When it receives the "go to definition" command, Babel supplies our plugin with a source location, and we have to return the filename and source location of the definition site, if any.

Fortunately, identifiers in the abstract syntax tree (at least the renamed and typechecked versions of the abstract syntax) contain information about the defining location of the identifier – this applies to all identifiers, including type names and local variables. So again, finding the information is a matter of searching the abstract syntax tree by location, and extracting the information from the identifier at the required location, if any.

The fact that we can "go to definition" for local variables in addition to top-level functions and types is a feature unique to Haskell amongst the Visual Studio languages. Additionally, our "go to definition" properly respects the scoping rules of the language. However, using this feature does require that the module is at least correctly scoped – this is a restriction that we hope to lift to some extent in the future (see Section 6).

The drop-down definition list.

The drop-down box displays the list of top-level definitions in the current source file, and allows selecting an item to jump directly to that definition.

To implement this feature, we extended Babel to manage the mechanics of the drop-down list itself. We added the `GetObjectBrowserList` method on the `IScope` object returned by `ParseSource` which is used from Babel to update the current list of top-level definitions after each successful call to `ParseSource`.

Word completion.

The full set of names in scope at the top-level of the module is known after a successful `checkModule` on a source file; it is available from the `ModuleInfo`. When a word completion is requested by the user, Babel interrogates the `IScope` from the last successful check for the list of names in scope, and the plugin returns the list of names obtained from the `ModuleInfo`.

We always use the top-level scope rather than attempting to take into account locally-bound names based on the location of the cursor in the source file. The reason is that at the point word completion is required, the source file is unlikely to be in a syntactically-correct state, so discovering the correct scope will rarely be possible. In languages like C++ and C♯, the scope only depends on the source code *before* the current point in the source file, so a correct scope can be calculated even if there are parse errors after the cursor position. This isn't the case in Haskell, where declarations are in scope over the entire source file. However, it may still be possible to improve on the "top-level-names-only" scheme to some extent, but we leave this for future work.

4. Implementation of the Project and ClassBrowser features

In Visual Haskell two different windows are used in order to express the contents of the project: all Haskell and C files and any other documents and scripts are accessible from the "Solution Explorer" (Figure 7) while the hierarchical namespace together with all Haskell definitions are visible in the "Class View" (Figure 10). The hierarchy of modules doesn't necessary match the directory hierarchy in the filesystems. The project may have a flat module namespace but nevertheless the user may want to separate its source files in different directories. The environment keeps a list of directories in which to look for Haskell files and automatically builds the hierarchy namespace which the project is expected to have. Cabal receives the same list and is able to compile the project properly[8].

There is a significant difference in the way in which these two views are generated. The information in the Solution Explorer is populated from Cabal's package description file and after that it remains static, at least until the user adds or removes any item from the project. At the same time, the Class View content is dynamic and is updated each time the user makes any changes to any Haskell module. The Class View, the Solution Explorer and the editor cooperate to achieve this functionality. After each modification in any source file, the editor calls `checkModule` to parse and typecheck the content. If `checkModule` succeeds, then the gathered information is used to update the declarations list and Class View. In this sense the Class View contains information which is collected and transformed from both the Solution Explorer and the editor.

In terms of the GHC API we described in the previous section, a Haskell project has a single `Session`, which is populated with a `Target` for each of the Haskell source files in the project. To populate the initial Class View when loading a project, a full `load` is performed on the `Session`. This can take a minute or two for a large project, but for our released version of Visual Haskell we plan to make this happen in the background (Concurrent Haskell is tremendously convenient for such tasks).

Project support in Visual Haskell is implemented by communicating directly with the Visual Studio COM APIs, which are rather large and complex – in Section 5 we recall some of our war stories.

4.1 Projects and Cabal

As we've mentioned earlier, the project support in Visual Haskell is based heavily on the Cabal library. Cabal provides a complete Haskell build system, which we use in Visual Haskell when the user requests to build the project.

Meta-data about the project is kept in a `.cabal` file (an example was given in Figure 11). When the project is being edited inside Visual Haskell, the `.cabal` file is under the control of the Visual Haskell environment: the contents can be manipulated through the controls provided.

