Tool Support for Refactoring Functional Programs

Huiqing Li
Computing Laboratory, University of Kent, UK
H.Li@kent.ac.uk

Simon Thompson
Computing Laboratory, University of Kent, UK
S.J.Thompson@kent.ac.uk

Abstract

We demonstrate the Haskell Refactorer, HaRe, and the Erlang Refactorer, Wrangler, as examples of fully-functional refactoring tools for functional programming languages. HaRe and Wrangler are designed to handle multi-module projects in complete languages: Haskell 98 and Erlang/OTP. They are embedded in Emacs (and gVim) and respect programmer layout styles.

In discussing the construction of HaRe and Wrangler, we comment on the different challenges presented by Haskell and Erlang due to their differences in syntax, semantics and pragmatics. In particular, we examine the sorts of analysis that underlie our systems.

Finally, drawing on our experience, we examine features common to functional refactorings, and contrast these with refactoring in the object-oriented domain.

Categories and Subject Descriptors D.2.3 [*SOFTWARE ENGINEERING*]: Coding Tools and Techniques; D.2.6 []: Programming Environments; D.2.7 []: Distribution, Maintenance, and Enhancement; D.3.2 [*PROGRAMMING LANGUAGES*]: Language Classifications—Applicative (functional) languages; Concurrent, distributed, and parallel languages; D.3.4 []: Processors

General Terms Languages, Design

Keywords Haskell, Erlang, refactoring, HaRe, Wrangler, program analysis, program transformation, static/dynamic semantics.

1. Introduction

Refactoring (Fowler et al. 1999) is the process of improving the design of a program without changing its external behaviour. Behaviour preservation guarantees that refactoring does not introduce (or remove) any bugs. Separating general software updates into functionality changes and refactorings has well-known benefits. While it is possible to refactor a program by hand, tool support is considered invaluable as it is more reliable and allows refactorings to be done (and undone) easily. Refactoring tools can ensure the validity of refactoring steps by automating both the checking of the conditions for the refactoring and the application of the refactoring itself, thus making refactoring less painful and less error-prone.

Our project 'Refactoring Functional Programs' (Refactor-fp), has developed the Haskell Refactorer, HaRe (Li et al. 2003), providing support for refactoring Haskell (Peyton Jones 2003) programs. HaRe covers the full Haskell 98 standard language, and is integrated with the two most popular development environments for Haskell programs: gVim and (X)Emacs. Because layout style tends to be idiomatic and personal especially when a standard layout is not enforced by the program editor, HaRe preserves the comments and layout of the refactored programs as much as possible.

HaRe is itself implemented in Haskell. The current (third) release of HaRe supports 24 refactorings, and also exposes an API (Li et al. 2005) for defining refactorings or more general program transformations. The refactorings supported by HaRe fall into three categories: structural refactorings which concern the name and scope of the entities defined in a program and the structure of definitions; module refactorings which concern the imports/exports of modules, and the relocation of definitions among modules; and data-oriented refactorings which concern the data type definitions. The ongoing work with HaRe currently focuses on data-related refactorings.

Following the 'Refactoring Functional Programs' project, we are developing Wrangler (Li and Thompson 2006; Li et al. 2006), a tool for refactoring Erlang/OTP (Armstrong et al. 1996; Armstrong 2007) programs. The current (second) release of Wrangler works with the complete Erlang/OTP language, and supports a few structural refactorings, such as *rename an identifier*, *generalise a function definition*, *function extraction*, *move a function definition between modules*, etc, and functionalities for *duplication code detection*. We are currently at the middle stage of this project, and Wrangler is still under active development.

Building a refactoring tool for Erlang allows us to continue our investigation of the application of refactoring techniques to the functional programming paradigm. Both Haskell and Erlang are general-purpose functional programming languages, but they also have many differences. Haskell is a lazy, statically typed, purely functional programming language featuring higher-order functions, polymorphism, type classes, monadic effects, and program layout sensitiveness. Erlang is a strict, dynamically typed functional programming language with built-in support for concurrency, communication, distribution, and fault-tolerance. The differences in syntax, semantics and pragmatics of Haskell and Erlang impose difference challenges, and result in different implementation strategies and techniques.

