Tool Support for Refactoring Functional Programs

Huiqing Li
H.Li@kent.ac.uk

Claus Reinke
C.Reinke@kent.ac.uk

Simon Thompson
S.J.Thompson@kent.ac.uk

Computing Laboratory, University of Kent

ABSTRACT

Refactorings are source-to-source program transformations which 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 enabled the trend towards modern agile software development processes. Refactoring has taken a prominent place in software development and maintenance, but most of this recent success has taken place in the OO and XP communities.

In our project, we explore the prospects for *'Refactoring Functional Programs'*, taking Haskell as a concrete case-study. This paper discusses the variety of pragmatic and implementation issues raised by our work on the *Haskell Refactorer*. We briefly introduce the ideas behind refactoring, and a set of elementary functional refactorings. The core of the paper then outlines the main challenges that arise from our aim to produce practical tools for a decidedly non-toy language, summarizes our experience in trying to establish the necessary meta-programming infrastructure and gives an implementation overview of our current prototype refactoring tool. Using Haskell as our implementation language, we also offer some preliminary comments on Haskell programming-in-the-large.

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; D.3.4 []: Processors

General Terms

Languages, Design

Keywords

Haskell, refactoring, program transformation, language-aware programming environments, semantic editors

1. INTRODUCTION

Refactoring is about 'improving the design of existing code' and as such, it has been practised as long as programs have been written. The key characteristic distinguishing refactoring from general 'code meddling' is the focus on structural changes, strictly separated from changes in functionality. The benefit of this separation is that refactoring does not introduce (nor remove) bugs or invalidate any tests, while changes in functionality are disentangled from structural reorganisations. The steps and potential pitfalls involved in refactoring are documented in catalogues of useful refactorings, which are supported by tools or validated by testing. Within the software engineering and object-oriented programming communities, refactoring has been identified as central to agile software maintenance and development processes [9, 7, 23].

In our project 'Refactoring Functional Programs' [1], we seek both to complement existing work on refactoring with a functional programming perspective and to make refactoring techniques and tools available to functional programmers.

The idea of continuous design improvement. In his 1978 (!) ACM Turing Award Lecture [6], Robert Floyd argued that serious programmers should spend part of their working day examining and refining their own methods: "After solving a challenging problem, I solve it again from scratch, retracing only the *insight* of the earlier solution. I repeat this until the solution is as clear and direct as I can hope for. Then I look for a general rule for attacking similar problems, that *would* have led me to approach the given problem in the most efficient way the first time.". Identifying "paradigms of programming" in this way – and developing support for such paradigms – would improve programmer abilities, computer science teaching and learning, and language designs.

In practice, industrial software development projects will not be restarted from scratch when they have already reached their prime objective – working code. Nevertheless, the idea of continuous design improvements finally became attractive and feasible because of: (a) the increasing pressure of maintaining the design quality of long-living "legacy software" in spite of large numbers of modifications, bug-fixes and functionality extensions, and (b) the realisation that the necessary redesign could be achieved in an incremental fashion, by employing *program transformations*.

Keeping software soft and malleable. Adapting program transformations that originated in derivational (or transformational) program development [1, 2] for languages with side-effects, Griswold introduced the idea of automated program restructuring to aid software maintenance [9, 8]. The techniques were extended to cover object-oriented language features, and have since come to prominence in the OO and extreme programming (XP) communities [7, 23] under the name of *refactoring*.

Functional program transformations have a long history [24], e.g., for deriving programs from specifications, or in optimising

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. To copy otherwise, to republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee.
Haskell'03, August 25, 2003, Uppsala, Sweden.
Copyright 2003 ACM 1-58113-758-3/03/0008 ...$5.00.

[1] `http://www.cs.kent.ac.uk/projects/refactor-fp/`

```
showAll :: Show a => [a] -> String
showAll = table . map show
    where
    format :: [String] -> [String]
    format []     = []
    format [x]    = [x]
    format (x:xs) = (x ++ "\n") : format xs

    table :: [String] -> String
    table = concat . format
```

Figure 1: The initial program

compilers, and so it is somewhat surprising to see this particular use of program transformations almost exclusively limited to OO languages. It seems that the problems of inflexible program structures have been more pressing in OO languages, triggering the need for complex program manipulation techniques. What is not clear is whether the cause of this is a relative inflexibility of OO programming constructs or the larger-scale use of OO languages in practice. Shedding more light on the differences in the way program transformations are used in the OO and FP communities has been one motivation for our project.

Problems of scale, complexity, and useability. The present paper focusses on the pragmatic aspects of trying to provide tool support for meaning-preserving program transformations in modern functional languages, which are used both as object and implementation languages. For concreteness, we focus on Haskell throughout, but at this stage, most of the problems and solutions should apply equally to other functional languages.

The 'tool support' of the title refers both to our current prototype of the *Haskell Refactorer* and to the tools that make its implementation feasable. We discuss our approach to the problems of extracting the syntactic and semantic information needed by the Refactorer, implementing meaning-preserving program transformations over Haskell's non-trivial and many-typed abstract syntax, preserving the original program layout and comments, and integrating our tool with standard Haskell programming environments.

In the following sections, we first introduce the main ideas behind refactoring and its role in functional software development, and identify a set of elementary functional refactorings that our current tool implements. We then outline the major challenges inherent in providing program transformations for a real-world functional language as complex as Haskell, and provide a summary of our experience in trying to establish the necessary meta-programming infrastructure. Then we give an overview of the implementation of our current prototype refactoring tool, built on top of that infrastructure and interfaced to the two popular programmers' editors Emacs and Vim; we conclude by examining related work and looking at future developments in our project.

2. FUNCTIONAL REFACTORING

2.1 First examples

Figures 2–4 illustrate a variety of 'structural' refactorings in Haskell, performed on the program presented in Figure 1. The examples presented here are necessarily small-scale, but they are chosen to illustrate aspects of refactoring which scale to larger programs and multi-module systems. It would be possible to reduce the example

```
-- First move format to the top level ...

showAll = table . map show
    where
    table = concat . format
format []     = ...

-- ... then move table to the top level.

showAll = table . map show
table   = concat . format
format []     = ...
```

Figure 2: Elementary scope refactoring 1

```
-- First make format local to table ...

showAll = table . map show
    where
    table = concat . format
        where
        format []     = ...

-- ... then move table to the top level.

showAll = table . map show
table   = concat . format
    where
    format []     = ...
```

Figure 3: Elementary scope refactoring 2

programs to one-liners by judicious use of library functions, but we avoid that for the sake of demonstrating the refactoring examples.

In presenting the examples we elide with '...' those parts of a program that are unchanged. The types of functions are initially given explicitly, but are omitted except when they change.

The function `showAll` has been developed to build a 'table' by `show`ing each line, and then interleaving the results with a newline. Once completed, the program can be developed in several ways, starting by making local functionality available in a wider scope.

