Refactoring Functional Programs (GR/R75052/01) *Final Report*

Simon Thompson & Claus Reinke (Investigators), Huiqing Li (Research Associate)
Computing Laboratory, University of Kent

1 Introduction and Background

Refactorings are source-to-source program transformations that change program structure and organisation, but not program functionality. Documented in catalogues and supported by tools, refactoring provides the means to adapt and improve the design of existing code, and has thus helped to address long-standing problems in software maintenance while also enabling the trend towards modern agile software development processes. Two typical refactoring scenarios are: adapting the structure of an existing code base to prepare for adding functionality or fixing a bug; cleaning up a code base after extensive functional changes.

The UK is a world leader in functional programming research. Haskell is the principal lazy functional programming language, and its main implementation, the Glasgow Haskell Compiler, is a UK product (whose original implementations were supported by EPSRC funding). Other activity sponsored by EPSRC in this area includes Runciman and Chitil's work on tracing and debugging, and the resource-bounded computation activity around Hume by Hammond, Michaelson and their co-workers.

Complementing these activities, our project has brought the ideas and techniques of refactoring to functional programming. The project has taken a practical approach to the investigation by building a fully-featured tool, called HaRe for 'Haskell Refactorer', for performing refactorings for the Haskell programming language. HaRe supports the full Haskell 98 standard, is integrated with the most popular Haskell development environments and is implemented to preserve the appearance of source code after refactoring. HaRe is thus designed to be a practical programmers' tool, rather than a prototype. More details of HaRe are given in Section 3.1.

It is an indication of the progress that functional programming has made that the project was able to build on the work of others. In particular, we were able to use the front-end of the Programatica [13] system to assist our program parsing and analysis and also to use the Strafunski [6] library of tree transformation code. We are very grateful to the international collaboration of these development teams, in the USA and Europe, both for their systems and for the advice and modifications that they have performed on our behalf.

The remainder of the report is structured thus. The next section reports our project's objectives and progress. Section 3 reports major activities, including the HaRe system and Huiqing Li's PhD thesis. Section 4 outlines our dissemination of the work, including papers published, presentations, meetings and summer school courses. We conclude with a discussion of plans for future work.

2 Project objectives and progress

The revised objectives for the project are listed here. The original objective 4 was not pursued, and was replaced by objective 6. Progress against each objective follows its statement, and refers to the narrative in later sections.

1. To develop a functional programming perspective on recent, practice-driven research into flexible program structure and refactoring. This is reported in Huiqing Li's thesis (Section 3.2) and disseminated by means of publications and presentations (Sections 4.3, 4.4 and 4.5).

2. To develop a catalogue of candidate refactorings for a modern functional language. This is available on the web; more details in Section 3.3.

3. To develop prototypical tool support for a selection of refactorings from this catalogue. The HaRe tool for refactoring in the Haskell language is described in detail in Section 3.1. Progress in this objective exceeds our original expectations: the system is not merely a prototype, but covers the full Haskell 98 language, is integrated with the most commonly used Haskell development environments and preserves the appearance of the source code which is being transformed.

4. To evaluate the work with reference to existing change histories of large Haskell systems, including that of the Glasgow Haskell Compiler. This objective was predicated on the existence of suitable open source software development archives for substantial Haskell projects. These development archives certainly exist, but Chris Ryder's PhD project [15] showed to us that the existing Haskell archives provided insufficient support for identifying individual changes and their intentions (refactoring/bug fix/feature change) without manual inspection of the code by the project researchers. In [15] this work was done for a small set of libraries, but the ratio of effort to outcome was too high to make it cost effective. For this reason this objective was not pursued.

We would still consider it valuable to investigate the change history of a large Haskell system, in terms of metrics, refactoring, design patterns, etc. . However, this would involve substantially more effort than originally envisioned, including a more disciplined use of revision control within a development project which was prepared to collaborate in the exercise, provision of tool support for analysis of versions as well as substantial, dedicated person-time.

5. To investigate the connection between the work on program structure and refactoring in the object-oriented and functional communities. Our experience in investigating this issue was that the differences between refactoring in the OO and the functional context were, initially at least, less marked than we had expected. This may well be due to the fact that simple refactorings affect program *representation and binding* structures, i.e., aspects of syntax and static semantics more than those of dynamic semantics, and, at this level, structural issues and refactorings are similar in OO and FP. More complex refactorings, such as that making a concrete type abstract, begin to show paradigm-specificity.

