Refactoring functional programs

Simon Thompson, Claus Reinke
Computing Laboratory, University of Kent

1 Executive Summary

Refactoring

Refactoring is about *improving the design of existing code* and as such it must be familiar to every programmer, software engineer and designer. Its key characteristic is the focus on structural changes, strictly separated from changes in functionality:

Before changing the functionality of a software system, it is often necessary to restructure it to make it more amenable to change. *After* establishing a working piece of software, it is often necessary to revise its structure of names, modules, types and so forth.

A common example of refactoring is the replacement of a concrete data type by an abstract data type (ADT): all direct references to the representation type must be replaced by uses of the ADT selectors and discriminators. After this refactoring, it is possible to change the implementation of the abstract data type without affecting its client code at all; refactoring thus enables system change, modifying a design to introduce information hiding as and when it becomes necessary.

This example illustrates two properties common to most refactorings. First, the impact of a refactoring is *diffuse*: to effect the change, edits and checks are needed not only in the module exporting the type, but in all importing modules as well, potentially affecting every file in the source tree of a large system. Secondly, the refactoring is *bi-directional*: in some situations it will be appropriate to replace an ADT by a concrete type (in functional languages, pattern matching offers a particularly succinct notation when working with concrete, algebraic types).

Within the SE and OO communities, refactoring has been identified as central to the software engineering process [10, 6, 18]. The web site `http://www.refactoring.com` contains a comprehensive, evolving catalogue of refactorings of object-oriented designs. Refactoring in object-oriented languages is supported by tools like the Refactory [23], whose purpose is to help a user to make the diffuse set of changes that comprise a refactoring.

The overall goals of the project proposed here are to compile a library of refactorings for functional programs and to implement a tool to support refactoring for programs in Haskell.

Functional Programming

Functional programming languages such as Haskell and ML embody a powerful, abstract programming model, including type systems with algebraic data types, polymorphism, type classes and overloading, (type-)parameterised modules called functors; higher-order functions; lazy and strict evaluation styles [12].

The languages have a clean mathematical semantic basis on which it is possible to establish the formal equivalence of programs. This has allowed functional programmers to describe and validate a variety of program transformations for programs in both source and compiler intermediate languages.

Functional languages have been used in substantial projects, initially within the research community [21, 2, 9]. More recently, evolving libraries of general purpose code, many extensions (including concurrent programming) and foreign function interfaces have allowed these languages to be used to build 'real world' systems, such as web servers (see [1, 27] for details).

Refactoring Functional Programs

Some refactorings are paradigm-independent:

- move a definition from one module to another;
- change the scope (i.e. visibility) of a definition;

others have a functional flavour:

- replace pattern matching over an algebraic data type with the operations of an abstract data type;
- generalise a function working over a single type into a *polymorphic* function applicable to a whole class of types
- replace a function by a constructor.

Some functional refactorings have OO counterparts; others are unique to the functional paradigm. Examples of functional refactorings are discussed in more detail in our online catalogue [25].

In the work on OO refactoring, heavy emphasis is put on testing to validate particular instances of refactorings. In refactoring functional programs we will be able to take a different approach. From experience, we know that the majority of the errors introduced by hand refactoring tends to be caught by the strong type system of Haskell. We also expect to be able to *prove* the validity of refactorings, in the presence of certain identified side-conditions; this is often not practicable for existing OO languages.

Existing work on program transformation in functional programming has concentrated on 'vertical', algorithmic transformations which move from abstract specifications to efficient implementations. The 'horizontal' transformations involved in refactoring operate on the structure of systems in an orthogonal way.

Project Aims

I to develop a functional perspective on recent, practice-driven research into flexible program structure and refactoring

II to develop a catalogue of candidate refactorings for a modern functional language

III to develop prototypical tool support for a selection of refactorings from this catalogue

IV to evaluate the work with reference to existing change histories of large Haskell systems, including that of GHC [9]

V to investigate the connection between the work on program structure and refactoring in the object-oriented and functional communities

The work described here is *novel*: it will investigate the notion of refactoring in an entirely new context, it will exploit the strengths of functional languages for a new application, and it will document correspondences between OOP and FP design patterns.