Figures 2–4 give three variations on lifting the definition of `table` to the top level. In the original program, `table` is defined locally to `showAll`, and depends upon `format` from the same `where` clause.

promote in dependency order: In Figure 2, `format` is 'promoted' to the top level first, and `table` secondly. This shows two examples of lifting local definitions to the top level.

package dependencies, then promote: In Figure 3, `format` is first 'demoted' to being a local definition of `table`; the definition of `table`, complete with its `where` clause, can then be promoted. (It would be possible to follow this by promoting `format` to the top level.)

abstract over dependencies, then promote: Finally, Figure 4 shows that format can be made a parameter of `table` to break the dependency and enable promotion.

```
-- Move table to the top level; it needs to take
-- format as a parameter.

showAll = table format . map show
    where
    format :: [String] -> [String]
    format []     = ...

table :: ([String] -> [String]) -> [String] -> String
table format = concat . format

-- Rename the formal parameter for readability.

showAll = table format . map show
    where
    format []     = ...
table fmt = concat . fmt

-- Move format to the top level.

showAll = table format . map show
format []     = ...
table fmt = concat . fmt

-- Unfold (inline) the definition of table

showAll = (concat . format) . map show
format []     = ...
table fmt = ...

-- Remove the definition of table

showAll = (concat . format) . map show
format []     = ...

-- Define table to be (concat . format)

showAll = table . map show
table = concat . format
format []     = ...
```

Figure 4: Refactoring with type change

> In the remaining steps the formal parameter `format` is renamed to `fmt`; the definition of `format` is promoted to the top level and, within `showAll`, the application of the function `table` is inlined. This has the effect of removing the only use of `table`, so its definition can be removed, and then `table` can be reintroduced with its orginal meaning[2].

In Figure 5 further refactorings are shown in a typical program development scenario. First the separator is made a parameter of `format` and subsequently of `table`, generalising their types.

A copy of `showAll`, named `makeTable`, is then generalised over both the separator and the `show` function. The result of this is a higher-order, polymorphic, function, which is ripe for reuse.

Finally, `showAll` is made an instance of the more general function `makeTable`, in an example of a *fold* transformation.

[2]The last three steps could be composed to form an 'inline parameter' refactoring.

```
-- The refactored program, typed.

showAll :: Show a => [a] -> String
showAll = table . map show

format :: [String] -> [String]
format []     = []
format [x]    = [x]
format (x:xs) = (x ++ "\n") : format xs

table :: [String] -> String
table = concat . format

-- Stage 1: make "\n" a parameter of format.
...
format :: [a] -> [[a]] -> [[a]]
format sep []     = []
format sep [x]    = [x]
format sep (x:xs) = (x ++ sep) : format sep xs

table = concat . format "\n"

-- Stage 2: now make "\n" a parameter of table

showAll :: Show a => [a] -> String
showAll = table "\n" . map show

format :: [a] -> [[a]] -> [[a]]
format sep []     = ...

table :: [a] -> [[a]] -> [a]
table sep = concat . format sep

-- Stage 3: copy showAll, calling it makeTable;
-- generalise by making show and "\n" parameters.
...
makeTable :: (a -> [b]) -> [b] -> [a] -> [b]
makeTable trans sep = table sep . map trans

-- Stage 4: make showAll an instance of makeTable.

showAll :: Show a => [a] -> String
showAll = makeTable show "\n"
...
```

Figure 5: Refactoring for generalisation

2.2 Reflecting on the examples

Refactoring steps can be applied manually, perhaps with a checklist for each kind of refactoring, or with partial tool support through editor macros and test suites, or they can be automated, as in our *Haskell Refactorer*. Even in these small examples, it is clear that refactoring moves beyond mere indentation control or textual search-and-replace: the transformation steps should *preserve program functionality*, they may be *valid only under certain side-conditions*, they tend to be *semantics-based*. It is also clear that refactoring could profit from language-aware *tool support*, provided such support is *integrated into standard development environments*.

Preserving functionality. This can mean a variety of things:

- strictly no change in the semantics of the program;
- no *observable* change, from the point of view of a well-defined interface, such as the `main` program. This latter allows functionality to be added, as in the example of generalising `showAll` to `makeTable`.

Side-conditions. Consider the scenario of Figures 1–4: it is not possible simply to move the definition of `table` to the top level unchanged, because it depends on `format`. The validity of attempted refactoring steps can be checked *before* or *after* each transformation, and can often be ensured by *compensating* for problems, turning incomplete transformations into valid refactorings:

Postcondition. Perform the transformation, then test whether the program functionality has changed. In the scenario, this will cause the program to fail to compile; if there were also a definition of `format` at the top level, then the meaning of the program might be modified. A good test suite covering the refactored program might flag up such problems.

Precondition. Reject transformations unless their validity as refactorings can be ensured in advance; this conservatism is safe, but not very helpful, leading to longer refactoring sequences. It is easy to miss some interactions when checking complex preconditions by hand.

Compensation. Starting from a local transformation, infer the intended global refactoring, and compensate for any problems, to ensure validity. In the promotion scenario, any of the three variations shown would do. This is the most flexible, but also the most error-prone choice (not recommended for manual refactoring, but very convenient in refactoring tools).

Of these options, the 'postcondition' approach is not very desirable, but testing is mandatory as a safety net in by-hand refactoring. Verifying preconditions, or weakening preconditions where suitable compensations can be inferred systematically, are more favourable, but difficult to manage and get right without tool support.

Semantics-based transformations. Refactorings are not simply syntactic; many require awareness of the static semantics of the program, of static program analyses, and of the type system.

The examples highlight several reasons why refactoring 'by hand' in one of the standard Haskell program development environments – Vi or Emacs – is not a straightforward matter.

- In renaming the *parameter* `format` to `fmt` it is important to change only those occurrences of `format` and not occurrences of the *function* of the same name. Thus, refactoring tools need to be aware of the *static semantics* of the language.
- In lifting `table`, it is necessary to check its dependencies: this requires analysis of the *free variables* of the definition, and how they relate to the environment in which it occurs.
- Moreover, in this case the type of the function `table` is modified: if the program contains a type declaration for `table` then it has to be amended. A refactoring tool therefore requires access to *type information*.

Tool support. Although a refactoring may appear to be trivial, it is usually tedious and error-prone to put it into effect by hand. Take again the case of renaming the function `format`: it is necessary to identify all and only those occurrences of that function, and not to rename, for example, the parameter `format`. Renaming a function will require editing of the module in which the function occurs and – potentially at least – all the modules which import this module.

Thus, in order to refactor successfully, there needs to be a high level of automated support. The refactorings seen in the examples are implemented in our prototype toolkit.

Tool integration. Refactorings are source-level transformations.

Integration. It is vital that a refactoring tool is an integral part of standard program development environments. According to our survey on the Haskell mailing list[3], tools have to be integrated with the Vi and Emacs families of editors.

Layout. For a refactoring tool to be useful, it must respect the layout style of the source; this requires the toolkit to keep source location information, *and to be able to transform this to maintain consistency of style of transformed definitions*.

Comments. Comments need to be retained; in many situations the comments themselves need to be refactored, in line with the program. Automatic support for comment refactoring is problematic in general; it may be possible to provide limited support within a documentation system such as Haddock.

2.3 Elementary refactorings

This section introduces the refactorings in our prototype tool. The catalogue on our project home page gives other examples.

Practically motivated refactoring candidates tend to be complex combinations of simpler components. To avoid duplication of functionality and to provide for flexibility in recombination, it is essential to break such complex transformations into indivisible components from which more complex refactorings can be built.

For example, instead of automatically generating fresh names when lifting definitions, they can be renamed prior to lifting to avoid any clashes during lifting, keeping programmers in control and offering renaming as a refactoring in its own right. This search for elementary refactorings is sometimes in conflict with the idea of automatically compensating for programmer-initiated changes, especially as we do not yet support composite refactorings. These details are expanded in the discussion below.