In developing the system we were able to confirm that it is possible to implement checks of refactoring side-conditions which guarantee the correctness of the transformation. This is due to the cleaner intuitive semantics of functional languages, and has enabled proofs of correctness of certain refactorings to be constructed (Section 3.2).

6. To build a reusable platform/research infrastructure for Haskell 98 program transformation and analysis (HaRe API). This objective arose directly from community feedback on our early prototypes and further develops the tool to allow the work of this project to be extended and reused by the Haskell community. Specifically, it provides a collection of libraries for the analysis and (source to source) transformation of Haskell 98 programs. The refactorings in HaRe are, in the released version, implemented using this API, and its availability allows others to implement their own refactorings in HaRe (instead of needing the HaRe team to do this). Moreover, the API is of value to all who seek to build tool support for program transformations in Haskell; the API has been used by colleagues at the University of the Minho amongst others. Further details are provided in Section 3.1.

3 Major activities

This section reports on the HaRe refactoring system and the PhD thesis *Refactoring Functional Programs* and the catalogue of Haskell refactorings, before discussing a number of other project-related activities.

3.1 The HaRe system

The major deliverable from the project *Refactoring Functional Programs* is the Haskell Refactorer, HaRe. Three major releases have taken place through the project, each extending the facilities of its predecessor. Minor updates of these releases have continually been released up to the present point. Features of the currently released system include:

The refactorings implemented Before discussing the details of which refactorings have been implemented, it is worth stressing the motto

Refactoring = Transformation + Condition

That is, the implementation of each refactoring has two distinct aspects. Taking the example of renaming a particular binding, from `foo` to `bar`, say. It is necessary to implement the *transformation* so that only those instances of `foo` referring to the definition of interest are renamed, throughout the project. Prior to performing the transformation it is necessary to check the *condition* that the renaming will not affect the bindings of the program, by any sort of 'name capture'.It has been our experience that it requires more programming effort correctly to implement the conditions than to implement the transformations themselves.

As should also be apparent from this short example, implementing the refactorings calls for much more than text editing facilities. Lexical analysis, parsing, static semantics, type and module analyses, in other words all the facilities provided by the front end of a compiler, are required correctly to implement the transformations and conditions of the refactorings.

The refactorings supported by HaRe are listed in groups of related operations now.

Names/Scopes These refactorings affect the naming or scope of a definition. They share preconditions which require that they leave unchanged the *binding structure* of uses of names to their defining occurrences.

- Rename any entity (function, variable, type) in the system.
- Lift a local definition to the top level in the module which contains it.
- Lift a definition by one level, i.e. to its innermost enclosing scope.
- Demote a definition to the unique definition which calls it.

Definitions These refactorings modify the definitions provided by a module, either by adding or removing definitions or by changing – for example generalising – existing definitions. A refactoring to move a definition between modules is also provided.

- Introduce a new definition for an existing piece of code, e.g. a sub-expression of a given function.
- Unfold a definition by replacing an instance of its left-hand side by an instance of its right.
- Generalise a definition. This replaces a sub-expression (e.g. a constant) within a definition by a formal parameter; at every existing call site the sub-expression is passed in as the actual value. This refactoring prepares the definition to be reused with different parameter values.
- Remove a definition which is no longer used.
- Duplicate a definition prior to making a modification to one of the instances.
- Add a new parameter to an existing definition.
- Remove a parameter which is not used within the body of the definition.

- Move a definition to another module.

Import/Export This set of refactorings allows control of the bindings that are imported and exported by a module.

- Clean the list of imports, so as to import only the modules and bindings that are used.
- Make an import explicit by listing the required bindings in the import statement, rather than importing a complete module.
- Add a binding to the list of exports.
- Remove a binding from the list of exports.

Data types This class of refactorings supports various atomic steps in the composite refactoring which replaces a concrete, algebraic, data type (`t`, say) by an abstract type. This in turn enables the implementation of the type to be changed without any client code being affected.