In this paper, we discuss the construction of HaRe and Wrangler, and comment on the challenges we had to solve. In particular, we examine the sorts of analysis that underline our systems. Finally, drawing on our experience, we examine features common to functional refactorings, and contrast these with refactoring in the object-oriented domain.

2. An Overview of HaRe and Wrangler

Both HaRe and Wrangler support interactive refactoring of multi-module programs. HaRe is integrated with the two most commonly used program editors for Haskell: (X)Emacs and gVim, while Wrangler is integrated with Emacs, which is the tool of choice for most Erlang programmers. Currently HaRe supports more refac-

PEPM'08, January 7–8, 2008, San Francisco, California, USA.

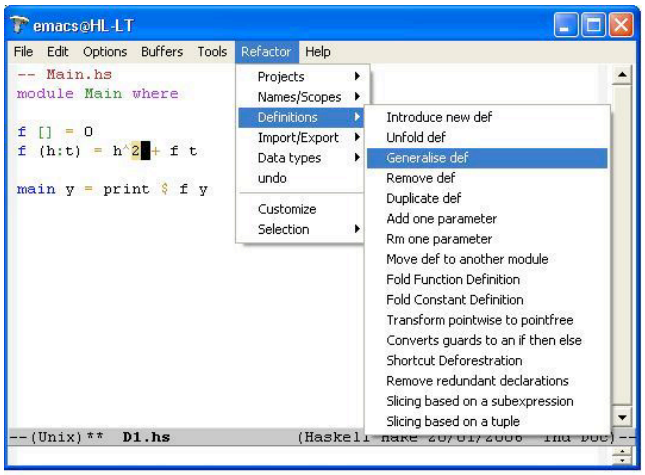

Figure 1. A snapshot of HaRe

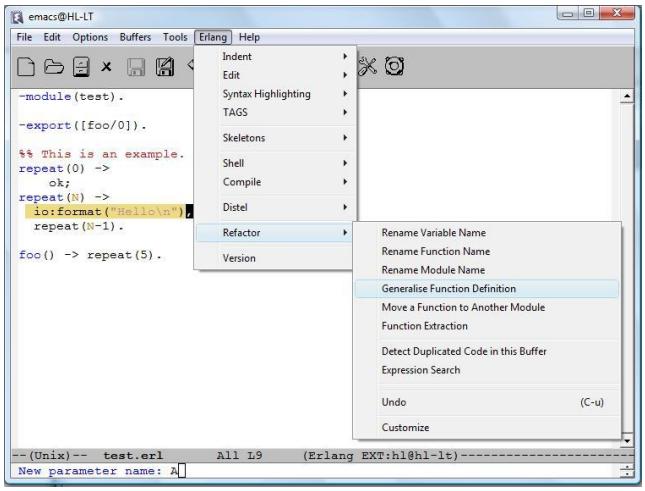

Figure 2. A snapshot of Wrangler

torings than Wrangler does, however more refactorings are being implemented for Wrangler. Snapshots of HaRe and Wrangler embedded in the Emacs environment are shown in Figures 1 and 2.

While the underlying implementation techniques are different, HaRe and Wrangler have very similar user interfaces. To perform a refactoring with HaRe or Wrangler, the focus of refactoring interest has to be selected in the editor first. For instance, an identifier is selected by placing the cursor at any of its occurrences; an expression is selected by highlighting it with the cursor. Next the user chooses the refactoring command from the refactor menu, and inputs the parameters(s) in the mini-buffer if required. Then the refactorer checks that the focused item is suitable for the refactoring selected, that the parameters are valid, and that the refactoring's *side-conditions* (or *pre-conditions*) are satisfied.

If all these checks are successful, the refactorer will perform the refactoring, and update the buffer with the new program source, otherwise it will give an error message, and abort the refactoring with the program unchanged. *Undo* is supported by both HaRe and Wrangler. Applying *undo* once reverts the program back to the state right before the last refactoring in the refactoring history was performed; *undo* can be applied multiple times until the refactoring history is empty. With the current implementation of HaRe and Wrangler, the refactoring *undo* does not interact with the editor-side *undo*/*redo*, therefore undoing a refactoring will lose the editing done after this refactoring. Tighter coupling of tool and editor would support the integration of the two *undo* mechanisms.