In a Cabal library, each module can be either "exposed", which means that it is available to a client of the library, or "hidden", which means that a client of the library is prevented from importing the module. We expose this option to the Visual Haskell programmer via an option on the property page for each module in the project, and the exposed/hidden status of each module is also indicated via an icon.

A Cabal package typically has a `Setup.lhs` file, which is a (usually tiny) Haskell script by which the Cabal build system can be invoked from the command line. In Visual Haskell, we make this file visible and editable via the Solution Explorer, but we have to be careful to ensure the editor is working with a separate GHC

[8] Cabal was restricted to have only one source directory but we have extended it. This feature is available in the development version of Cabal.

Session when editing this file, because it is not part of the project proper.

4.2 Multiple projects

Visual Studio has a concept of a *solution*, which is essentially a collection of projects, with dependencies between the projects. Typically each project builds a single executable or library. Building a solution consists of building each of the component projects in the correct order.

Visual Haskell fully supports solutions; Haskell projects can coexist with project from other languages in a solution. When there are multiple Haskell projects in the solution, each one is given its own Session in the GHC API. However, there is only a single Visual Haskell plugin running, and a single instance of the GHC library, so all these Sessions are managed in a single heap. Ideally, the Sessions would be able to share a lot of state: the interfaces for common libraries, for example. We have not implemented this yet (the fact that different projects may depend on different external libraries makes it non-trivial).

A good example of a solution is the Visual Haskell plugin itself, which consists of three C++ projects (our modified Babel, a library of utilities on which Babel depends, and a small library of user-interface utilities), the Haskell plugin code itself, and a Windows Installer project for building the installer.

5. Writing COM components in Haskell

Microsoft's Common Object Module (COM) is a language-independent standard and set of APIs for communicating between software components. It is widely used on the Windows platform[9], and in particular Visual Studio uses COM as its sole interaction substrate for communicating with plugins.

The size of the Visual Studio APIs is daunting. The IDL code, that is, just the specification of the interfaces that Visual Studio exposes, runs to 30,000 lines. Fortunately for many of the editor features we were able to build on Babel which abstracts many of these interfaces down to a manageable core. However, for the project support, we had to talk directly to Visual Studio.

A tool for generating interface code from the IDL specification is essential. The Haskell tool in this space is H/Direct [7, 8], which does exactly what we want: it reads IDL specifications and produces the low-level marshaling code that enables high-level Haskell code to consume and offer COM interfaces in a convenient and type-safe way. H/Direct also provides a library of code providing basic COM functionality: COM datatypes, marshaling primitives, and so on.

Sadly our experience with H/Direct has not been altogether positive. Many implementation bugs were discovered along the way, and to this date our low-level COM interface code is partially generated by H/Direct and partially edited by hand, to work around bugs and limitations in H/Direct.

5.1 Reference counters and finalizers

COM objects are explicitly reference counted. Two methods are used to manipulate the reference count on an object: AddRef() and Release(), and these methods must be implemented for every object. Release() is expected to deallocate the object if the reference count drops to zero.

[9] Although in the future it may be increasingly supplanted by .NET.

Object pointers in H/Direct are represented by the ForeignPtr type. A ForeignPtr has a finalizer – an arbitrary piece of code which runs when the object is found to be no longer referenced by the garbage collector. Finalizers are used by H/Direct to manage reference counts: the idea is that every in-coming object pointer is converted into a ForeignPtr, and its reference count increased by a call to AddRef(). When the garbage collector detects that the object pointer is no longer used, the finalizer calls Release().