- Add field labels to the type `t`, so that instead of using pattern matching for selecting fields of a data value of type `t`, the labels (*qua* selector functions) can be used instead.
- Add discriminator functions which can be used to decide which constructor has formed a data value of type `t`.
- Add constructor functions, which are used to construct data values of `t`, replacing the constructors themselves.
- Before patterns involving the constructors of type `t` can be eliminated, it is necessary to eliminate any patterns over other types which are nested within `t` patterns.
- Once nested patterns have been removed, it is possible to eliminate patterns in favour of the selector and discriminator functions.
- Create an ADT module, so that all but the constructors of `t` are exported.
- If the preceding five refactorings are composed, this has the effect of replacing a concrete data type by an ADT.

Undo It is possible to undo the effect of any sequence of refactorings, so that refactoring can be carried out in a speculative way without incurring any cost or commitment to any undesired consequences.

The project could only implement a selection of the possible refactorings for Haskell; in order to allow others to implement further refactorings, we have developed a programmers' API; more details of this below.

System uses existing components and libraries A fully-featured refactoring tool requires all the facilities provided by the front end of a compiler. At the start of the project, the Programatica system [13] emerged as the only system to provide these facilities with a relatively accessible and stable interface, and so HaRe builds on the Programatica infrastructure. An API [1] has recently been developed for the Glasgow Haskell Compiler, and the HaRe team are actively exploring the possibility of porting HaRe to GHC while using our own project experience to provide feedback on the evolving API design (see also Section 3.4).

Each refactoring implemented consists of two parts:checking that the *preconditions* of the refactoring hold, and then performing the *transformation* of the AST. In both cases, it is necessary to walk through a tree representing the program structure. Such tree walks consist in the main of 'boilerplate' code performing default actions with non-default actions occurring only at particular points. Support for this style of code, within the strongly-typed environment of Haskell, is provided by the Strafunski system [6].

Integration with programmers' preferred tools HaRe, which is implemented in Haskell, can be used stand-alone, but it is also embedded within the programmers' editors *emacs* and *vim*, as it is the experience of previous projects that stand-alone tools fail to be adopted, if they require the programmer to change their development routine. These particular editors were selected as they were the most popular program development environments chosen in a survey of Haskell developers at the start of the project.

Preservation of source code appearance If programmers are to use the system they must recognise the output from a refactoring. Initial experiments showed that it was insufficient to *pretty print* the refactored abstract syntax tree (AST): programmers have definite, individual layout and comment styles. So, it was necessary for the system to retain whitespace and comment information. Off-the-shelf systems such as the Programatica front end do not retain this information in the AST, and so, if the Programatica system was to be used without modification, it was necessary simultaneously to process the AST and the token stream from the lexical analyser. This caused substantial complication, but resulted in a system with high fidelity to a programmer's style.

Coverage of the Haskell 98 standard The third requirement necessary to the system being usable by working programmers is that it covers the full language: in this case the standardised language Haskell 98 [5].

Fully module-aware All the HaRe refactorings operate across multiple-module programs, since any practical Haskell system will consist of a collection of modules. In a typical refactoring, such as generalisation, the module containing the generalised definition certainly must change, but all clients of the generalised function, from any module in the project, need to be modified too.

System widely tested The correctness of refactorings can be assessed in multiple ways. The first criterion is that the abstract syntax tree representing the transformed program meets the appropriate conditions. More strongly, it is possible to check that the layout of the resulting program is as required. Both aspects of the systems have been extensively tested.

System widely available To avoid locking out any group of potential users, HaRe is designed to be platform independent (including MacOS, Windows and Solaris), and the separation of refactorer and GUI should facilitate integration with newly emerging Haskell IDEs.

Application Programmer Interface The application programmer interface (API) is designed to allow users to define new refactorings for themselves. The API is structured as two layers over the syntax provided by Programatica and the traversal infrastructure of Strafunski. The lower layer allows a programmer explicitly to modify the AST and the layout of the program. However, it is the upper layer, which hides token-stream manipulation, which is intended to serve as the general purpose programmers' API. Using this, it is possible to modify the program structure in a number of general ways, and it is this layer that is used internally to implement the refactorings in HaRe.

Other aspects of the system, including plans for future developments, are outlined below.

3.2 *Refactoring Functional Programs* PhD thesis by Huiqing Li

The project research associate, Huiqing Li, registered for a PhD during the project period. She submitted her thesis in September 2005, and it will be examined by the end of 2005.

Introduction Introduction to the fields of functional programming and refactoring, including discussion examples, tool support and formal description of refactorings.