Each of the refactorings has *side conditions*, which emphasises the fact that these are more than textual manipulations. In the group of refactorings presented here the prime conditions concern the *static semantics* of the program: the scope of definitions, the binding structure of the program (*i.e.* the association of the use of an identifier with its definition), the uniqueness of definitions in each scope and so forth.

Renaming (α-conversion)

Any program identifier can be renamed, as in Figure 4 where `fmt` replaces `format`. It is important to ensure that all and only the uses of one binding are renamed.

Conditions. The existing binding structure must not be affected. No binding for the new name may exist in the same binding group. No binding for the new name may intervene between the binding of the old name and any of its uses, as the renamed identifier would be *captured* by the renaming. The binding to be renamed must not intervene between existing bindings and uses of the new name.

[3] July 2002; results available from our project home page

Duplication

Any definition can be duplicated, at the same level as the original definition. A name for the duplicated definition needs to be supplied; alternatively a fresh name can be generated.

Conditions. For conditions on the new name see renaming.

Deletion

Any unused definition can be deleted.

Conditions. The condition on deletion is that there are no uses of the name whose definition is to be deleted.

Promote one level

A definition is lifted from a `where` or `let` into the surrounding binding group. Such lifting widens the scope of the definition.

Conditions.

- Widening the scope of the binding must not capture independent uses of the name in the outer scope.
- There should be no existing definition of the name in the outer binding group (irrespective of whether or not it is used).
- The binding to be promoted must not make use of bindings in the inner scope.

The third condition is frequently violated in practice, so we compensate by *lambda lifting* the definition to turn such uses of inner bindings into formal parameters[4]. All the uses of the name have to be modified to take the extra actual parameters. Two conditions apply because of this.

- The binding must be a simple binding of a function or constant, rather than a pattern binding.
- Any argument must not be used polymorphically; recall that whilst it is possible to use a defined name at two different (incompatible) instances, the same is not true of an argument. Note that to enforce this constraint it is necessary to have access to type checking facilities.

Demote one level

A definition is demoted to a local scope given by a `let` or `where`.

Conditions.

- All uses of the binding to be moved must be within the inner scope to which the definition is being moved.
- Free variables in the binding must not be captured when it is moved over nested scopes.
- Moreover, there should be no definition of the name in the inner binding group (irrespective of whether or not it is used).

Add an argument

Parameters can be added to definitions of constants and functions. This is one way to prepare for generalisation of a definition, followed by specialisaton of its uses. *Generalise a definition* is often the preferred alternative.

Conditions. No variable uses may be captured by the new formal parameter. Default values for actual parameters must be supplied and added to all uses of the definition. Adding those defaults must not introduce errors to the program.

[4] This temporarily deviates from the ideal of elementary refactorings, as the extra parameters could be introduced in a separate step, or other compensation could be used. This will become a composite refactoring once our tool supports those.

Remove an argument

Any unused argument to a definition can be removed.

Conditions. There must be no uses of the formal parameter to be removed. All uses of the definition must be adapted to remove the corresponding actual parameters.

Generalise a definition

A definition is generalised by selecting a sub-expression of the right-hand side of the definition and introducing that sub-expression as a new parameter.

Conditions. No variable uses may be captured by the new formal parameter. Since the chosen expression becomes a parameter to the function at each of its call sites, that expression must be defined – and have the same value – at each of these sites:

- Any free names in the selected expression should be bound to names visible at the level of the definition itself (*i.e.* not be formal parameters or be defined locally to the definition).
- Moreover, the same bindings need to be visible at all call sites of the function, which ensures that the expression has the same meaning at each call site.

Inline a definition (unfold)

The application of a function is replaced by the right-hand side of the definition, with actual parameters replacing formals.

Conditions. Single-equation functions (or constants) can simply be inlined. For a more complicated function, pattern matching may have to be replaced by a `case` statement, for example.

Name capture of free variables in right-hand side or actual parameters must be avoided when unfolding the right-hand side and substituting actuals for formals at the call site.

Introduce a definition[5]

A definition is introduced to denote an identified expression.

Conditions. The new definition must not disrupt the binding structure of the program. Moreover, the free names in the right-hand side of the new definition must be bound to the same definitions as they were in the original context.

2.4 Interactions with language design

The small set of elementary refactorings presented here focusses on the functional core of Haskell: names, bindings (parameters and definitions) and their uses. At this level, specifics of Haskell's design do not seem to have an influence, but that appearance is misleading. Alert readers will have noticed that some of these seemingly innocent refactorings need careful adaptation for Haskell, to take issues such as polymorphic vs monomorphic bindings and the monomorphism restriction into account, increasing the dependence of refactoring tools on type information.

Adapting the language design principle of *correspondence* [29] to refactorings suggests that for each refactoring of definitions, there should be a corresponding refactoring of parameters, and vice versa. Such a correspondence is indeed beginning to emerge, and it will be useful to look into the cases not yet covered, but it is necessary to keep in mind that, partially due to its type system, Haskell does not comply with those language design principles (see above; rank-n types and explicit type annotations provide work-arounds).

[5] Combined with 'Generalise a definition', this provides part of the functionality of 'fold' transformations. The missing part involves folding instances of existing definitions. Once we fully support fold, we will have to address the issue of total correctness.

3. TOOLING UP FOR REFACTORING OF FUNCTIONAL PROGRAMS

Refactoring tools support interactive program manipulation – they are used as part of the editing process, but operate on syntactic and semantic information, not on character strings. Continuing the evolution from plain text editors to syntax- and semantics-aware editors and IDEs, they share many of the characteristics of optimising compilers: source texts need to be parsed to extract abstract syntax trees from concrete syntax strings; programs need to undergo static analysis to determine, e.g., the scope of identifiers, and type analysis to extract type information (initially mostly to update existing type annotations, later to guide type-level refactorings).

Then, the real work of refactoring begins: based on abstract syntax trees annotated with static semantics information, analyses have to be implemented to validate the side-conditions of refactorings (which are meaning-preserving only if those side-conditions are met) and to compute the global changes that are needed to compensate for the local changes initiated by the tool user. For example, adding a parameter to a function definition requires follow-on changes throughout the program wherever that function is applied.

Finally, the program transformations themselves have to be implemented and – unlike in traditional compilers – the modified abstract syntax trees have to be presented to the programmer in concrete syntax form. This seemingly minor issue will turn out to have a major impact on our implementation design, as we discuss below.

For proper refactoring support, a complete compiler frontend is needed, plus an analysis and transformation engine similar to those employed in optimising compilers. Last, but not least, the results have to be presented to the programmers, and the whole refactoring process needs to be integrated with and interactively controlled via their favourite development tools.

Given that feature-rich modern functional languages have come a long way since the days of Lisp S-expressions [12], it quickly becomes clear that implementing tool support for refactoring programs in such languages requires tool support itself.

3.1 Meta-Programming I: Information

Ideally, tool-builders and other meta-programmers would reuse the functionality already present in the language implementations, via a well-defined standard reflection API such as the ADA Semantic Interface Specification ASIS[6]. Unfortunately, meta-programming support in functional languages like SML and Haskell has not kept pace with their overall development. If it exists at all, it tends to be relegated to implementation-specific extensions (SML/NJ's `Compiler` structure, or GHC's template meta-programming).