This approach makes things a lot simpler for the Haskell programmer: he doesn't have to worry about explicit reference counting (a common source of bugs in C++ COM code). However the approach suffers from three problems:

- Performance: this requires incrementing the reference count (with a function call) for every incoming interface pointer, just in case it is stored in a Haskell data structure. This is a significant overhead for a simple function call.
- Space leaks: if the finalizer does not run promptly, the object cannot be freed, resulting in a space leak. What's worse is that this artificial space leak can lead to whole libraries being retained in memory longer than necessary, because a library is unloaded when there are no longer any objects managed by the library still in use.
- More importantly, this use of finalizers just doesn't work in a multithreaded setting. Many objects are written to be single threaded; that is, they assume that certain method calls (including the reference counting calls AddRef() and Release()) are called from a single thread. When the Haskell runtime invokes a finalizer, it may invoke it in a different thread than the one in which the finalizer was created; in fact, the original thread that created the finalizer may be long gone, because it was probably a thread that briefly called into Haskell to invoke a COM method.

The upshot is that we can't use finalizers to call Release(), because the finalizer might be invoked in the wrong thread. This problem actually lead to a lot of instability in our early versions of the Visual Haskell plugin.

To fix this problem, we modified H/Direct and its libraries to represent interfaces by plain Ptrs, and we modified our Haskell plugin code to do explicit reference-count management of COM pointers. Managing reference counts properly is tricky and error-prone, but we found that the number of places which had to be modified was relatively small (most calls don't store interface pointers, and therefore don't need to alter reference counts). And because Haskell is such a great language for expressing abstractions, we were able to reduce the overhead and the potential for mistakes by using a few well-chosen combinators. One of the common cases that requires reference count manipulation is the QueryInterface() call, which returns an interface pointer that has to be Release()'d. We wrapped QueryInterface() in a combinator:

```
withQueryInterface
        :: IID (IUnknown b)
        -> IUnknown a
        -> (IUnknown b -> IO c)
        -> IO c
```

the idea being that withQueryInterface automatically calls Release() when the IO action has completed, and it can do this in an exception-safe way too: if the IO action raises an exception,

the interface will still be `Release()`'d, eliminating another cause of reference-count leakage.

This change to explicit reference counting improved the performance of Visual Haskell.

5.2 GUIDs

A GUID is a Globally Unique IDentifier, namely a 256-bit immutable value used to uniquely identify classes and interfaces in COM. H/Direct currently represents GUIDs with a `ForeignPtr` in the same way as interface pointers, but this leads to problems. GUIDs aren't explicitly reference counted, and by convention if a callee needs to keep a GUID passed to it, it should make a copy of the GUID rather than retaining a pointer to it. H/Direct didn't follow this convention, which has been a source of errors in our interface code.

Fortunately the solution is simple: we should represent GUIDs by immutable objects in Haskell, and create a copy of any GUID that is passed into Haskell via a foreign call. Copying the GUID imposes a small overhead, but it is less that H/Direct's current implementation which involves a fully-fledged `ForeignPtr` with a finalizer. We also need to be able to pass GUIDs to foreign calls from Haskell - fortunately GHC has immutable array types which support this.

5.3 Client vs. Server interface code

The IDL compiler in HDirect has an option whether to generate client side or server side code. In a real application it is a common to have both client side and server side implementations for one interface. The compiler can generate two different files but then many declarations will be duplicated. In Visual Haskell we have manually changed these files to share all common definitions. H/Direct should really have an option that generates both server and client side code, but ideally we would have the facility to select whether we wanted client or server code or both on an interface-by-interface basis to avoid generating large amounts of unnecessary interface code.

5.4 Object state

We often found that an object implemented in Haskell needs to gain access to its own interface pointers from inside a method call. The only state available to methods is the object state, which can be a value of any type, but does not necessarily contain a pointer to the object itself. Making the object state contain a pointer to the object itself requires recursion when creating the object. One way to write it is to use the `mdo` extension:

```
createFoo = mdo
  let state = FooState foo
  foo <- createComInstance state
                           (relaseFoo state)
  return foo
```

We found this pattern to be quite common in our interface code, so it would be natural to instead provide a method for object creation that allows creation of the state from the object pointer.