The work is *timely*, as it will leverage developments in functional compilers and build on the 'real world' capabilities mentioned above to support a community which is beginning to commercialise the functional approach [7]. It will also establish a bridge between current software engineering and FP research.

The work has a number of different *beneficiaries*. Functional programmers will gain immediately from the catalogue produced and in the longer term by being able to use the refactoring tool. Reflections on refactoring will inevitably have implications for language designers and implementors, and work on functional refactoring will complement ongoing work on OO refactoring.

2 Background

Refactorings are source-to-source program transformations that change program structure and organisation, but not program functionality. Seen as the realisation of software re-design, such changes to software representation and organisation are immediately visible to software developers who operate on the source code as an artifact, whereas other users of the software are only affected by changes in functionality. Refactoring is therefore most relevant to software development and maintenance, with longer-term implications for language design.

The aim of early work on assistance for program restructuring [11] was *to reduce the negative impact of fixed program structure on the costs of software maintenance*: over time, adding small changes to a program without ever adapting its fundamental structure degrades the overall quality of the code and makes successive maintenance activities ever more difficult, until finally the original structure is so encrusted in modifications that it becomes cheaper to re-start from scratch.

Manual restructuring is a time-consuming long-term investment, error-prone and hard to justify in the short term. Tool support for program restructuring, proven to preserve program functionality, can alleviate this problem, going a long way towards making software soft, i.e., malleable for the maintenance programmer. Refactoring itself should ideally not involve any risk of introducing or hiding bugs, and if this is guaranteed by good tool support, even short-term refactorings suddenly become feasible, e.g., providing simplified program views by separating out bug-related program fragments can ease debugging.

It is important to realise that the need for design changes is not necessarily a consequence of bad design: designers cannot "get the design right" from the start, simply because there is no single best design. What constitutes a good software design depends on the uses the software is put to, and as these uses vary over the software lifetime, so does the value of any given design.

This realisation, combined with tool support for the code manipulations involved in re-design, has opened a new focus of interest in refactoring, as *a core technique for a new approach to software development*. Taking the shift from waterfall models to iterative development methods to its extremes, new lightweight or agile methods [5], such as extreme programming [28], acknowledge the importance of maintenance, being at least as costly as the original development, the difficulties of achieving perfect up-front design, and the importance of software evolution: "Refactor mercilessly to keep the design simple as you go and to avoid needless clutter and complexity" [28].

The positive influences of refactoring on software maintenance and software design have *brought the issue of program structure flexibility to the attention of language designers*, who are now faced with the problem of evaluating those recent advances and elevating them into improved language designs. Research on this third, language design aspect of program restructuring is still in its early, experimental stages, and not clearly separated from developments in tools and programming environments. The issues of flexible program structure and multiple views of source code are currently being researched from several angles [13, 15, 19].

In brief, *successful software evolution depends on design evolution, and the role of refactoring research is to formulate the transformations involved, and to develop tools that translate semantic-level design changes into code-level representation changes*.

2.1 Why investigate functional refactoring?

Functional languages differ from object-oriented languages in their theory and practice so that an investigation of refactoring in the context of functional languages promises results at two levels. The functional programming community will gain access to the benefits of refactoring tools and theories, and the project will also offer a useful perspective on object-oriented refactoring.

Good Theoretical Foundations

Refactorings form a theory which is ultimately grounded on the semantics and program equivalences for the language in question. Functional programming research provides rich theoretical foundations for reasoning about programs using denotational and observational program equivalences, and a store of related work.

Meaning-preserving program transformations themselves have been used for reasoning about programs, to specify operational semantics for functional languages [22], for deriving efficient implementations from readable specifications at the source level [3, 4, 20]; for compiler optimisations at an intermediate language level [14] and for implementing functional languages directly according to their operational semantics [8].

