CHALMERS
UNIVERSITY OF TECHNOLOGY

UNIVERSITY OF GOTHENBURG

Automatic refactoring for Agda

Master's thesis in Computer science and engineering

Karin Wibergh

Department of Computer Science and Engineering
Chalmers University of Technology
University of Gothenburg
Gothenburg, Sweden 2019

Master's thesis 2019

Automatic refactoring for Agda

Karin Wibergh

UNIVERSITY OF GOTHENBURG

CHALMERS
UNIVERSITY OF TECHNOLOGY

Department of Computer Science and Engineering
Chalmers University of Technology
University of Gothenburg
Gothenburg, Sweden 2019

Automatic refactoring for Agda
Karin Wibergh

Supervisor: Ulf Norell, Department of Computer Science and Engineering
Examiner: Andreas Abel, Department of Computer Science and Engineering

Master's Thesis 2019
Department of Computer Science and Engineering
Chalmers University of Technology and University of Gothenburg
SE-412 96 Gothenburg
Telephone +46 31 772 1000

Typeset in LaTeX
Gothenburg, Sweden 2019

Automatic refactoring for Agda
Karin Wibergh
Department of Computer Science and Engineering
Chalmers University of Technology and University of Gothenburg

Abstract

The task of making changes to an existing code base to improve performance, legibility, or extensibility while preserving behaviour is important to virtually any program. Many times this involves making changes requiring a great deal of typing in various places, which is tedious and error-prone. Consequently, programs known as refactoring engines are used to take over the predictable parts of this task. However, although common for imperative and object-oriented languages, refactoring engines for functional languages like Haskell are rare and those for dependently typed languages are nonexistent. This project lays the groundwork for a refactoring engine for Agda by describing useful refactorings and a handful of implementation strategies.

Keywords: Agda, dependent types, refactoring, thesis

Contents

1
Introduction

It seems obvious that there is no point to a program which is not being used. Strangely, however, the most salient result from multiple usage studies is that refactoring engines, if not entirely unused, are generally considerably under-used. In a survey of Microsoft developers, the participants reported that they did some 86% of refactorings manually, while just over half reported doing all their refactoring manually [9]. Another survey of people at the Agile Open Northwest 2007 conference found that respondents failed to use available tools for refactoring 32% of the time. The authors speculate that programmers not adhering to the notoriously refactoring-happy agile method would use tools even less [14]. In a study of Eclipse users at Portland State University workstations, only 6 of the 42 Eclipse users used refactoring tools at all [13]. Particularly amusing is a study which examined the commit logs of four developers working on Eclipse's refactoring tools, whose results suggest that the developers used available tools only between 11% and 27% of the time. The study authors suggest, however, that the method used might sometimes be unable to correctly detect refactoring tool use [15].

So why are refactoring engines not being used as much as they could be? There are a handful of explanations which turn up frequently in the literature and seem so far to have remained uncontradicted. One is that programmers are not aware of the tools' existence in the first place [20] [17] [4]. This is likely related to the issue that refactorings may have names which the developers find incomprehensible – i.e. they cannot guess what a refactoring is likely to do based on the name [20] [4]. This means that programmers have difficulty discovering refactoring tools merely by browsing the refactoring menu.

A related problem is complex configuration and setup for refactorings [20]. One paper asserts that some programmers believe – correctly or not – that they can perform the refactorings faster by hand than with a tool. The authors suggest too many shortcuts and too complicated configuration dialogs as possible reasons [4]. Another paper confirms the ideas that programmers do not understand the tools, and feel that they can refactor faster by hand [13]. Yet another paper reports that only 10% of configuration defaults are changed [15]. Some researchers go so far as to recommend avoiding setup steps as much as possible to mitigate these problems [14].

Another issue is that some refactorings are simply rarely needed, and programmers may therefore choose to refactor by hand rather than learning to use the appropriate refactoring command [20]. Also, there is a tendency among programmers to let the automated tools do only simple, clearly defined tasks. Refactorings making a large number of complicated changes may go unused [20].

There is another explanation which is discussed frequently: correctness issues. Unlike the other explanations, this one is considerably more contested. One uncontroversial preliminary observation is that refactoring engines are no less error-prone than any other code. While there are no well-tested established refactoring tools for Haskell, much less for Agda or any other dependently typed language, one may get a rough idea of the number of errors in such tools by looking at studies of those for other languages, most notably Java and C. For example, one 2007 study found 9 bugs in the Eclipse refactoring engine for Java and 10 in that of Netbeans [5]. In 2013, another team which tested the Eclipse refactoring engines for Java and C found 77 bugs for Java and 43 bugs for C; this in spite of the fact that the only test oracle (a test to determine if the result is the one desired) used was compilation [8], which would be easy enough to pass by simply making no changes at all or by replacing the entire project with a Hello-World-program.