A Model of Refactoring This chapter examines the notion of refactoring for functional programs in more detail, including addressing the question of the precise basis of behaviour preservation, and other properties, such as source code layout preservation, desirable in a refactoring tool. The chapter then gives an overview of the refactorings implemented in HaRe.

Technology Background A tool for refactoring requires infrastructure support, both for language processing and for effective implementation of tree processing algorithms. HaRe uses Programatica and Strafunski for these; this chapter gives the rationale for this before describing aspects of the systems relevant to HaRe.

The Design of HaRe This chapter gives a high-level overview of the design rationale for HaRe, referring to a running example. This is followed by a description of the high-level architecture of the system and in particular the interfaces that it presents to the user and the application programmer.

The Implementation of HaRe Here the detailed implementation is described, referring to two refactorings whose full implementation is contained in an appendix.

An API for Writing Refactorings The application programmer interface is described in this chapter. The account includes coverage of the two levels of the API which, respectively, hide and reveal manipulation of the lexical token stream.

Formalisation of Refactorings This chapter formalises two refactorings – generalising and moving a definition between modules – in the context of the λ_{letrec}-calculus. In order to formalise the latter refactoring, it is necessary to extend the calculus to the λ_M-calculus, which includes a notion of modules akin to those of Haskell.

Related Work This chapter summarises a variety of work, principally in machine support for refactorings for a variety of languages and for a spectrum of different kinds of program transformation for functional programs.

Conclusions and Future Work The future work flagged here includes use of type information in refactorings, more fine-grained user interactions, further formalisation, and 'bad smell' detection.

Appendices The appendices include the algorithm for layout preservation; a complete description of the HaRe API and complete implementations of renaming and moving definitions between modules.

3.3 Catalogue of refactorings

An online catalogue of refactorings for Haskell – some of which are implemented in HaRe – is available at `http://www.cs.kent.ac.uk/projects/refactor-fp/catalogue/`.

3.4 Haskell 98 and GHC Haskell

The project team made an early decision to target the full Haskell 98 standard in designing the system. To this end, it was necessary to choose a front-end system which would support the standard and which would provide all the facilities that we required, not simply parsing and lexical analysis. Programatica provided this facility, and we would like to take this opportunity to acknowledge the support given by the Programatica team, and in particular Thomas Hallgren, to the HaRe project.

It has become clear, however, that despite the existence of the *de jure* standard Haskell 98, a *de facto* standard has emerged, namely Glasgow Haskell as implemented in the Glasgow Haskell Compiler(GHC). At the point when we made the decision about front ends, GHC was not in a state suitable for providing a front end, but recently, and indeed in collaboration with the HaRe team, the GHC team have devised an API which, among other things, packages up the GHC front-end services (such as the parser, type checker and module analysis) [1]. In order to explore the feasibility of porting HaRe to the GHC API, Chris Ryder has performed a feasibility study, the results of which are reported in [14].

3.5 Related projects

Chris Ryder, whose PhD degree was confirmed in summer 2005, has written a thesis on the subject of metrics and program visualization for functional programs in Haskell. His Medina system is available for download [11], and is reported in [16]. This work was used by the summer intern Jonathan Cowie, who investigated ways of detecting 'bad smells' in Haskell during his internship in summer 2004 and in a subsequent final-year student project.

Chau Ngyuen Viet, received a Nuffield Undergraduate Bursary URB/01607/G in summer 2004 to work on implementing 'traditional' functional transformations, such as deforestation, in HaRe. This informed the development of the API for HaRe [10], and Chau's work is reported in [12].

4 Interactions and Dissemination

The HaRe project has been active in its interactions with the research community, including students and peers.

4.1 Visits & visitors

A number of researchers visited the Computing Lab at the University of Kent to interact with the Kent research team.

Ralf Lämmel (CWI, Amsterdam; September 2002): Ralf visited the department to explain and demonstrate to us the *Strafunski* [6] system, which is used within HaRe to support tree traversals in both condition checking and effecting refactoring transformations.

Lars-Åke Fredlund (SICS, 2003, 2004): Lars-Åke visited the department to collaborate with an ongoing project on verification for the Erlang [2] programming language, and engaged in discussions with the HaRe team about refactoring for Erlang, and particularly about the semantics of such refactorings.

Francesco Cesarini (Erlang Consulting; May 2004): Francesco presented a a one-day version of his five day course on programming in Erlang using the Open Telecoms Platform. The context of this was ongoing discussions about refactoring in Erlang and other work in the department on verifying Erlang/OTP systems.