We will exploit these foundations for our work: existing transformations for optimisation or derivational program development can provide guidance for the choice of refactorings and their decomposition; proving that refactorings are functionality-preserving should minimize the need for testing, and the program analyses needed for tool support should also profit from established theory.

Note however that refactoring is different from optimisation and program derivation. These 'vertical' transformations tend to be localised, addressing a program's control or data flow. The kind of program structure considered for refactoring is often non-localised and related to the overall program design and knowledge representation, i.e., to large-scale declarative aspects rather than smaller-scale operational ones.

Refactoring's 'horizontal' transformations address program structure, and as such can be applied equally well to high-level executable specifications as to more efficient lower-level programs.

Functional Refactoring

Program restructuring in imperative languages has to cater for the presence of side-effects, which weakens the theoretical basis available for reasoning about programs and complicates the analysis of programs, necessary to decide whether program transformations are applicable and functionality-preserving [11, 10, 17]. Work on refactoring in the object-oriented world over the last decade has focussed on program restructuring at a larger scale, taking into account the corresponding language features, such as objects and classes [18, 24, 6], but inheriting the same problems in program theory and analysis from the imperative core languages.

In pure functional languages, links to theory tend to be stronger and reasoning about programs less complicated than in imperative languages. At a small scale, it should be possible to import refactorings directly from the theory of functional languages. Even at this level, differences in programming style and language features (e.g., the wide-spread use of algebraic data types, pattern-matching, and higher-order functions in functional languages[12]) should become apparent in the catalogues of refactorings.

At a larger scale, language features supporting programming in the large differ completely, in that functional programming is inherently compositional, objects and object classes are absent, and type theory and logic exert a strong influence on language design. Type (inference) systems and type classes will have to be taken into account, as well as patterns for modular programming that build on general abstractions instead of specific language features (such as monads and monad transformers).

Design Changes characterise Design Patterns

A catalogue of refactorings effectively provides programmers with a list of pairs of program design patterns, together with discussions of their pros and cons, and instructions about how to transform one pattern into the other without introducing bugs. Work on refactoring can thus help to formalise and document design experience, furthering the exploitation of the results of functional programming research, by giving an extended answer to the question: "What constitutes good functional programming style, and why?".

Refactoring interacts with Language Design

Just as language design decisions are reflected in refactorings, program structures and the problems of manipulating them have an impact on language and program design.

A key example is that of *monads*, which represent an abstraction over various control-flow-related program aspects, such as threading data structures through a program, or implementing algorithms involving backtracking [26]. A program written in monadic style is easily extended to include more of the aspects covered by the monad abstraction, whereas retrofitting a monadic structure into a program that was designed without it is a major undertaking. Tool support for this monadic restructuring will allow programmers to choose a monadic structure on a by-need basis only, rather than conservatively adopting a monadic (and thus more imperative) style from the start.

Choice of Language

The interactions between refactoring and language design require some consideration in the choice of functional language to use. Haskell combines a rich base language with a rather simple module system, whereas the situation is just the other way round for languages in the ML family. In addition, Haskell strives for a clear separation between side-effecting and side-effect-free (pure) code, and features a non-strict semantics. To clearly separate our perspective from refactoring in object-oriented, imperative languages, we choose Haskell as our research vehicle, but we expect that many of the results will carry over to ML-like languages.

2.2 Prototype Tools

In our experience, functional programs tend to start small, as prototypes focussing on the critical aspects of problems, to get approximative results quickly. If one approach does not work, we would still like to reuse functionality for the next prototype, which will – almost by definition – employ a different program structure. Once an approach has shown itself to be feasible in principle, we then need to fill in the details for the less problematic aspects and bring the code into an acceptable shape for the next level of work. This involves identifying code patterns for reuse, capturing those patterns in (higher-order) functions and modules, and cleaning up the interfaces between the program parts. *Prototyping, fill-in-the-details, and design consolidation all tend to involve substantial, complex and wide-ranging program transformations, and as the code base grows in size, tool support becomes indispensable.*

Currently, we use programmable editors, which employ "syntax"-highlighting based on regular expression matching, but otherwise operate at the level of characters, words, lines, and paragraphs. We rely on static typechecking to flag up most of the errors that creep in during extensive non-local program manipulations, on short edit-compile cycles to fix simple typos and gaps, and on multi-level undo to be able to revert to the last-known-good version of our programs if a partially performed program transformation turns out to be incompletable after several steps.