The next question is how important it is that the refactoring engine works correctly. There seem to be different opinions about this. Huiqing Li, in her Ph.D. thesis on the Haskell refactoring tool HaRe, devotes an entire chapter to the verification of refactorings [10]. The developers of the tool SafeRefactor, which is supposed to prevent unsafe refactorings, consider bugs introduced by automated refactorings to be unusually nefarious because typically the project's test suite will need to be refactored too (for example in the case of renaming). In the case of a bug in the refactoring engine, the tests may no longer test the same thing as before [19]. On the other hand, one paper points out that correctness often conflicts with speed requirements, which are also very important for usability when refactoring large code bases [3]. It has even been argued that the continued existence of buggy refactoring engines is an indication that correctness is not vital to this type of task, and that speed is typically more important than accuracy [16]. Given the usage statistics above, though, this view seems a little too optimistic.

Surveys of programmers are also somewhat inconclusive. In one survey, subjects said that they believed they could easily fix errors caused by a refactoring tool. However, the same study reports that when programmers do use tools, they will ignore unsafe-refactoring warnings 79% of the time – and suggests that false positives (i.e. bugs) may be to blame for this [20].

Given these problems and controversies, it is obvious that writing a refactoring engine is not straightforward. This project is intended to collect information which may be relevant in building a fully-fledged refactoring engine for the dependently typed language Agda, whether based on the prototype developed in this project or integrated into the current compiler. In this report you will find a laundry list of possibly useful refactorings, of which some come from existing refactoring engines while others have been suggested by active Agda programmers at Chalmers. Focussing primarily on refactorings particular to or affected by dependent types, we discuss useful strategies for implementing them for Agda, drawn partly from previous work and partly on our experiences implementing these refactorings for a subset of Agda. In doing this, we ignore efficiency issues in favour of easy-to-understand and correct code.

2 RefactorAgda

RefactorAgda is the working name of the refactoring tool produced in this project. It works on a subset of Agda called Baby-Agda which is detailed below. RefactorAgda is restricted to making changes to only one file per refactoring. This allows us to ignore any problems with speed and undoability and makes the ones relating to hard-to-understand refactoring specifications a little easier – programmers can safely try out refactorings and discard the changes with the press of a button if they do not like what they see. In view of the fact that a study found that 84% of automated refactorings only change one file [20], and of the limited scope of this project, this seems like a reasonable limit.

In the following we will detail the tools used and high-level design choices made for the refactoring tool, along with some comments about their suitability in practice.

2.1 The parser

The real Agda parser is written using Alex and Happy. These tools, however, have the disadvantage of requiring the use of a special programming language each. Considering that the rest of the program is written in Haskell and Agda, this would mean juggling four different programming languages in the same project, which goes against the project goal of readability.

Consequently, the Haskell parser combinator library Megaparsec is used instead. This library has the advantage over its more established ancestor Parsec in that functions for parsing layout-sensitive languages are available in the library, whereas with Parsec an external package would have to be used. Also, Megaparsec is unlike Parsec under active development.

This proved to be quite a convenient choice. Megaparsec encourages the use of sub-parsers which can be easily combined in various ways. Some of these sub-parsers we have reused in other places, such as when interpreting the output from the Agda compiler when this was necessary. Also, since there is essentially no restriction on what kind of parsers you can write (unlike Alex and Happy, which are restricted to languages expressible in their custom syntax), it is quite easy to get any result you like once you have got used to the available combinators. In particular, Megaparsec can easily be made to reject some programs which fit into the data structure being parsed into but do not make sense, such as programs which use holes on the left-hand side of a function definition.

Design-wise, the trickiest problem is what to do about comments and formatting. Unlike parsers used in for instance compilers, a refactoring parser must preserve formatting and comments as best it can. There are two possible approaches to the formatting problem. One is to essentially leave comments and whitespace in place in the source file and change only the code. This is the approach chosen by the inventors of the Haskell refactoring tool HaRe [1]. However, there are problems with this approach. One is the assumption that the programmer's comments and layout will be faithfully preserved as long as they are not touched. This, however, is far from being the case: in fact, comments and layout are very much determined by the code they surround. This is especially prominent in the case of refactorings which move large amounts of code, such as EXTRACT FUNCTION, where the tool must make assumptions about what a comment might have been about to know whether to move it or not. Also, in many cases the refactoring tool inserts extra code, for which it needs some type of layout which may not be easy to determine from the surrounding code. Another big problem is the practical implementation: HaRe uses both an abstract syntax tree (AST) for the code and a token stream for the layout information, which it takes a great deal of effort to keep in sync. As a matter of fact, this was not a deliberate design choice by the developers, but an unfortunate result of the fact that the Haskell front-end HaRe is based on throws away a great deal of important information. Building a custom parser and pretty-printer for Haskell was considered infeasible [10].