All the refactorings implemented in HaRe and Wrangler are module-aware. For a refactoring that could possibly change a module's interface, it might have an effect in not only the module where the refactoring is initiated, but also those modules that import this module directly or indirectly. To ensure the correctness of transformation, the refactorer needs to know which modules are in the scope of the current programming project. Because of the different underlying infrastructure, HaRe and Wrangler use different ways to specify the project boundary. With HaRe, a project should be created before doing any refactorings. To create a project, first start a new project with only one module (usually the `Main` module) in it, then use HaRe's *chase* functionality to include into the project those modules on which the current module depends. With Wrangler, the user takes the responsibility to customise the refactorer with the lists of Erlang source directories belonging to the project under consideration.

We return to the snapshot of Wrangler in Figure 2, which shows a particular refactoring scenario: the user has selected the expression `io:format("Hello\n")` in the definition of `repeat/1`, has chosen the *Generalise Function Definition* command from the *Refactor* menu, and is just entering a new parameter name `A` in the mini-buffer. Then, the user would press the *Enter* key to perform the refactoring. After side-condition checking and program transformation, the result of this refactoring is shown in Figure 3: the new parameter `A` has been added to the enclosing function definition `repeat/1`, which now becomes `repeat/2`; the highlighted expression has been replaced with `A()`; and at the call-site of the generalised function, the selected expression, wrapped in a *fun*-expression, is now supplied to the function call as its first actual parameter. We enclose the selected expression within a function closure because of its side-effect, so as ensure that the expression is evaluated at the proper points.

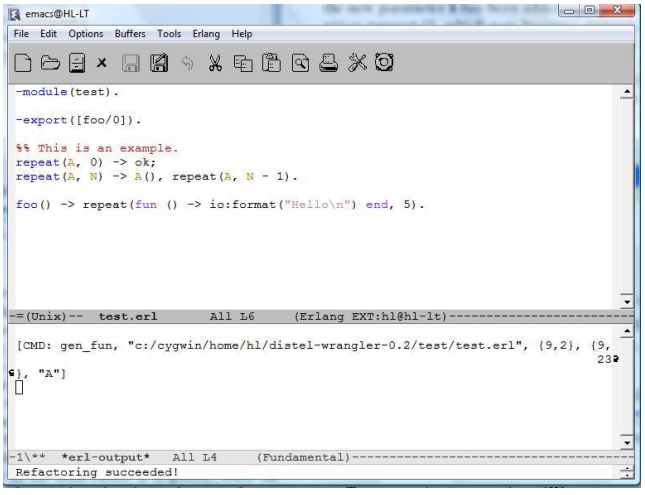

Figure 3. A snapshot of Wrangler showing the result of generalising a function definition

The current implementation of Wrangler pretty-prints the refactored program source according to the standard layout rules; it therefore does not preserve the program's original layout unless the original layout is itself the result of pretty-printing. This limitation is to be addressed by adding layout information to the abstract syntax tree (AST), which is the internal representation of programs used by the Wrangler system, and by using this information

to guide the pretty-printing process. The current Wrangler release preserves the program's comments as much as possible.

Unlike Wrangler, HaRe preserves both the comments and the program layout of the refactored program, due to the fact that there is no standard Haskell layout which is accepted by most Haskell users, and Haskell users tend to have their own personal layout style.

3. Implementation

This section discusses the construction of HaRe and Wrangler, comments on the challenges presented by Haskell and Erlang due to their differences in language design and programming idioms, and address how they are handled by the two systems in section 3.3.

3.1 Semantics and Transformation

Each refactoring comes with a set of *side-conditions* (or *pre-conditions*), which embody when a refactoring can be applied to a program without changing its meaning. In order to preserve the functionality of a program, refactorings require awareness of various aspects of the semantics of the program. The following semantic information is needed by either HaRe or Wrangler, or both of them.