This situation is unsatisfactory: practical program transformations consist of large numbers of individual source code manipulations, potentially distributed over the whole code base. Apart from the difficulties of keeping track of progress, goals and recovery points, the worst problem is that *programmers and compilers communicate via an unreliable low-level channel*: both programmers and compilers operate in terms of syntactically valid semantics entities, such as functions, expressions, types, and modules, whereas the tools currently used for source-code manipulation are almost entirely unaware of such concepts, so that any of the many text editing steps can introduce syntactic or semantic errors.

Language-aware refactoring tools can improve the situation by supporting "semantic editing": suppose we have a working prototype of a program and want to move some reusable functionality to a separate module; it should only be necessary to highlight the declarations involved and to issue a command *"move marked declarations to new module, named M"*. To make this work, the tool has to be aware of syntactic restrictions, variable scoping, type information, the very concepts of "declarations" and "modules".

The tool has to verify that the declarations to be moved are self-contained, create the necessary module header, export and import declarations, clean up the context from which the declarations are removed, and verify that the newly imported functionality will actually be visible at its original place and will not interfere with other definitions. Of course, the tool should refuse the operation with an instructive message if it cannot guarantee that the original functionality will be preserved. Actually moving the code is the least of the problems, and the only one solved by current tools.

Griswold [11, 10] characterises the main benefits of refactoring tools as follows: *engineers communicate intended restructurings at a high level, and the tools either infer and perform all necessary follow-on modifications to preserve the program functionality, or flag up the requested change as not implementable.*

Refactoring tools are a considerable improvement over syntax-aware editors (aka structure editors) and integrated development environments which are aware of static semantics, because they employ both static and dynamic semantics for non-local program manipulations. However, having worked with a syntax-directed editing environment for a functional language [16], we know that the pros and cons of such language-specific tools lie close together and that many programmers refuse to let their creativity be restricted to valid transformations and language-specific editors.

We thus imagine a combined approach, in which a syntax- and semantics-aware refactoring tool is used for code browsing and refactoring, while programmers are free to use their favourite general purpose editors for tasks not covered by our tool. We will avoid getting involved in editor wars or having to implement

extensive functionality not relevant to our project, and the dual user interface clearly distinguishes between safe, functionality-preserving refactorings and potentially dangerous free-form editing.

In his thesis [24, chapter 6], Roberts analyses the differences between his first, stand-alone refactoring tool ("While technically interesting, it was rarely used, even by ourselves.") and his more recent, highly successful Refactoring Browser [23]. He lists both technical and practical success criteria for a refactoring tool: maintaining a source code data base with easy access to accurate syntactic and semantic information (type of object under cursor, scope-aware search for identifiers, definitions and uses, etc.), speed of analyses and transformations/recompilations, support for recovery of last-known-good code version via undo, and tight integration of refactoring into standard programming environment.

Roberts' advice clearly outlines the issues we will have to address: unless we can reuse existing code, maintaining a data base of semantic information implies building the front-end of a Haskell compiler, including parsing, type checking and semantic analysis. The program transformations themselves will then have to be implemented on top of this. To achieve acceptable response times, the analyses should ideally be incremental, avoiding complete re-"compilations" for every small refactoring step.

3 Work Packages and Deliverables

The work packages are organised to cover the aspects of refactoring discussed in Section 2 while producing deliverables that match objectives I-V given in the project overview. The project will build on the initial groundwork we have done, but it will provide us with the resources to extend the scope and the depth of our work. We will investigate the issues of theory and tool support for functional refactoring in the concrete context of a catalogue of refactorings for the functional language Haskell.

