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 present 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, (gVim and Eclipse) 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.

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, semantics.

1. Introduction

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.

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 2007) programs. The current (fifth) release of Wrangler works with the complete Erlang/OTP language, and supports a few structural refactorings and functionalities for duplicated code detection. 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, we give an overview of our recent work in extending the Wrangler system and in formal verification of refactorings.

2. An Overview of HaRe and Wrangler

Both HaRe and Wrangler support interactive refactoring of multi-module programs. Snapshots of Wrangler embedded

WRT'08, October 19, 2008, Nashville, Tennessee, USA.

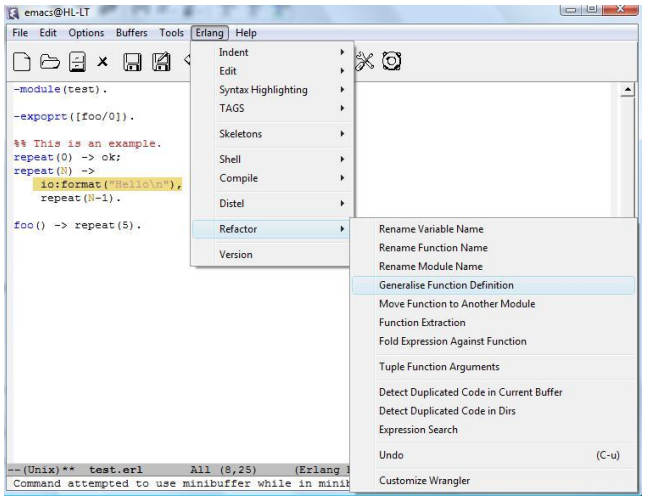

Figure 1. A snapshot of Wrangler showing the menus

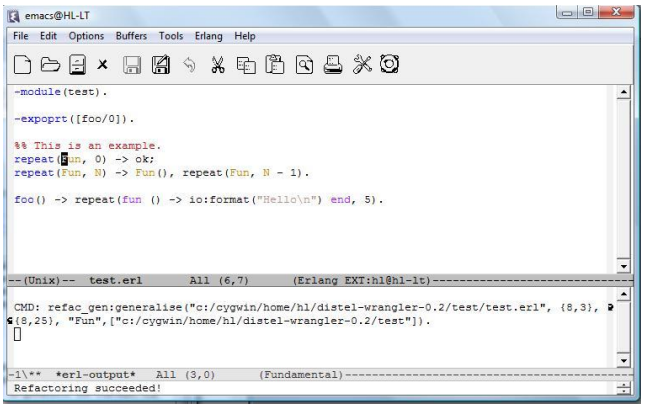

Figure 2. Wrangler after generalising a function definition

in the Emacs environment are shown in Figures 1 and 2; HaRe has a similar appearance.

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. 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 preconditions (or *side-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, and 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. To ensure the correctness of transformation when multiple modules are invovled, 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. 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 1, 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 `Fun` 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 2: the new parameter `Fun` has been added to the enclosing function definition `repeat/1`, which now becomes `repeat/2`; the highlighted expression has been replaced with `Fun()`; 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.

To retain the recognisability of programs, both HaRe and Wrangler preserve comments and layout of the refactored programs as much as possible, though the approaches taken are different.

While HaRe accepts only syntactically correct programs, Wrangler is able to refactor programs with or without syntax errors. When refactoring an Erlang program with syntax errors, function/attribute declarations to which these errors belong are not affected by the refactoring, but warnings asking for manual inspection of those parts of the program are given.

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 *preconditions*, 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.
- Module structure.
- Type information.
- Side-effect information.
- Comment and Layout information.

More detail on how each of these is used can be found in our paper tool support for refactoring functional programs (Li and Thompson 2008a).

3.2 Tool Support for Refactorings

Given a refactoring command, most static analysis-based refactoring tools (or engines) go through the following process, 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 *preconditions* of the refactoring. If the *preconditions* 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 *precondition* 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. 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.