- **The binding structure of the program.** Binding structure refers to the association of uses of identifiers with their definitions in a program. An identifier could be a variable name, a function name, or a module name. The general principle is that a refactoring should not disrupt the existing binding structure.

  Both Haskell and Erlang allow static scoping of variables, but Erlang has more complex binding rules for variables. In Erlang, a pattern can contain both binding and applied occurrences of variables; furthermore, a variable may have more than one binding occurrence in a `case/receive` expression.

  Deciding the binding structure of function names, or finding the call-sites of a function, is in general more challenging for Erlang programs for the following reasons.

  - Firstly, a function name in Erlang is an atom literal, but an atom name could also be a module name, a process name or just a literal. This makes it difficult to see whether an atom refers to a function name or not.
  - Secondly, Erlang allows atoms to be created at run time, thus makes it possible to compose function names dynamically, which again makes finding the call-sites of a function impossible at compile time.
  - Thirdly, Erlang allows meta-applications using the built-in functions, such as `apply/3`, `spawn/3` and their variants, by passing a function name and the function's actual parameters as parameters.
  - Finally, function names or even function definitions can be passed between Erlang processes as Erlang terms.
- **Module structure.** Both Haskell and Erlang have a module system. A module-level call graph is needed when a refactoring affects the interface of a module. Overall, Haskell's module system is more complicated than Erlang's relatively simple system, and a refactoring process can be made complicated by some features of the Haskell module system, such as the transitive exporting of entities, the lack of mechanisms for excluding entities using *hiding* in an export list, etc.
- **Type information.** Both Haskell and Erlang are typed programming languages, however Haskell features static typing whereas Erlang features dynamic typing. Type information is needed by some Haskell refactorings in order to succeed, especially when the interface of a function definition which has a type signature declared has been changed. Erlang is a weakly typed programming language. For most Erlang refactorings, type information is not needed, though sometimes type information can help.
- **Side-effect information.** Unlike Haskell which is a pure lazy functional language, Erlang is a strict functional language with side-effects, and mutable stuff (message sends/receives and state-dependent responses) plays an important part in most large programs. When the evaluation order or process context of an expression is going to be changed by a refactoring, Wrangler needs to know whether the expression itself has side-effects or not, and whether the expression needs to get access to any state-dependent information.

  Note that variable assignment in Erlang does not cause side-effects, as Erlang is a single-assignment language. Single-assignment of variables frees us from the complex control flow or data dependency analyses which are generally critical issues for programming languages with side-effects.
- **Comment and Layout information.** Comment and layout information is needed by HaRe to preserve the original program's layout and comments as much as possible. For Wrangler, only comment information is currently needed, however layout information is also needed if layout is to be preserved.

3.2 Tool Support for Refactorings

A refactoring tool needs to get access to both the syntactic and static semantic information of the program under refactoring. Given a refactoring command, most static analysis-based refactoring tools (or engines) go through the following process, which is also illustrated in Figure 4, although detailed implementation techniques might be different.

First transform the program source to some internal representation, such as an abstract syntax tree (AST); then analyse the program to extract the static semantic information needed by the refactoring under consideration, such as the binding structure of the program, type information and so forth.

After that, program analysis is carried out based on the internal representation of the program and the static semantic information to validate the *side-conditions* of the refactoring. If the *side-conditions* are not satisfied, the refactoring process stops and the original program is unchanged, otherwise the internal representation is transformed according to the transformation rules of the refactoring. Some interaction between the refactorer and the user might be or helpful during *side-condition* checking and/or program transformation.

Finally, the transformed representation of the program needs to be presented to the programmer in program source form, with comments, and even the original program appearance, preserved as much as possible.

Almost all the available refactoring tools are embedded within one or more programming environments, therefore the integration of a refactoring tool with the intended programming toolkit(s) is an unavoidable part when tool support for refactorings is concerned. Another unavoidable issue for a refactoring tool to be useful in practice is the support for undoing refactorings. Being able to *undo*/*redo* a refactoring quickly, people are more willing to explore different refactoring ideas. The underlying implementation mechanism for both the integration with programming environments and the supporting for *undo* could vary significantly from system to system.