As functional language implementations tend to be bootstrapped, extracting the relevant code from the functional implementation seems to be the next best bet, until one sees just how implementation-specific and interwoven with other components that code tends to be. For our concrete example of Haskell, earlier efforts had produced at least reusable parsing and pretty-printing libraries, but for the more complete kind of front-end needed for refactoring, the tools we need have only just started to become available.

3.1.1 Concrete and Abstract Syntax

The best known Haskell frontend is probably the `haskell-src`[7] package in the hierarchical libraries (also known as hsparser or hssource), which comprises abstract syntax data types and support for parsing and pretty-printing of Haskell 98 code. A variant of the parser is used in the Haddock Haskell documentation tool [22]. This tends to track GHC's extensions to Haskell, and has been used successfully in a companion project on Haskell metrics [27].

In an early prototyping exercise, we implemented a few refactorings with syntactic information only, building on no more than the Haddock parser and the `haskell-src` pretty-printer. We soon had to add our own static analysis to determine the scope of variables, and then we would have had to add our own type analysis for refactorings interacting with Haskell's complex type system. Dealing with these non-trivial aspects of Haskell would have delayed work on the core of our project substantially, so we kept looking for more complete Haskell-in-Haskell frontends. We also noticed that we had to write (and later maintain!) large amounts of 'boilerplate' code for each refactoring – code that had to spell out 'obvious' aspects of abstract syntax traversal in painful detail, so we started to look for better meta-programming alternatives in general.

3.1.2 Adding Types and Static Semantics

The most popular approach towards such more complete Haskell frontends has been to combine `haskell-src` with Mark Jones' "Typing Haskell in Haskell" (thih) [13]. We compared Hatchet[8], and a snapshot of the as yet unreleased *Programatica*[9] frontend, again by prototyping a small number of refactorings. It soon became clear that more effort had been invested in the *Programatica* frontend, but Hatchet remained attractive for its simplicity.

Hatchet, in version 0.1 (released May 2002), offers access to abstract syntax, as well as type and kind information (a slight modification was needed to get type information for locally declared variables), and information about the predefined class hierarchy and instances. It lacks support for multi-module programs, user-defined classes and instances, and record syntax.

Programatica's frontend, in a snapshot dating from October 2002, offers almost complete support for Haskell 98 (plus a small number of extensions). While inspired by the original ideas of hsparser and thih, additional thoughts have been put into many aspects of the system. Examples of such extensions include dealing with the gap between assumptions in thih and Haskell's static analysis and module system [3], replacing the lexer to preserve more information about source programs [11], replacing the abstract syntax with a parameterized version supporting syntax variants and extensions, and providing for (limited forms of) generic programming [28].

Not all of the additional features of *Programatica*'s frontend are beneficial for our own project, but the completeness and ongoing development of *Programatica* provided the decisive advantages.

3.2 Meta-Programming II: Transformation

The result of parsing programs will be some representation of their abstract syntax trees in terms of a collection of mutually recursive types reflecting the structure of the language grammar. Now, writing compilers and other program processing tools is often cited as a shining example for the advantages of functional programming, but we have found that this argument tends to ignore issues of scaling. Naively applying standard functional programming techniques to a collection of data types representing the Haskell grammar leads to impractically large amounts of boilerplate code.

The cause of the trouble is the large number of algebraic types, each being a sum of a large number of constructors, many of which may recursively contain elements of types representing other parts of the abstract syntax. A naive implementation of a simple recursive traversal replacing each constructor by itself would consist of a function for each type, with a case for each constructor in that type,

[6] `http://www.acm.org/sigada/wg/asiswg/`

[7] `http://haskell.org/ghc/docs/latest/html/haskell-src/`

[8] `http://www.cs.mu.oz.au/~bjpop/hatchet.html`

[9] `http://www.cse.ogi.edu/PacSoft/projects/programatica/`

and since the function call graph would mimic the recursive grammar, these functions would be so heavily entangled that they could not be reused in isolation. So each new kind of traversal would require an effort reflecting the size of the whole grammar, even if the purpose of the traversal was just to rename all occurrences of an identifier in a certain scope.

An obvious improvement – defining generalised higher order traversal operators, such as fold and map, which could then be instantiated for more specific purposes – is slightly impeded by the complex and many-typed nature of the abstract syntax. Fold and map can be defined generically, and a variety of Haskell extensions permit datatype-specific instances of generically defined functions to be generated automatically [4, section 3.4], so the use of general higher-order traversal operators can be translated to the problem at hand. However, generic programming support, while necessary, is not sufficient to reduce our problem to practical dimensions: instantiating a recursive nest of folds over each of the parts of the grammar is still proportional to the size of the grammar.

One way to attack this problem, and move towards composable traversals, are updateable fold-algebras [21], where for each type, the higher-order parameters to its fold operation are collected in a record, and default records specifying trivial traversals (generated using generic programming techniques) only need to be updated with non-trivial functions for those constructors relevant to a specific traversal (e.g., those related to variables).

The next problem is the organisation of transformation code. Here, we wanted to profit from work on rewriting strategies, e.g., for program transformations in optimising compilers [32, 31]. Strategic programming separates recursively applied transformations into local rewrite rules and global reduction strategies, aiming to provide both as elements of a domain-specific language for composing transformations. In functional languages, both rewrite rules and strategies can be supported as first-class functions, so a function implementing a local rewrite rule can be passed to another function implementing a bottom-up reduction strategy, and finally the combination can be applied to a complex data structure.

The *Strafunski* project [18] has recently translated the ideas of strategic programming to statically typed frameworks. Design experiments were still in full swing when our Refactoring project started, but the API for the library of strategic programming combinators appears to have stabilised now and isolates us somewhat from ongoing developments in the underlying implementation technologies [17]. Combining generic and strategic programming in a pragmatic way, *Strafunski* aims to scale the promises of functional programming over algebraic types to the complexities of realistic programming language grammars, listing functional transformations of COBOL programs as a major applicaton area and recently even experimenting with language-parametric refactorings [16].

We have found *Strafunski* an indispensible tool for our purposes. It successfully addresses the issues of generic programming, reuse, and succinct specification of complex transformation schemes, and by freeing us from unmaintainable amounts of boilerplate code it even reduces our dependence on the choice of Haskell frontend.

Programatica comes with support for a small selection of generic traversal operations, defined explicitly as type class instances for all types involved in the abstract syntax representation. To limit the detrimental effects of recursion on reuse, the data type definitions follow a 2-level scheme which splits recursive algebraic types into separate parts describing structure and recursion (Sheard [28] describes the basic techniques as well as the use of general traversal operations to define a generic unification algorithm).

While mostly successful within *Programatica* itself, the coding of generic functions using only type classes leads to large amounts of boilerplate code for the general traversal functions. That code does not have to be repeated for each specific traversal, but is sensitive to changes in grammar or traversals. Also, the 2-level approach to datatype definitions substantially complicates the types representing Haskell's abstract syntax, compared with `haskell-src`.

Currently, we avoid any explicit use of *Programatica*'s traversal support in favour of *Strafunski*'s StrategyLib, which also helps to isolate us from the complexities of *Programatica*'s data types.

3.3 Meta-Programming III: Interaction

Refactoring tools provide source-level assistance for programmer-guided software redesign and thus have to be integrated into the program development process in as seamless a way as possible.

In his thesis [26, 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 for Smalltalk [25]. He lists 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.