Tom Mens (University of Mons-Hainaut; June 04): Tom visited Kent to present his work on describing and implementing refactorings using graph transformation techniques, in contrast to our use of tree transformations (implemented as Haskell functions). This allowed us to clarify the advantages and disadvantages of our respective approaches.

Martin Erwig (University of Oregon, March 2005): Martin visited Kent to discuss work on *monadification* of Haskell programs. His work [4] describes a particular style of monadification and shows that neither the existing monadification algorithms nor their outputs are comparable; we wrote a discussion paper on the various forms of monadification and solicited comments on this on-line [3] and are investigating an alternative, type-directed approach that does generalise previous work[1].

José Proença (University of the Minho; April-May 2005): José visited the HaRe team to implement a number of 'pointfree' transformations using the HaRe programmers' API. In the course of this José was able to give valuable feedback on the design and details of the API

[1]Claus Reinke, work in progress. Prototype code and successful test runs for examples in Erwig's [4] and our work [3] available on request.

4.2 Mid-project workshop, February 2004

A mid-project workshop was held in the Computing Lab at Kent in February 2004. This allowed us to do three things. We were able to disseminate information about the current state of the HaRe system. Equally important we were able to gather valuable feedback from the community, not only on the generalities of Haskell refactoring and HaRe, but also on the detailed implementation of the system. Finally, we were able to acquaint ourselves with related work in the field.

Attendees included Stephen Drape, Ran Ettinger, Jeremy Gibbons and Ganesh Sittampalam (Oxford University Computing Lab); Thomas Hallgren (PacSoft, Oregon Graduate Institute, Beaverton, USA); Simon Marlow (Microsoft Research, Cambridge, UK); Patrik Jansson (Computing Science, Chalmers University, Göteborg, Sweden); Eelco Visser (Center for Software Technology, Utrecht University, The Netherlands); Ralf Lämmel (CWI and VU, Amsterdam, The Netherlands); Joost Visser (Departamento de Informática, Universidade do Minho, Portugal) as well a number of Computing Lab staff. Further details of the workshop are available at `http://www.cs.kent.ac.uk/projects/refactor-fp/workshop.html`.

4.3 Presentations by the HaRe team

- *A Case Study in Refactoring Functional Programs*, Invited keynote presentation by Simon Thompson, 7th Brazilian Symposium on Programming Languages (SBLP'2003), May 2003. [19]
- *Tool Support for Refactoring Functional Programs*, presented by Claus Reinke, ACM SIGPLAN 2003 Haskell Workshop, August 2003. [7]
- *HaRe: The Haskell Refactorer*, presentation by Huiqing Li, Workshop on Datatype-Generic Programming, Oxford, June 2004.
- *Progress on HaRe, the Haskell Refactorer* poster presentation by Huiqing Li and Simon Thompson, International Conference on Functional Programming (ICFP 2004), September 2004. [9]
- *The Haskell Refactorer, HaRe, and its API*, tool demonstration by Huiqing Li, Fifth Workshop on Language Descriptions, Tools and Applications (LDTA 2005), April 2005. [10]
- *Designing and implementing a refactoring tool for an existing functional programming language: Haskell*, presentation by Simon Thompson, Meeting on Transformation Techniques in Software Engineering, Schloss Dagstuhl, April 2005. [17]
- *HaRe: the Haskell Refactorer* invited presentation by Simon Thompson at the Second EPSRC ASTReNet Workshop, London, June 2005.
- *Formalisation of Haskell Refactorings*, presented by Huiqing Li, Trends in Functional Programming, Tallin, Estonia, September 2005. [8]

4.4 Summer Schools

Simon has also delivered short courses on *Refactoring Functional Programs* to audiences of advanced undergraduate and postgraduate students, industrialists and academics at

- *5th International Summer School on Advanced Functional Programming*, Tartu, Estonia, 14-21 August, 2004. [18]
- *Central-European Functional Programming School* Etvs Lornd University, Budapest, Hungary, 4-16 July, 2005.

4.5 Research Presentations

Members of the HaRe team have also made presentations about the project at the University of the Minho, Braga, Portugal, at the Centrum voor Wiskunde en Informatica, Amsterdam, and in the UK at the Universities of Exeter, Durham, Manchester, Oxford, Kingston and Heriot-Watt.