Unsurprisingly, this analysis applies to the implementation of both HaRe and Wrangler.

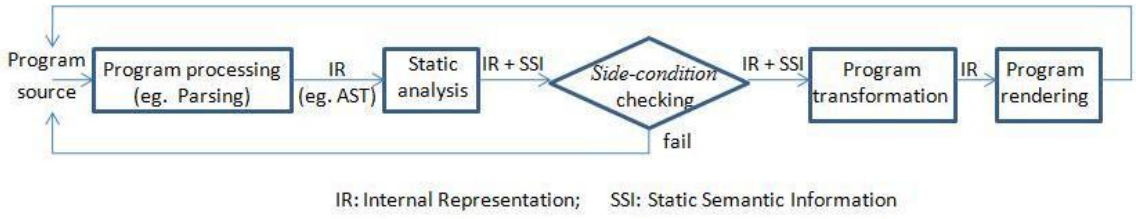

Figure 4. A General Framework of Refactoring Engines.

3.3 Implementation Techniques

Different techniques have been used in the implementation of HaRe and Wrangler. HaRe is implemented in Haskell using the Programatica (PacSoft) frontend (including lexer, parser and module analysis) for Haskell, and the Strafunski (Lämmel and Visser 2001) library for generic AST traversals. For efficiency reason, we used the type checker from GHC (GHC), instead of Programatica, to derive type information. In HaRe, we use both AST and token stream as the internal representation of source code. Layout and comment information is kept in the token stream, and some layout information is kept in the AST. The refactorer carries out program analysis with the AST, but performs program transformation with both the AST and the token stream, that is, whenever the AST is modified, the token stream will also be modified to reflect the changes. After a refactoring, we extract the new source code from the transformed token stream. More details are presented in (Li 2006).

Wrangler is implemented in Erlang using the Erlang Syntax Tools (Carlsson 2004) library from the Erlang/OTP release and Distel (Gorrie 2002) which is an extension of Emacs Lisp with Erlang-style processes and message passing, and the Erlang distribution protocol. Distel provides a very convenient way to integrate the refactoring tool with the Emacs editor. Erlang Syntax Tools provides functionalities for reading comments from Erlang source code and for inserting comments as attachments to the AST at correct places; and also the functionality for pretty-printing of Erlang AST(s) decorated with comments. Traversing an Erlang AST generated Syntax Tools is straightforward because all the non-leaf nodes in the AST have the same type.

We have extended the Erlang Syntax Tools library with functionalities for adding static semantic and location information to the AST. For example, binding structure of identifiers is stored in the AST by annotating each identifier occurrence with its defining location; each syntax phrase within the AST is also annotated with its start and end locations within the program source in terms of line and column numbers, etc.

As mentioned earlier, the multiple roles of atoms in an Erlang program, and the facility for dynamic composition of atom names impose real challenges for the correct implementation of certain refactorings. Currently, when an uncertainty arises regarding to an atom, Wrangler issues a warning message indicating which occurrence(s) of the atom causes the problem. Wrangler currently relies on the user to ensure that a refactoring is not affected by the dynamic composition of atoms, but we plan to tackle this problem by collecting and analysing run-time information of the project under consideration.

3.4 Availability of the Tools

HaRe and Wrangler can be downloaded respectively from

`http://www.cs.kent.ac.uk/projects/refactor-fp/hare.html`
`http://www.cs.kent.ac.uk/projects/forse`

Together with the downloads are `README` files containing installation instructions and information about how to turn on/off the refactoring engine, as well as documentations describing the meaning of each refactoring implemented in terms of *side-conditions* and transformations. HaRe's release contains a test suite for each refactoring. A wiki is available from our FORSE project webpage where we document our thoughts on refactorings.

4. Experience report

In this section we report on our experience of building the tools, contrast the two languages and tools, and also look at how this compares with the experience of object-oriented refactoring.