5.5 Performance

The COM component uses multithreaded RTS but this isn't without extra cost. The FFI calls are much slower than in the single threaded RTS. The frequent calls can force garbage collections which slows down the things even more. In particular the `AddRef`, `Release` and `QueryInterface` methods from the `IUnknown` interface are called quite frequently. The `QueryInterface` method takes GUID as argument which adds extra cost. The overall overhead can be significant for some tasks. We have found this in our syntax colouring component. It has to work in real time while the user is typing any text in the editor, so the foreign calls overhead in this case is significant. Fortunately in COM it is expected that `IUnknown`, `IDispatch`, `IClassFactory`, `IConnectionPoint` and `IConnectionPointContainer` interfaces always have a standard predefined behaviour. This is used in HDirect and it provides predefined implementations in Haskell. The performance will be much better if they were in C, because the standard operations will be performed without any FFI calls.

5.6 Sub-classed interfaces

H/Direct currently can't generate code for an interface that does not derive directly from `IUnknown` or `IDispatch`, as might be the case if an interface sub-classes an existing interface. This happens regularly in the Visual Studio COM APIs.

Our workaround involved editing the H/Direct-generated code directly to implement the derived interfaces, but this is clearly not a long-term solution.

5.7 Further reflections on H/Direct

H/Direct is an old tool; it was designed together with the first Haskell FFI (Foreign Function Interface) definition, but since then the FFI has progressed significantly and is now standardised along with a collection of standard marshaling libraries. H/Direct's marshaling libraries duplicate functionality is now found in the standard libraries, and this makes it difficult to mix existing marshaling code using the standard FFI libraries with H/Direct.

Arguably, COM is an old technology and we should be concentrating on .NET. Indeed, Microsoft has started to provide .NET interfaces for extending Visual Studio; however, we believe that for our purposes going via a .NET interface would impose a significant performance penalty, because every call would go from native code through the .NET runtime and back into native code again.

For us to proceed with Visual Haskell and to keep our code maintainable, we need a reliable tool that can generate the interface code we need, direct from the original IDL source that Microsoft provides. The only solution may be to start from scratch; H/Direct being too large to modify. We certainly don't need to duplicate all the functionality that H/Direct provides: it does a lot more than just generate COM interface code. However, the experiences learned from using H/Direct in this setting have will no doubt be useful in designing future tools.

6. Conclusions and Future Work

Visual Haskell fulfills the design aim that we have targeted: it provides syntax colouring, interactive error checking, quick info, word completion, source code browser and projects. The project is self-hosting: it can be developed and built inside Visual Studio itself. We expect to make the first public release shortly, and we anticipate making further improvements based on user feedback.

From an overall perspective, the implementation of this environment seems like 90% infrastructure and 10% real features. However, now that we have the basis of the plugin connecting Visual

Studio with GHC, there are plenty of fun bits still to come: we can implement more useful programmer features based on the wealth of information that GHC collects about the source code.

There are plenty of editing features still to be implemented. Among the more interesting features we have in mind are:

- Displaying the inferred type of a selected subexpression.
- Refactoring: incorporating the HaRe project [12] would allow refactorings to be performed directly in the editor.
- Automatic outlining: collapsing code to an "outline", such as the first line of each definition, where individual collapsed chunks can be expanded or collapsed.
- GHCi integration: a separate GHCi window in the environment for interactively evaluating expressions or running tests in the context of the project.

Some other future work plans we have are:

- Most of the editor features (with the exception of word completion) require the source code to be at least syntactically correct, and in some cases type correct, before they work. Lifting this restriction as much as possible is something we plan to tackle in the future.
- The documentation for GHC and the standard libraries is already integrated with Visual Studio. Unfortunately Visual Haskell can't find the documentation about the current identifier in the source code automatically. The user has to open the documentation browser and has to look up manually. Cabal provides built in support for Haddock which we would like to integrate with Visual Haskell.
- Visual Studio provides a concept called "configurations", which essentially corresponds to multiple sets of options under which the project can be built. For example, in the "Debug" configuration, we might compile with optimisation off and with assertions turned on, whereas in the "Release" configuration we would compile with optimisation turned on. Visual Haskell currently supports only a single configuration, but we hope to be able to support multiple configurations in the future, perhaps by extending the syntax of the `.cabal` file.
- The Hackage project aims to provide easy installation of external packages; integrating this into the Visual Studio environment would improve the experience even further.
- Most of the integrated languages provide fully featured source level debugger. Unfortunately the debugging of lazy functional language is still an open research problem. Even that providing at least limited support for debugging is better than nothing. The environment already has C code debugger. It is common practice to have projects with mixed Haskell and C code so we can reuse the C debugger.