Survey of Existing Work

We have started to collect references and URLs for relevant existing work, and this collection will need to be expanded and kept up to date throughout the project. *Expected deliverable:* an annotated bibliography, made available on the web, and covering publications as well as links to groups, projects and tools.

A Catalogue of Functional Refactorings

As a concrete focus for this project, we will develop a catalogue of refactorings for Haskell, deriving language-independent refactorings by generalisation wherever possible; this work is started in [25]. We have chosen Haskell because it is standardised and has several freely available implementations. It also has a large user community, ensuring widespread dissemination and evaluation of our research results, both inside and outside the academic community. Finally, it has reached a stage where the existence of libraries and tool support has led it to be used in applications of various sorts.

As in object-oriented languages, typical refactoring tasks will have to be identified and broken down into smaller steps, small enough to allow for convincing arguments that neither these elementary refactorings nor their composition changes the externally observable program functionality. Some of the elementary refactorings, such as adding or removing parameters in the definition and all applications of a function, are likely to carry over without change. However, the differences between object-oriented and functional languages are substantial enough to expect considerable differences in the catalogue of useful refactorings.

Methodology: we have experience of refactoring by hand, and by reflecting on this experience we will be able to generate initial examples and general classes of refactorings. Building on this, we expect to deploy a range of methods in gathering a library of refactorings, including the following.

- We will undertake surveys of other functional programmers, by means of the Haskell mailing list and the newsgroup `comp.lang.functional`.
- We will invite functional programmers working on substantial projects to keep logs of changes that they make to their code, and collate these logs into collections of refactorings.
- We will investigate existing change histories, such as the CVS repository for GHC, to ascertain what refactoring has taken place during the evolution of Haskell systems.
- In addition to continuously monitoring our programming practice, the researcher will undertake an initial, small-scale, case study and keep a record of the refactorings made during the exercise.
- We envisage using Haskell as the principal platform for building refactoring tool support. This activity itself will be monitored for candidate refactorings.

Expected deliverables: the catalogue itself as well as a commentary describing the problems of -and opportunities for- refactoring in Haskell. As discussed in Section 2.1, the catalogue will also go some way in documenting functional program design patterns and their pros and cons.

Theoretical Basis and Tool Support

While the catalogue provides the focus for this project, maintaining a solid theoretical basis and establishing prototypical tool support form the research core. The topic for this project is practice-driven, and so the project itself needs to be connected to programming practice. Implementing refactorings ensures that all relevant aspects, such as program analyses, specification of refactorings, etc., are formalised in a sufficient level of detail to be practical. We expect the work on theory and implementation to be strongly correlated (emphasised here by placing both in a single work package): the implementation will need to be based directly on theory and will thus stress-test any theoretical claims, and it will also point out areas were further theoretical work is needed.

Progress in this work package will depend on the availability of meta-programming infra-structure (at least parsing and pretty-printing) for Haskell. While practical experience with the suggested refactorings on real programs is considered essential for the success of the project, the development of a production-quality refactoring tool is not. The dual interface approach outlined in Section 2.2, complementing existing text editors with a separate refactoring tool, should give us a gradual route, both for implementation and for adoption of our tool.

Stage 0 (preparations): the elementary refactorings and their combinations need to be formalised. Side-conditions have to be established under which refactorings are demonstrably correct, and program analyses have to be devised to determine the validity of those conditions, to identify the program fragments affected by

a refactoring, and to compute complementary follow-on changes needed to preserve program functionality throughout a refactoring. This will overlap with the catalogue creation.

Stage 1 (initial tool support): in the beginning, few refactorings will be available, and programmers will operate mostly in their editor, switching to our tool only temporarily, for substantial refactorings. At this stage, the tool can be relatively simple, and will need to re-generate syntactic and semantic information on each invocation. When more refactorings are added, and the tool is used more frequently, the costs of re-generating information by processing source files again and again will become more and more of a problem. While this is unavoidable for changes initiated from outside the tool, it should be possible to predict the effects of refactorings initiated from within the tool.