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 – the Glasgow Haskell Compiler, 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.

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.

Different from the approach taken by HaRe to program appearance preservation, with Wrangler only ASTs are transformed during the refactoring process. After a refactoring, code for those functions/attributes that are not affected by the refactoring is extracted from their token stream annotation; and code for other functions/attributes are formatted by an improved pretty-printer which respects the code's original layout.

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`
`http://www.cs.kent.ac.uk/projects/forse`

4. Recent developments

In this section we summarise our recent work on developing facilities for duplicate code detection and data- and process-oriented refactorings in Erlang, on integrating Wrangler with Eclipse and also on the topic of the mechanical verification of refactorings.

4.1 Duplicate code detection and refactoring

We have implemented a tool for detecting duplicate code (or code clones) in Erlang projects, either in a single module or across the whole project. The detection algorithm, reported in the more detail in (Li and Thompson 2008b), uses a combination of the token stream and the AST to ensure efficient detection of syntactically-meaningful duplicate code.

Once a code clone set is found, it is possible to use refactorings in Wrangler to remove the clones. First, the duplicate code is extracted into a function, next it may be generalised to abstract over any literals (which are neglected in the clone detection algorithm). Finally, it is possible to step though all the instances of this function in the code base, deciding for each one whether or not to replace it by a function call.

4.2 Additional Erlang refactorings

Our report (Li et al. 2008) describes refactorings, implemented by M. Tóth, to turn a sequence of function arguments into a single (tuple) argument, as well as describing preliminary work on refactoring to introduce record structures.

In the same paper we report on having begun work on refactorings which address the process structure of an Erlang system. Since processes and the communication channels between them are implicit in an Erlang program, it is necessary to perform static and dynamic analysis to extract as much information as possible, as well as requiring non-trivial user intervention too.

4.3 Wrangler and Eclipse

Using the Erlide Eclipse plugin for Erlang (Erl) G. Orosz has incorporated a number of refactorings for Erlang into Eclipse, using the LTK framework. This allows a full integration of the refactorings into the Eclipse system, including the undo/redo system. On the other hand the LTK framework imposes a number of restrictions on the refactoring workflow, and this and other challenges are discussed in more detail in (Li et al. 2008).

4.4 Mechanical verification of refactorings

Nik Sultana and the second author have investigated ways in which refactorings can be verified mechanically using the Isabelle/HOL system (Sultana and Thompson 2008a) and how code for refactorings can be extracted from proofs of the correctness of the corresponding transformations (Sultana and Thompson 2008b).

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. A number of general reflections on the process of building tools for refactoring functional programs can be found in Section 4 of (Li and Thompson 2008a), of which this position paper is an abbreviated and updated version.

References

Erlide - the Erlang IDE. `http://erlide.sourceforge.net/`.

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

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

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 and S. Thompson. Tool Support for Refactoring Functional Programs. In *Partial Evaluation and Program Manipulation (PEPM)*, San Francisco, California, USA, January 2008a.

H. Li and S. Thompson. A Comparative Study of Refactoring Haskell and Erlang Programs. In M. Di Penta and L. Moonen, editors, *SCAM*, 2006.

H. Li and S. Thompson. Clone Detection and Removal for Erlang/OTP within a Refactoring Environment. In *Draft Proceedings of the Ninth Symposium on Trends in Functional Programming(TFP)*, The Netherlands, May 2008b.

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

H. Li, S. Thompson, G. Orosz, and M. Tóth. Refactoring with Wrangler, updated. In *Proceedings of the Seventh ACM SIGPLAN Erlang Workshop*, ACM Press, September 2008.

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.

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

N. Sultana and S. Thompson. Mechanical Verification of Refactorings. In *Partial Evaluation and Program Manipulation (PEPM)*, San Francisco, California, USA, January 2008a.

N. Sultana and S. Thompson. A Certified Refactoring Engine. In *Draft Proceedings of the Ninth Symposium on Trends in Functional Programming (TFP)*, May 2008b.