Based on the insight that refactoring tools continue the evolution towards semantics-aware editors and IDEs, it would have been very tempting to implement such an IDE from scratch, including a fully fledged syntax-aware editor for Haskell. However, the substantial efforts involved in such an adventure would not only have distracted from our current project's main research questions, such an approach would also be questionable for pragmatic reasons.

Placing our refactoring tool out of current development tool chains would have involved us in editor wars at best and condemned our tool to irrelevance at worst. So we decided to design our *Haskell Refactorer* to be independent of a specific GUI, in favour of a generic textual API and bindings to this API for two of the most popular programmer's editors, Vim and Emacs [10]. According to our survey, these two editor families cover the vast majority of Haskell programmers' development environments.

4. IMPLEMENTING A REFACTORING TOOL FOR FUNCTIONAL PROGRAMS

Developing a refactoring tool for a real-world popular functional language not only provides us with a framework for exploring the prospects of refactoring functional programs, but also makes refactoring techniques and tools available to functional programmers.

Our current prototype *Haskell Refactorer* is built on top of *Strafunski* and *Programatica*'s frontend, is integrated with Emacs and Vim, and supports a small set of basic refactorings. The tool is straightforward to use, and more importantly, it preserves both comments and layout style of the source. The refactorings are so far only supported within a single module and are not supported by type analysis, but implementing them has been essential in identifying technical problems, finding suitable supporting tools and establishing the current implementation architecture. In this section, we give a brief overview of our tool, including its interface and its implementation architecture and techniques.

4.1 The Interface

Figure 6 shows a snapshot of the current Haskell refactoring tool embedded in Emacs. To perform a refactoring, the source of inter-

[10] `http://www.vim.org` `http://www.gnu.org/software/emacs/`

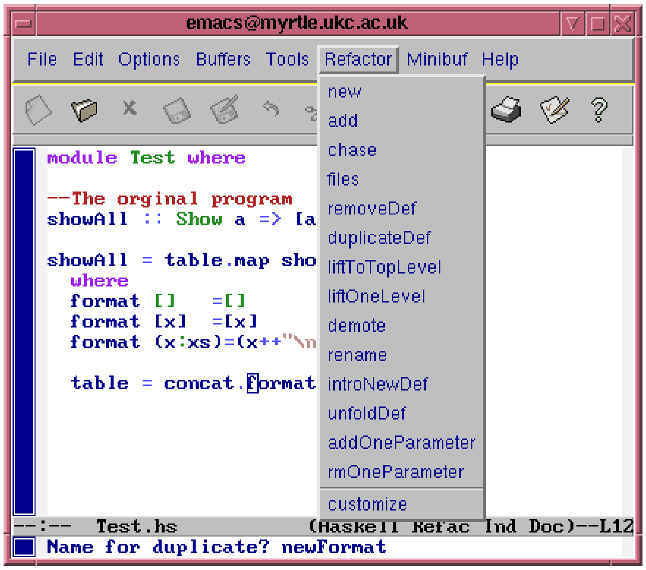

Figure 6: A snapshot of the Haskell refactoring tool embedded in Emacs

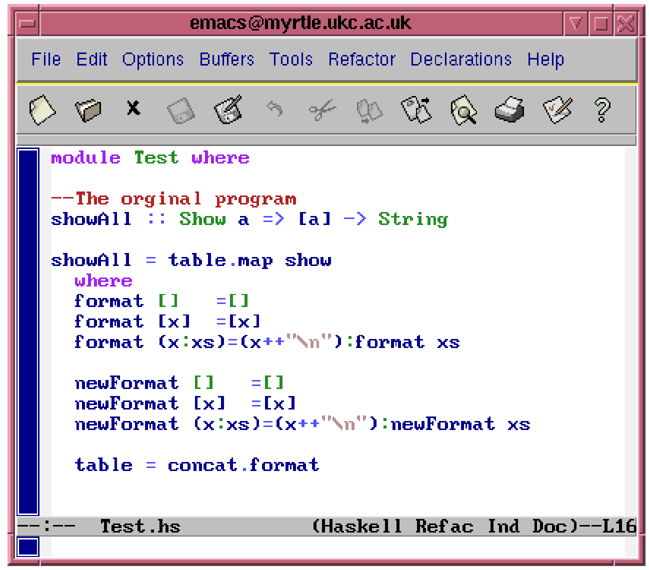

Figure 7: A snapshot showing the result of duplicating a definition

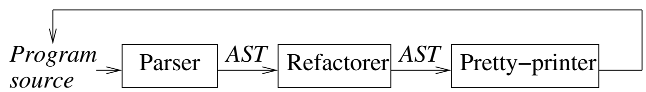

Figure 8: The original implementation architecture

est 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 any parameters in the mini-buffer if prompted. After that the refactorer will check the selected source is suitable for this refactoring, the parameters are valid, and the refactoring's side-conditions are satisfied. If all checks are successful, the refactorer will perform the refactoring, otherwise it gives an error message and aborts the refactoring. Using the refactoring tool embedded in Vim is similar.

Figure 6 also shows a particular refactoring scenario. The user has selected the identifier `format` in the definition of `table`, has chosen the *duplicateDef* command from the *Refactor* menu, and is just entering a new definition name `newFormat` in the mini-buffer. After this, the user would press the *Enter* key to perform the refactoring. The result of this is shown in Figure 7: a new declaration defining `newFormat` has been added to the module after the definition of `format` (note that, unlike editor-based copy&paste, the refactorer ensures consistent renaming, including recursive calls).

4.2 The Implementation Architecture

The design of the implementation architecture has evolved through a number of stages. Figure 8 shows a graphical overview of the original implementation architecture: to perform a refactoring, the parser takes the program source and parses it into an abstract syntax tree (AST), the refactorer then carries out program analysis and transformation on the AST and after that, the pretty-printer presents the modified AST to the programmer in concrete syntax form. This architecture is straightforward, but has two fatal disadvantages:

- The program is parsed before each refactoring, even if there has been no editing activity since the previous refactoring. Frequent reanalysis of large programs can be time-consuming, which could discourage programmers from using an automatic refactoring tool[11]. To avoid this, one would want to reuse the AST if possible. However, the AST contains information about the position of identifiers (which turns out to be very useful for both program analysis and transformation) and, after a refactoring, some position information in the AST may have become invalid. The new layout is first computed in the pretty-printer, so one could try to update the position information in the abstract syntax tree there, but this involves dramatic changes of the pretty-printer.

- Comments are not preserved in the AST and the pretty-printer produces output which completely ignores the style – let alone the concrete layout – of the input program, so after a refactoring, programmers may find all the comments in their programs are missing and the layout style produced by the pretty-printer is completely different from that before refactoring.

  Clearly, this is unacceptable from a programmer's point of view, so one would want to preserve comments in the abstract syntax tree, analyse the input program for the style of layout used, and modify the pretty-printer to adapt to the input style and to reproduce comments. However, this is far from ideal.

Inspired by the fact that *Programatica*'s lexer preserves position information, and that comments and white space are also preserved in the first passes of the lexer, we adjusted the implementation architecture as shown in Figure 9. Two separate improvements address the problems of the original architecture:

- In the new architecture, the refactorer operates on two views of the program: the token stream produced by the first pass of the lexer (with comments and white spaces still intact) and the AST produced by full lexing and parsing. The AST is used only as an auxiliary representation of the program to guide the direct modification of the token stream. The refactorer still performs program analysis and transformation on

[11] Realistic refactorings are composed of multiple small steps, each of which appears to users as an advanced editing operation, lowering the threshold for what is deemed acceptable processing time.