Stage 2 (advanced tool support): there are two components to crossing this barrier: (a) the analyses performed by the tool will have to become incremental, so that refactorings can update the information instead of invalidating it. (b) While we have no hope or aim to replace the programmers' favourite editors, we need to make our refactoring tool capable and attractive enough to take on the lead role in the dual programming interface. Instead of the editor occasionally calling out to the refactorer, the refactorer has to be the main code inspection tool, calling out to the editor only for free-form editing tasks it is not designed to handle. The key to this switch of roles will be making the syntactic and semantic information known to the refactorer available to programmers – the refactorer can become a language-aware code inspection and visualisation tool. Re-analysis can then be limited to those fragments of source files that have been made available for external editing.

Expected deliverables: prototypical tool support for a suitable selection of refactorings of Haskell programs. Proofs that the refactorings are functionality-preserving. Documentation of the analyses, descriptions and other techniques employed in the implementation of the tool. User documentation.

Evaluation and Comparison

Once refactoring support for functional languages has been established to match that for object-oriented languages, we will be able to draw parallels or spot differences, and to investigate the novel interaction between refactoring and functional programming. For instance, program restructuring highlights problems with program structure and has initiated research into extensions of object-oriented languages. Will similar implications become apparent for functional language designs?

Other questions include: Is Haskell (and particularly its type system) suited for refactoring, or what changes could be suggested? How does existing work on flexible program structure in functional languages fit into the picture? Does type inference help program analysis for refactoring or does typing make program analysis more complex? What additional refactorings does the advanced type system suggest? Are our tools being accepted and used by the Haskell community?

Expected deliverables: Discussions with the functional programming community at large, and especially with our contacts at Microsoft Cambridge (UK), Oregon (US), and York (UK). Documentation of specific issues throughout the project, overall review at the end.

Workplan, Beneficiaries, Dissemination

A detailed graphical workplan is attached to the case for support. The project group (Reinke, Thompson, researcher) will work collaboratively on objectives I, II, IV and V. The researcher's main task will be to fulfill objective III; the investigators will supervise this activity.

As functional languages have developed into widely used tools for research, teaching and prototyping, any deliverables for objectives II and III can be expected to be disseminated and exploited in this community.

Specific beneficiaries nationally would include our colleagues in York and Cambridge and, internationally, the Pacific Software Research Center in Oregon (US). All groups are involved in substantial Haskell implementation projects, as well as the development of tool support and applications.

Work on objectives II, III, and IV will feed into objectives I and V, and further work will branch off from there. Both investigators hope to exploit the results of this project for their related research interests in language design, software engineering, and functional programming in research and education. UKC's computing lab has strong interests in software and systems engineering, and is involved in industry cooperations (OMG, pUML), so we expect lab-internal discussions and dissemination of results as well.

Bibliography

[1] The Haskell Home Page. `http://www.haskell.org`, language definition, implementations, libraries, tool.

[2] The LOLITA system as a contents scanning tool. In *Proceedings of the 13th International Conference on Artificial Intelligence, Expert Systems and Natural Language Processing, Avignon*, 1993.

[3] R. M. Burstall and J. Darlington. A Transformation System for Developing Recursive Programs. *Journal of the ACM*, 24(1):44–67, 1977.

[4] J. Darlington. Program Transformations. In J. Darlington, P. Henderson, and D. A. Turner, editors, *Functional Programming and its Applications*, pages 193–215. Cambridge University Press, 1982.

[5] M. Fowler. The New Methodology. `http://www.martinfowler.com/articles/newMethodology.html`.

[6] M. Fowler, K. Beck, J. Brant, W. Opdyke, and D. Roberts. *Refactoring: Improving the Design of Existing Code*. Addison-Wesley, 1999. see also: `http://www.refactoring.com/`.