4.6 Other Dissemination

The HaRe project is open source, with a BSD-style licence, and so is available to others to develop further. As well as making HaRe available, it provides additional leverage on other open source projects upon which we have built.

Martin Fowler has written the key text on object-oriented refactoring, and maintains the website `www.refactoring.com`. HaRe is linked from the Tools section of that site.

5 Future work

Work on the HaRe system is being continued by Chris Brown, a PhD student supported by a Brian Spratt Bursary in the Computing Lab at the University of Kent. It is anticipated that Chris will add a suite of data-oriented, type-aware, refactorings to HaRe and investigate the development of a user-level language for composing refactorings.

The experience gained in the HaRe project will be invaluable in building refactoring support for Erlang as part of the EPSRC supported *Formally-based Tool Support for Erlang Development* (EP/C524969/1) jointly with the University of Sheffield. Huiqing Li, the HaRe research associate, is the named researcher on this project, which started in July 2005.

The team will continue to collaborate with HaRe users and programmers to develop the system as an open source enterprise, and to publish and disseminate the results as widely as possible. In particular, we plan to seek funding to port the system to the GHC API (see Section 3.4) to ensure the even wider exploitation of HaRe and its API.

Bibliography

[1] Krasimir Angelov and Simon Marlow. Visual Haskell A full-featured Haskell development environment. In Daan Leijen, editor, *Haskell Workshop 2005*. ACM Press, 2005.

[2] Joe Armstrong, Robert Virding, and Mike Williams. *Concurrent Programming in ERLANG*. Prentice Hall, 1993.

[3] Martin Erwig, Claus Reinke, and Simon Thompson. Online survey on *Monadification as a Refactoring*. `http://www.cs.kent.ac.uk/projects/refactor-fp/Monadification.html`.

[4] Martin Erwig and Deling Ren. Monadification of Functional Programs. *Science of Computer Programming*, 52(1-3):101–129, 2004.

[5] John Hughes and Simon Peyton Jones, editors. *Report on the Programming Language Haskell 98*. `http://www.haskell.org/report/`, 1999.

[6] Ralf Lämmel and Joost Visser. Generic Programming with Strafunski, 2001. `http://www.cs.vu.nl/Strafunski/`.

[7] Huiqing Li, Claus Reinke, and Simon Thompson. Tool Support for Refactoring Functional Programs. In *ACM Sigplan Haskell Workshop*, 2003.

[8] Huiqing Li and Simon Thompson. Formalisation of Haskell Refactorings. In *Trends in Functional Programming*, 2005.

[9] Huiqing Li, Simon Thompson, and Claus Reinke. Progress on HaRe, the Haskell Refactorer. In *International Conference on Functional Programming 2004*. ACM Press, 2004.

[10] Huiqing Li, Simon Thompson, and Claus Reinke. The Haskell Refactorer, HaRe, and its API. In *Fifth Workshop on Language Descriptions, Tools and Applications (LDTA 2005)*, 2005.

[11] The Medina Metrics Library. Available from `http://www.cs.kent.ac.uk/people/staff/cr20/medina/`.

[12] Chau Nguyen Viet. Transformation in HaRe. Technical report, Computing Laboratory, University of Kent, 2004. `http://www.cs.kent.ac.uk/pubs/2004/2021`.

[13] The Programatica project. `http://www.cse.ogi.edu/PacSoft/projects/programatica/`.

[14] Chris Ryder. Porting HaRe to the GHC API. Technical Report 8-05, Computing Lab, Univ. of Kent, 2005.

[15] Chris Ryder. *Software Measurement for Functional Programming*. PhD thesis, University of Kent, 2005.

[16] Chris Ryder and Simon Thompson. Software Metrics: Measuring Haskell. In *Trends in Functional Programming*, 2005.

[17] Simon Thompson. Designing and implementing a refactoring tool for an existing functional programming language: Haskell. In J. Cordy, R. Lämmel, and A. Winter, editors, *TransformationTechniques in Software Engineering*. `http://www.dagstuhl.de/05161/`.

[18] Simon Thompson. Refactoring Functional Programs. In *5th International Summer School on Advanced Functional Programming, Tartu, Estonia*, volume XXX. Springer Lecture Notes in Computer Science, 2005.

[19] Simon Thompson and Claus Reinke. A Case Study in Refactoring Functional Programs. In *Brazilian Symposium on Programming Languages*, 2003.