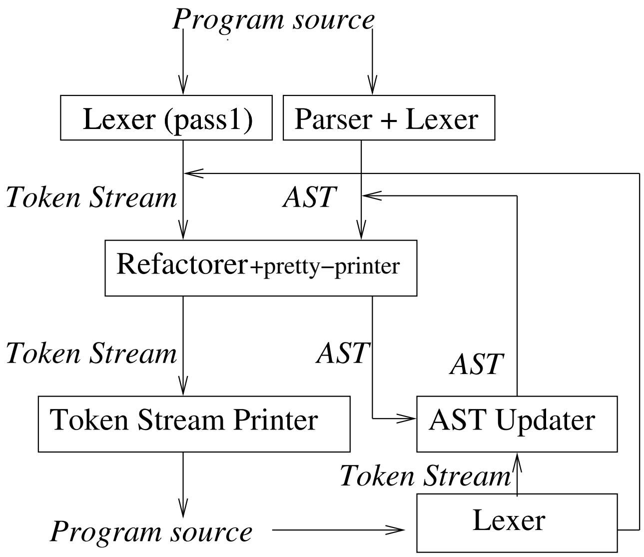

Figure 9: The new implementation architecture

the AST but, once the AST has been modified, the refactorer will modify the token stream as well to reflect the changes in the AST. The token stream also needs adjustment to counteract the side-effects of the transformation on the layout rules.

- If there is no editing activity between the current refactoring and the previous one it is possible to update and reuse the AST. Lexing the program output yields a token stream in which each token has correct position information, and we can use this to update the position information in the AST from the previous refactoring. After that, the updated token stream and AST are ready for the next refactoring.

Instead of using the pretty printer to present the modified AST to the programmer in concrete syntax form, the new program will be extracted from the token stream, preserving both comments and layout style for the majority of programs. For certain refactorings, the refactorer may produce new code in which no layout information can be inferred (e.g., when unfolding a definition with multiple guards, the multiple clauses may have to be translated into a conditional expression). In this case, the pretty-printer will be used to print the new code, which is then inserted into the token stream. Also, just preserving comments is not sufficient, so we apply simple heuristics to associate and move comments with nearby code.

Next, we describe the components in the new architecture in some more detail, explaining our use of *Strafunski*-style generic programming and *Programatica* front-end components.

4.2.1 The Lexer

Programatica's lexer [11] is generated from a lexical syntax specification and is split up into several passes. As mentioned before, position information is preserved by the lexer, and the first passes of the lexer preserve comments and white space. These two features make it possible for our refactoring tool to preserve comment and program layout style as well as avoiding reparsing the program. The lexer splits an input `String` into a list of token `Strings`:

```
type Lexer = String -> [(Token,(Pos,String))]
```

where `Token` is a data type classifying tokens, and `Pos` represents the position of the token in the source.

4.2.2 The parser

Programatica's parser is based on hsparser, but with a parameterised abstract syntax supporting syntax variants and extensions. We outline the treatment of identifiers only, as these are central to the refactorings presented. In the AST produced by the parser, each identifier is paired with its position information in the source file.

A further scoping process on the AST adds more information to each identifier, and produces a new variant of the AST. In this variant, called the scoped AST, each identifier is associated with not only its position in the source file, but also the information about where it is defined and which name space it belongs to. The type used for identifiers in the scoped AST is called `PNT` and defined as:

```
data PNT=PNT (PN HsName Orig) (IdTy Pid) OptSrcLoc
```

Roughly, `HsName` contains the name of the identifier, `Orig` specifies the identifier's origin information which usually contains the identifier's defining module and position, the identifier's name space information is contained in `(IdTy Pid)`, and `OptSrcLoc` contains the identifier's position information in the source file.

The scoped AST makes our life easier in the following aspects.

- Source position information makes the mapping from a fragment of code in the source (editor view) to its corresponding representation in the scoped AST (refactorer view) easier.
- Identifiers in different scopes can be distinguished by just looking at the `PNT` values themselves. Two identifiers are same if and only if they have same origin.
- Given an identifier, the scoped AST makes it convenient to find the binding definition of the identifier if there is one, as well as the places where it is used.

4.2.3 The Refactorer

The refactorer is the engine that actually performs the program analysis and source-to-source transformation. Program analysis aims at validating the side-conditions of refactorings. Program transformation performs rewriting of the AST and token stream.

Both program analysis and transformation involve traversing the scoped AST frequently, which is where *Strafunski* [20, 19] comes into play. As discussed in Section 3.2, *Strafunski* was developed to support generic programming in application areas that involve term traversal over large abstract syntaxes, such as Haskell's. The key idea is to view traversals as a kind of generic function that can traverse into terms while mixing uniform and type-specific behaviour. *Strafunski* offers both a generic traversal combinator library *StrategyLib* and a generative tool support based on *DrIFT*[12] to use the library on large systems of data types.

Two kinds of generic functions can be constructed using the combinators provided in *StrategyLib*: type-preserving generic functions dealing with program transformation and type-unifying generic functions dealing with program analysis. The result of applying a type-preserving generic function to a term of type t is of type t in a monadic form (so that transformations can fail or return multiple results), whereas the result of a type-unifying generic function application is always of a specific type, say a (again in a monadic form) regardless of the type of the input term.

Figures 10 and 11 give two simple examples, illustrating the form of code implementing program analysis and transformation using *Strafunski*'s combinators with *Programatica*'s data types.

The example in Figure 10 defines a type-unifying generic function which collects all the data constructors in a fragment of Haskell code. Here, the functions `applyTU, stop_tdTU, failTU` and

[12] `http://repetae.net/john/computer/haskell/DrIFT/`

```
--type PName = PN HsName Orig

hsDataConstrs::(Term t)=>t->Maybe [PName]
hsDataConstrs = applyTU worker
   where
   worker = stop_tdTU (failTU `adhocTU` pntSite)
   pntSite ::PNT->Maybe [PName]
   pntSite (PNT pname (ConstrOf _  _) _)
             = Just [pname]
   pntSite _ = Nothing
```

Figure 10: Collecting data constructors

```
-- data HsName = Qual ModuleName String
--             | UnQual String
-- type PName = PN HsName Orig

rename::(Term t)=>PName->HsName->t->Maybe t
rename oldPName newName = applyTP worker
   where
   worker = full_tdTP (idTP `adhocTP` pnameSite)
   pnameSite :: PName -> Maybe PName
   pnameSite  pn@(PN name orig)
    | pn == oldPName = return (PN newName orig)
   pnameSite pn = return pn
```

Figure 11: Renaming an identifier

`adhocTU` are type-unifying variants of strategy combinators from *StrategyLib*. `stop_tdTU s t` denotes a top-down traversal of term `t`: it applies the strategy `s` at each level, stops if `s` succeeds, recurses if `s` fails, and collects the results. The polymorphic strategy `failTU` always fails, `s1 `adhocTU` s2` extends a polymorphic default strategy `s1` with a type-specific strategy `s2`.

The function `worker` performs a top-down traversal of the AST to the PNT level, where it calls `pntSite`. This latter function returns the data constructor name in the `Maybe` monad if the current `PNT` is a data constructor, otherwise, it returns `Nothing`. We use the list data type to deal with the case where there are several data constructor names. In combination with `stop_tdTU`, the default strategy `failTU` indicates that `worker` always recurses when faced with terms of any other type than `PNT` and that only applications of `pntSite` to subterms of type `PNT` contribute to the result of applying `worker` to a term of arbitrary type.