4.1 Refactoring = Condition + Transformation

We tend to think of refactorings simply as transformations, but in our tools the side-conditions for correct refactoring are typically more complex than the transformations themselves. An extreme example is *renaming*, where the side condition requires examination of the binding structure whereas the transformation involves replacing a single text field, but this is also the case for most other refactorings. The first refactoring systems tended to give less emphasis to elucidating the complete side-conditions, and to validate refactorings by post-refactoring regression testing, but side condition checking is now more common.

4.2 Are you sure that is what you mean?

Programmers typically refer to refactorings in a high-level, informal way; once an attempt is made to implement a refactoring, it becomes clear that there are a number of choices to be made about what exactly the refactoring in question might actually mean.

Take the example of a particular refactoring, such as *generalisation*. We generalise a function, `f`, say, by replacing a subexpression, `e`, by an additional formal parameter to the function, passing in the generalised expression as the actual parameter at call sites. It quickly becomes clear that this does not fully specify the refactoring, and in particular, it is not clear whether within the function body we should replace a *single* occurrence of `e`, *all* occurrences of `e`, or *some* occurrences, chosen by the user. This *'one/all/some'* choice is an issue for many refactorings.

Another set of choices is offered when a side-condition fails: continuing the previous example, suppose that `e` contains one of the formal parameters of `f`. It is possible for the refactoring to *fail*, or to *compensate* for this by lambda-lifting `e` and making corresponding adjustments to the body of `f`. Finally, in the context of Erlang, it is not possible statically to determine all calls to a particular function. On renaming the function, should a stub be left, redirecting calls to the new function, or should these calls simply *fail*?

4.3 Languages

The experience of tool building for programming languages gives a particular perspective on those languages; this section pulls a

number of these points. We targeted HaRe on the *de jure* standard, Haskell 98, but its use has been limited because almost all Haskell projects (including HaRe itself!) go beyond Haskell 98, making GHC (GHC) Haskell the *de facto* standard. GHC then becomes not only the standard, but also the standard platform, with which it becomes necessary to integrate the tools. The relative volatility of GHC as a proxy language standard has led the Haskell community – and particularly its tool builders and software vendors – to lobby for a further standardisation of the principal Haskell98 extensions; this *Haskell′* process is currently underway.

The absence or presence of certain language features cause problems for the tool builder. For instance, in Haskell it is not possible to hide items in export lists, making module-modifying refactorings more cumbersome. The more dynamic aspects of Erlang, such as the conversion of (computed) strings into atoms, make it impossible to give fully accurate flow analyses for all programs.

The idiosyncrasies of actual languages make it impossible in our view to build language-generic tools that are usable in practice. To take the particular example of two languages, Haskell and Erlang, which are both functional languages, there are significant differences of various different kinds:

- Their binding structures are very different, with the possibility in Erlang of multiple binding occurrences of a single variable, for instance; something that is impossible in Haskell.
- Their different semantics – evaluation in Haskell is lazy and Erlang it is strict – make a difference to the correctness of refactorings, such as *unfolding*.
- At the concrete level, their layout styles are radically different.

These are just a small number of examples among many differences, but serve as evidence of the difficulty of defining tools that are generic even between two programming languages.

4.4 Extensibility

We aim to build systems that are extensible, and we have provided a programmers' API in HaRe (and will in Wrangler). This API collects functions which are useful for the analysis and transformation of programs, and this is embedded in a declarative meta-language, namely Haskell or Erlang itself. This gives a declarative, straightforward and complete toolset, but at a relatively low level.

We have also investigated whether or not we can provide a higher-level library of combining forms, for assembling composite refactorings from simpler components, much as tactics are used in theorem provers. We have been unable to find such a simple, elegant solution, not least because of the 'two sorted' nature of refactorings, pointed out in Section 4.1 above. Our provisional conclusion is therefore that the language-embedded API provides the best balance between expressiveness and abstraction.

4.5 Verification and Validation

Both Haskell and Erlang are declarative languages, and it is therefore arguable that it is more straightforward to write formal proofs of correctness for refactorings written in these languages. Preliminary work on this reported in (Li and Thompson 2005), and further work in (Sultana 2007). We have also investigated using manual and automated testing infrastructures in testing our systems.