Acknowledgments

We would like to thank Microsoft Research Ltd. Cambridge, for supporting an internship for Krasimir Angelov during which much of the implementation work on Visual Haskell was done, and this paper was written.

We would like to thank the Cabal designers and developers: Isaac Jones, Ross Paterson, Simon Peyton Jones, and Malcolm Wallace, for accommodating the requirements of Visual Haskell in their design.

Many thanks to Sigbjorn Finne for providing support for H/Direct, and to Daan Leijen for developing Babel.

References

[1] The Glasgow Haskell Compiler. `http://www.haskell.org/ghc`.

[2] Haste (haskell turbo edit). `http://haste.dyndns.org:8080/index.php`.

[3] Kdevelop. `http://www.kdevelop.org`.

[4] Vim. `http://www.vim.org`.

[5] J. Axelsson. HaskellDoc. `http://www.ida.liu.se/~jakax/haskell.html`.

[6] M. Chakravarty. IDoc - a no-frills Haskell interface documentation system. `http://www.cse.unsw.edu.au/~chak/haskell/idoc`.

[7] S. Finne, D. Leijen, E. Meijer, and S. Peyton Jones. H/Direct: a binary foreign language interface for Haskell. In *ACM SIGPLAN International Conference on Functional Programming (ICFP'98)*, volume 34(1) of *ACM SIGPLAN Notices*, pages 153–162. Baltimore, 1998.

[8] S. Finne, D. Leijen, E. Meijer, and S. Peyton Jones. Calling Hell from Heaven and Heaven from Hell. In *ACM SIGPLAN International Conference on Functional Programming (ICFP'99)*, pages 114–125, Paris, Sept. 1999.

[9] L. Frenzel, A. Granicz, A. de Araujo Formiga, and V. Rocha. Haskell support for Eclipse. `http://eclipsefp.sourceforge.net`.

[10] A. Groesslinger. Hdoc. `http://www.fmi.uni-passau.de/~groessli/hdoc`.

[11] I. Jones. The Haskell Cabal, a common architecture for building applications and libraries. Submitted to the Haskell Workshop 2005.

[12] H. Li, S. Thompson, and C. Reinke. The Haskell refactorer, HaRe, and its API. In *Fifth workshop on Language Descriptions, Tools and Applications*. ACM Press, April 2005.

[13] S. Marlow. Alex: A lexical analyser generator for Haskell. `http://www.haskell.org/alex`.

[14] S. Marlow. Haddock, a Haskell documentation tool. In *Proceedings of the ACM SIGPLAN workshop on Haskell*, Pittsburgh Pennsylvania, USA, October 2002. ACM Press.

[15] S. Marlow, S. P. Jones, and W. Thaller. Extending the Haskell foreign function interface with concurrency. In *Proceedings of the ACM SIGPLAN workshop on Haskell*, pages 57–68, Snowbird, Utah, USA, September 2004.

[16] G. E. Moss, T. Thorn, and S. Marlow. Haskell mode for Emacs. `http://www.haskell.org/haskell-mode`.

[17] M. Neubauer and P. Thiemann. Demonstration abstract: Haskell type browser. In *Proceedings of the ACM SIGPLAN workshop on Haskell*, Snowbird, Utah, USA, September 2004.

[18] D. Rogerson. *Inside COM. Microsoft's Component Object Model*. Microsoft Press, 1997.

[19] J. Svensson, D. Coutts, and S. Kurtzberg. hide. `http://www.dtek.chalmers.se/~d99josve/hide`.

[20] R.-J. van Haaften. Jcreator with Haskell support. `http://www.students.cs.uu.nl/people/rjchaaft/JCreator/`.