A type-preserving generic function renaming all occurrences of a specified identifier to a new name is defined in Figure 11. Using the combinators `applyTP, full_tdTP, adhocTP` and `idTP` from *StrategyLib*, this function carries out a full top-down traversal over the AST as specified by `full_tdTP`. This way, it will reach each node in the input AST. Most of the time, it behaves like `idTP` which denotes the polymorphic identity strategy, but it will call the function `pnameSite` whenever a term of type `PName` is encountered. The function `pnameSite` replaces the identifer name contained in current `PName` by `newName` if this identifier is same as the identifier to rename. Otherwise, it returns the `PName` unchanged.

Our early experience in implementing the refactoring tool using *Strafunski* indicates that this style of generic programming to some extent liberates us from the complexity of Haskell syntax, avoiding large amounts of boilerplate code that is tiresome to write, vulnerable to change and error prone. We need only to describe functionalities for the types and constructors that are immediately relevant for an analysis or transformation at hand, thus concentrating on the significant part of the implementation.

So far, the code needed to implement a refactoring (not counting general support code) appears to average out at about 200 lines of code. For comparison: in our experiments without generic traversal support, even the simple renaming example of Figure 11 amounted to that many lines. Chris Ryder, whose metrics library [27] tackles similar analysis tasks over the Haskell grammar without *Strafunski*, corroborates our own impression that the refactoring implementations are comparatively concise and readable.

However, even with the help of *Strafunski* and *Programatica*'s frontend, implementing a practical refactoring tool is still non-trivial:

- Dozens of refactorings are going to be implemented – an initial catalogue (that predates our implementation and needs to be revised) is available from our project home page.
- Each refactoring has its own side-conditions and transformation rules, usually involving both syntactical and semantical analysis. We have not even started to support type-based or type-level refactorings, nor have we addressed multi-module refactorings or their interaction with separate compilation.
- The refactoring tool should support composite refactorings, built from series of elementary refactorings. *StrategyLib* provides a starting point, but the challenge is to facilitate user-defined composite refactorings.
- The refactoring tool should support undo/redo of refactorings. Editor-side undo/redo does not interact well with changes initiated by an external tool, possibly affecting multiple files.
- We need to develop theory to support the implementation.

4.2.4 The Token Stream Printer

After a refactoring, the token steam printer gets the modified token stream from the refactorer, extracts the strings contained in tokens, concatenates them together getting the new program source, and finally, presents it to the programmer.

4.2.5 The AST Updater

If one refactoring immediately follows another, the scoped AST from the previous refactoring can be reused. However, some position information in this scoped AST might be invalid, and some tokens in the modified token stream from the refactorer might not be really tokens. So, we lex the new program source getting a token stream with correct information associated with each token, then use the new token stream as a guide to update the position information in the scoped AST, in a two-pass process. The first pass collects the identifiers and their associated source positions in a topdown, left-to-right manner. The second pass updates the position information in the AST according to a mapping between the positions from the first pass and the positions from the token stream. Our experiments show that this method is much faster than reparsing and rescoping the program.

4.3 Haskell programming in-the-large

Our own project is just at the beginning of what is going to be a substantial code base, but in order to keep the size of code for each refactoring manageable, we are building on other projects, namely *Strafunski* and *Programatica*. Each of these is a substantial package of Haskell code in its own right, building on both standard and project-specific support libraries, and we are integrating these packages with our own code and support libraries, so we can offer some preliminary comments on Haskell programming in-the-large:

- Of the roughly 470 Haskell files in the *Programatica* snapshot, our project recursively imports some 210 modules. *Strafunski*'s StrategyLib only adds about 20 modules, but it uses *DrIFT* to generate application-specific instances of generic traversal functions. We automatically extract the AST-related types from the various *Programatica* modules and feed them to *DrIFT* without changing the *Programatica* sources or inspecting the generated boilerplate code (<1k lines). Not counting the editor interfaces, we have so far added about 4400 lines of code (one module per kind of refactoring, roughly 200 lines each, plus some 2k lines of support code).
- Haskell has a simple and flat module system. With a large number of modules, distributed over several directory hierarchies, it is often not obvious which module an item comes from and where that module is. The hierarchical namespace extension should help with the latter, consequent use of explicit import interfaces helps with the former, but not with recursive imports or with type class instances.

  Tool support for program understanding is indispensible here, e.g., we have applied *Programatica* to itself to generate a module dependency graph and have fed that into a standard graph layout tool, but for a graph of this size, interactive navigation and focussing would be much more useful. A prototype module graph browser under development in the MEDINA project [27] shows promise here.

  Another obvious tool is the generation of tag files for function and type definitions, by which both Emacs and Vim permit interactive browsing from uses to definitions. However, both *Programatica* and *Strafunski* make good use of higher-order functions and overloading, so that jumping from one module to another to follow chains of definitions often does little to reveal the overall picture of control and data flow.

  With extensive use of type inference and advanced type-based features such as monad transformers (*Programatica* even overloads standard IO operations to ease lifting), non-type-aware tools are at a loss at figuring out which monads a particular fragment of code might be running in, and what information and operations might be available at any given point.
- If build times are inexplicably slow, suspect a bug, and not necessarily in your Haskell compiler. In our case, the problem could be traced to the linker. Coercing GHC to use Gnu `ld` instead of the default `ld` on Solaris brought link times down from 20 minutes to under one minute!
- Some Haskell tools and libraries are limited-purpose or proof-of-concept implementations: the version of *DrIFT* used in *Strafunski* ran into problems with *Programatica*'s more complex data structures representing the Haskell AST, but the open-source model made it possible for us to fill the gaps.
- Haskell's module system lacks control over the import/export of class instances. This can lead to conflicts when linking third-party libraries (such as *Programatica*'s frontend and *Strafunski*'s StrategyLib) together: all instances defined in any modules imported by either of the two projects spill out to any code that imports them. Avoiding conflicts requires extra careful information hiding wrt the types (use newtypes instead of type synonyms when defining not-quite-standard instances, e.g., functor and monad instances for error monads based on `Either`). Even for standard instances (e.g., functor and monad instances for environment monads based on `((->) a)`), conflicts arise if equivalent instances are defined within two sub-projects or their support libraries – common instances have to be factored out into shared modules.

5. RELATED WORK

We have already mentioned the origins of our work in early program transformation and language-aware development environments. The last comprehensive survey of these areas we are aware of dates from 1983 [24], so we have started to compile a list of URLs and references in the related work section on our project home page and only give examples of recent developments here.

Within the SE and OO communities, refactoring has been identified as central to the software engineering process [9, 7, 23]. There is a growing collection of tools for refactoring in a variety of languages. Notable among these is the Refactoring Browser [25], which was the first successful tool, and which supports Smalltalk refactoring. Among the other tools, listed at Fowler's [7] and at our web site, are systems for refactoring C, C#, Java, Python and UML; Java is by far the most popular. Many of the tools are components of IDEs (Integrated Development Environments) of one sort or another; many of the tools are also commercial products.

Fowler proposed [13] support for the 'extract definition' refactoring as an indication that a refactoring tool had 'crossed the Rubicon'. It is disappointing that this test appears to be seen by many as a sufficient rather than a necessary condition for the tool to be complete. It is interesting to speculate that this might be due to the effort involved in writing refactoring support with all its attendant boilerplate in, say, C#, as compared to using a high-level strategy library such as *Strafunski*. Also, the presence of unrestricted side-effects hinders program analysis and transformation.