4.6 Infrastructure

In developing both tools we have chosen to re-use as much existing programming language infrastructure as possible. For Erlang this has meant building on the standard release and a number of widely-distributed libraries, but for Haskell the choice when we began the project was more difficult. GHC at that time provided no API to its internals, and so we used the Programatica framework. This has the advantage of providing many tools, but suffers from the disadvantage of not keeping up with language usage in the Haskell community.

We had also chosen to use existing tools for the front end of the refactorer. This we see as essential if the tools are to become part of a practitioner's standard toolkit, but developing for Emacs and (particularly) gVim is not a rewarding task. Finally, our systems need to integrate with other language tools, such as makefiles and test frameworks; we are addressing the latter point in a future project.

5. Conclusions

We have shown two tools, one mature and one under active development, for refactoring functional programs, and as well as giving details about their implementation, we have reported a number of conclusions based on our experience.

We are very grateful to the UK Engineering and Physical Sciences Research Council for its support for the projects to build HaRe and Wrangler.

References

J. Armstrong. *Programming Erlang*. Pragmatic Bookshelf, 2007.

J. Armstrong, R.Virding, C. Wikström, and M. Williams. *Concurrent Programming in Erlang*. Prentice-Hall, second edition, 1996.

R. Carlsson. Erlang Syntax Tools. `http://www.erlang.org/doc/doc-5.4.12/lib/syntax_tools-1.4.3`, 2004.

M. Fowler, K. Beck, J. Brant, W. Opdyke, and D. Roberts. *Refactoring: Improving the Design of Existing Code*. Addison-Wesley, 1999.

GHC. GHC – The Glasgow Haskell Compiler. `http://www.haskell.org/ghc/`.

L. Gorrie. Distel: Distributed Emacs Lisp (for Erlang). In *The Proceedings of 8th International Erlang/OTP User Conference*, Stockholm, Sweden, November 2002.

R. Lämmel and J. Visser. Generic Programming with Strafunski. `http://www.cs.vu.nl/Strafunski/`, 2001.

H. Li. *Refactoring Haskell Programs*. PhD thesis, Computing Laboratory, University of Kent, Canterbury, Kent, UK, September 2006.

H. Li, C. Reinke, and S. Thompson. Tool Support for Refactoring Functional Programs. In Johan Jeuring, editor, *ACM SIGPLAN Haskell Workshop, Uppsala, Sweden*, August 2003.

H. Li and S. Thompson. A Comparative Study of Refactoring Haskell and Erlang Programs. In M. Di Penta and L. Moonen, editors, *Sixth IEEE International Workshop on Source Code Analysis and Manipulation (SCAM 2006)*, 2006.

H. Li and S. Thompson. Testing Erlang Refactorings with QuickCheck. In *The Draft Proceedings of the 19th International Symposium on Implementation and Application of Functional Languages*, Freiburg, Germany, 2007.

H. Li and S. Thompson. Formalising Haskell Refactorings. In Marko van Eekelen, editor, *Trends in Functional Programming 2005*, 2005.

H. Li, S. Thompson, and C. Reinke. The Haskell Refactorer, HaRe, and its API. *Electr. Notes Theor. Comput. Sci.*, 141(4):29–34, 2005.

Huiqing Li, Simon Thompson, László Lövei, Zoltán Horváth, Tamás Kozsik, Anikó Víg, and Tamás Nagy. Refactoring Erlang Programs. In *The Proceedings of 12th International Erlang/OTP User Conference*, Stockholm, Sweden, November 2006.

PacSoft. Programatica. `http://www.cse.ogi.edu/PacSoft/projects/programatica/`.

S. Peyton Jones, editor. *Haskell 98 Language and Libraries: the Revised Report*. Cambridge University Press, 2003. ISBN 0-521-82614-4.

Refactor-fp. Refactoring Functional Programs. `http://www.cs.kent.ac.uk/projects/refactor-fp/`.

N. Sultana. Verification of Refactorings in Isabelle/HOL. MSc thesis, Computing Laboratory, University of Kent, 2007.