[7] Galois Connections Inc. `http://www.galconn.com/`.

[8] D. Gärtner and W. Kluge. π-RED$^+$: An interactive compiling graph reduction system for an applied λ-calculus. *Journal of Functional Programming*, 6(5), Sept. 1996.

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

[10] W. G. Griswold. *Program restructuring to aid software maintenance*. PhD thesis, University of Washington, Dept. of Computer Science and Engineering, 1991. Tech. Rep. No. 91-08-04.

[11] W. G. Griswold and D. Notkin. Automated Assistance for Program Restructuring. *ACM Transactions on Software Engineering and Methodology*, 2(3):228–269, July 1993.

[12] P. Hudak. Conception, Evolution, and Application of Functional Programming Languages. *ACM Computing Surveys*, 21(3):359–411, Sept. 1989.

[13] W. Hürsch and C. V. Lopes. Separation of Concerns. Technical Report NU-CCS-95-03, Boston, Massachusetts, 1995.

[14] S. P. Jones. Compiling Haskell by program transformation: a report from the trenches. In *European Symposium on Programming (ESOP'96)*, Apr. 1996.

[15] G. Kiczales, J. Lamping, A. Mendhekar, C. Maeda, C. Videira, J.-M. Loingtier, and J. Irwin. Aspect-Oriented Programming. In *ECOOP'97*, June 1997. see also: `http://www.parc.xerox.com/csl/projects/aop/`.

[16] W. E. Kluge. A User's Guide for the Reduction System π-RED$^+$. Technical Report 9419, Institute of Computer Science and Applied Mathematics, Christian-Albrechts-University, Kiel, Dec. 1994. `http://www.informatik.uni-kiel.de/~base/`.

[17] J. D. Morgenthaler. *Static Analysis for a Software Transformation Tool*. PhD thesis, University of California, San Diego, 1997.

[18] W. F. Opdyke. *Refactoring Object-Oriented Frameworks*. PhD thesis, University of Illinois, 1992.

[19] H. Ossher and P. Tarr. Multi-Dimensional Separation of Concerns and the Hyperspace Approach. In *Symposium on Software Architectures and Component Technology: The State of the Art in Software Development*. Kluwer, 2000. see also `http://www.research.ibm.com/hyperspace/`.

[20] H. Partsch and R. Steinbrüggen. Program transformation systems. *ACM Computing Surveys*, 15(3), Sept. 1983.

[21] L. C. Paulson. *Isabelle – A Generic Theorem Prover*. Springer-Verlag, Lecture Notes in Computer Science, 828, 1994.

[22] G. D. Plotkin. Call-by-name, call-by-value and the λ-calculus. *Theoretical Computer Science*, 1:125–159, 1975.

[23] D. Roberts, J. Brant, and R. Johnson. A Refactoring Tool for Smalltalk. *Theory and Practice of Object Systems (TAPOS), special issue on software reengineering*, 3(4):253–263, 1997. see also `http://st-www.cs.uiuc.edu/users/brant/Refactory/`.

[24] D. B. Roberts. *Practical Analysis for Refactoring*. PhD thesis, University of Illinois at Urbana Champaign, 1999.

[25] S. Thompson and C. Reinke. A Catalogue of Functional Refactorings, Version 1. `http://www.cs.ukc.ac.uk/people/staff/sjt/Refactor/`.

[26] P. Wadler. The essence of functional programming. In *POPL '92, Albuquerque*, 1992.

[27] P. Wadler. Functional Programming: An angry half-dozen. Column, SIGPLAN Notices 33(2):25-30, Feb. 1998. see also `http://cm.bell-labs.com/cm/cs/who/wadler/papers/sigplan-angry/sigplan-angry.ps.gz` and `http://cm.bell-labs.com/cm/cs/who/wadler/realworld/`.

[28] J. D. Wells. Extreme Programming, A Gentle Introduction, Apr. 2001. `http://www.extremeprogramming.org`.