Tool support for functional refactoring has been explored substantially less. Lämmel has used *Strafunski* to describe generic and functional refactorings [16, 14, 15]; this work has tended to focus on principles of program transformation rather than on tooling for specific, complete, languages. Martin Erwig has investigated a Haskell Update Language [5]; this shares some of its aims with our project, but treats programs as members of ADTs rather than as (both) concrete strings and Abstract Syntax Trees.

Systems such as Ultra [10], PATH [30], MAG and HsOpt [4, section 5.2.2] support Haskell program transformation for program derivation or for optimisation; both areas have a different emphasis from refactoring. Program derivation refines simple, obviously correct, programs into more complex, more efficient variants; program optimisation tends to focus on localised transformations, 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, that is to large-scale declarative aspects rather than smaller-scale operational ones. In spite of these differences, the substantial overlap in concepts and infrastructure requirements suggests integrated tool support for derivation, optimisation, and refactoring.

6. FURTHER WORK AND CONCLUSIONS

We expect our work to develop in a number of directions. The tool reported in this paper has crossed a 'Refactoring Rubicon' of sorts, but it is in its infancy. There is a host of more complex transformations to be added to the system, which will require the system to be aware of Haskell types and modules. Examples include

[13] `http://www.martinfowler.com/articles/refactoringRubicon.html`

turning a concrete data type into an abstract one, or transforming a non-monadic computation into a monadic form. These will be *composite* refactorings, to be built by assembling a number of atomic refactorings of the kind presented here. Support for undo and redo operations will be fundamental in allowing programmers to take a speculative approach to refactoring.

The refactorings we have written will have to be properly documented in a *catalogue*; an early draft can be found at our project web site[14]. Central to the catalogue – as we saw in the examples described in Section 2 – is a description of the conditions under which each refactoring can be applied. 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. We expect to develop proofs of correctness for a variety of the refactorings implemented.

In conclusion, we have built a tool which shows the utility of refactoring in the functional domain[15]. The tool is integrated with the usual Haskell IDEs, namely Emacs and Vim, and preserves program layout and comments; we view both of these as essential if the system is to be used by practising programmers rather than being an object of polite curiosity. Implementing the tool has been made possible only because of various toolkits – *DrIFT*, *Programatica*, *Strafunski* – which support the analysis of Haskell programs and the construction of generic, typed syntactic transformations.

Strafunski's StrategyLib has, in particular, allowed us to express the essence of the refactorings in concise and readable form, in contrast to hiding the effects in pages of boilerplate.

7. REFERENCES

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

[2] 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.

[3] Iavor S. Diatchki, Mark P. Jones, and Thomas Hallgren. A Formal Specification for the Haskell 98 Module System. In *ACM Sigplan Haskell Workshop*, 2002.

[4] C. Reinke (ed.). Haskell Communities and Activities Report. `http://haskell.org/communities/`, May 2003.

[5] M. Erwig and D. Ren. A Rule-Based Language for Programming Software Updates. In *3rd ACM SIGPLAN Workshop on Rule-Based Programming (RULE'02)*, 2002.

[6] Robert W. Floyd. The paradigms of programming. *CACM*, 22(8), August 1979. Also appears in ACM Turing Award Lectures: The First Twenty Years 1965-1985.

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

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

[9] William G. Griswold. *Program restructuring to aid software maintenance*. PhD thesis, Univ. of Washington, Dept. of CS and Engineering, 1991. Tech. Rep. No. 91-08-04.

[10] W. Guttmann, H. Partsch, W. Schulte, and T. Vullinghs. Tool Support for the Interactive Derivation of Formally Correct Functional Programs, Ext. Abstr. In *FM-TOOLS*, July 2002. `http://www.informatik.uni-ulm.de/pm/ultra/`.

[11] T. Hallgren. A Lexer for Haskell in Haskell. `http://www.cse.ogi.edu/~hallgren/Talks/LHiH/2002-01-14.html`.

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

[13] Mark P. Jones. Typing Haskell in Haskell. `http://www.cse.ogi.edu/~mpj/thih/`, November 2000.

[14] Jan Kort and Ralf Lämmel. A Framework for Datatype Transformation. In *Proc. of Language, Descriptions, Tools, and Applications (LDTA 2003)*. Elsevier, April 2003.

[15] R. Lämmel. Reuse by Program Transformation. In Greg Michaelson and Phil Trinder, editors, *Functional Programming Trends 1999*. Intellect, 2000.

[16] R. Lämmel. Towards Generic Refactoring. In *Proc. of Third ACM SIGPLAN Workshop on Rule-Based Programming RULE'02*, Pittsburgh, USA, October 2002. ACM Press.

[17] R. Lämmel and S. Peyton Jones. Scrap your boilerplate: a practical design pattern for generic programming. In *Proc. of the ACM SIGPLAN Workshop on Types in Language Design and Implementation (TLDI 2003)*. ACM Press, 2003.

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

[19] R. Lämmel and J. Visser. Design Patterns for Functional Strategic Programming. In *Third ACM SIGPLAN Workshop on Rule-Based Programming RULE 2002*. ACM Press, 2002.

[20] R. Lämmel and J. Visser. Typed Combinations for Generic Traveral. In *Practical Aspects of Declarative Programming PADL 2002*, volume 2257. Springer-Verlag., January 2002.

[21] R. Lämmel, J. Visser, and J. Kort. Dealing with Large Bananas. In J. Jeuring, editor, *Proceedings of WGP'2000, Tech. Report, Universiteit Utrecht*, pages 46–59, July 2000.

[22] Simon Marlow. Haddock: A Haskell Documentation Tool. `http://www.haskell.org/haddock/`.

[23] William F. Opdyke. *Refactoring Object-Oriented Frameworks*. PhD thesis, Univ. of Illinois, 1992.

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

[25] Don Roberts, John Brant, and Ralph Johnson. A Refactoring Tool for Smalltalk. *TAPOS special issue on software reengineering*, 3(4):253–263, 1997. see also `http://st-www.cs.uiuc.edu/users/brant/Refactory/`.

[26] Donald Bradley Roberts. *Practical Analysis for Refactoring*. PhD thesis, Univ. of Illinois at Urbana Champaign, 1999.

[27] Chris Ryder. The Medina metrics library for Haskell. `http://www.cs.kent.ac.uk/~cr24/medina/`, 2002.

[28] Tim Sheard. Generic Unification via Two-Level Types and Parameterized Modules. In *ICFP'01, Firenze, Italy*, September 2001. expanded version submitted to JFP.

[29] R. D. Tennent. Language Design Methods Based on Semantic Principles. *Acta Informatica*, 8:97–112, 1977.

[30] Mark Tullsen. *PATH, A Program Transformation System for Haskell*. PhD thesis, Yale University, May 2002.

[31] Eelco Visser. Stratego – Strategies for Program Transformation. `http://www.stratego-language.org/`.

[32] Eelco Visser, Zine-el-Abidine Benaissa, and Andrew Tolmach. Building Program Optimizers with Rewriting Strategies. In *ICFP'98*. ACM Press, September 1998.

[14] `http://www.cs.kent.ac.uk/projects/refactor-fp/`

[15] A first release will be available from our project home page.