The other possible approach is to parse comments and layout information as a legitimate part of the AST, make the necessary transformation, and pretty-print the result without reference to the original token stream. This is the approach taken by Haskell-tools [2], which combines information from different parts of the GHC parsing process to produce an AST annotated with comments and whitespace.

This project has the advantage of being able to use its own custom parser instead of having to make do with an already existing one for a completely different purpose. This allows it to go a little further still and parse comments as legitimate parts of the parse tree and include their movement in the refactoring specifications. This approach would also allow experimentation with refactoring comments in the case of for example renaming, although such experiments are not part of this project.

As regards layout, this project takes a radically different route from the two Haskell refactoring tools by discarding it altogether when parsing and instead using a customizable pretty-printer. This is based on the fact that any refactoring will in any case considerably disrupt a programmer's natural style by adding or changing code, and better results can therefore be expected by specifying the output style than by simply inserting something randomly formatted into the code. Further, this will give us a reindentation refactoring for free.

A closely related subject is syntactic sugar. Agda has many constructs with identical meanings. Preserving the particular choice would cause the definition of the AST to explode in size, and there would have to be an implementation of the functionality for each sugared form. However, for some types of syntactic sugar, it is quite reasonable to suppose that the programmer always prefers a particular style over another, while in others the most suitable style is context-dependent and must be parsed. Consider, for example, the difference between do-notation and bind-notation: which one is the most legible depends very much on the particular context. On the other hand, the optional arrows in Agda's function types need not be parsed into a separate structure.

To minimize any adverse side effects from this strategy, the pretty-printer will rewrite only those parts of the input affected by any given refactoring. For details on this, see section 2.3.

2.2 Baby-Agda and its associated data structures

This project currently uses a small subset of Agda to try out refactorings on, which we call Baby-Agda. The syntax of this language is shown in listing 1.

```
ParsedProgram ::= [ParseTree]

ParseTree ::= TypeSignature
            | FunctionDefinition
            | DataStructure
            | OpenImportStatement

TypeSignature ::= Identifier : Expression

FunctionDefinition ::= Identifier [Expression] = Expression

DataStructure ::= data Identifier [TypeSignature] : Expr where [TypeSignature]

OpenImportStatement ::= open QualifiedName
        | import QualifiedName
        | open import QualifiedName

Expression ::= ExpressionA ExpressionB
             | Literal
             | Hole
             | Identifier
             | {Expression}
             | _
             | (x : ExpressionA) -> ExpressionB
             |{x : ExpressionA} -> ExpressionB
             | ExpressionA -> ExpressionB
```

Listing 1: Baby-Agda's syntax

Baby-Agda also has comments, which can appear essentially anywhere where whitespace may appear and are therefore not shown in the syntax. We choose to attach these comments to the closest identifier, literal, or hole so we will not have to deal with comment fields in all parts of the tree. This is reasonably accurate but causes inevitable ambiguity when parentheses are used, because we cannot distinguish between comments after an identifier but before the closing parenthesis or comments immediately following the parenthesis. This could be partially solved by attaching comments to expressions instead, but this would add another layer of complication to to the expression transformation which we describe below. Even in this case, however, we would still have the same problem with regard to redundant parentheses, which have no representation in the parse tree.

There is one thing in the way the syntax is parsed which is particularly relevant to refactoring. In the specification of Baby-Agda above we chose to parse function application as seen in the first part of listing 2. This way of parsing reflects the fact that function application is left-associative. However, there is another possible data structure, shown in the lower part of the same listing, which resembles the one used in the actual Agda compiler. This form reflects the common programmer's view of some functions as having several arguments, as opposed to having one argument and returning a function. The main advantage of this form is that the name of the function is in an easily accessible position, whereas in the other form it would be buried at the very bottom of the parse tree.

```
Expression ::= ExpressionA ExpressionB

 Expression ::= Identifier [Expression]
```

Listing 2: Two ways of storing expressions

We have done some implementation experiments with both these alternatives and found that EXTRACT FUNCTION and the expression-changing refactorings are particularly affected by which data structure we choose. For EXTRACT FUNCTION, using the list structure causes a large, unwieldy algorithm for determining which subexpression is supposed to be extracted. The expression-changing algorithm, on the other hand, is trivial with the list structure but much harder with the tree data structure, because you need to reach the very bottom of the expression tree before you even find out if it is one of the expressions to be refactored.

Since both data structures essentially contain the same information, it is possible to convert one into the other. We note that regardless of which structure we select as the “base” used by the parser, we need to define the transformation in both directions in order to be able to fit the result into the final parse tree. Even if one direction is more difficult than the other, therefore, the point is academic.

The first option is to use the list version as base data structure, and try to rewrite it into a tree structure for EXTRACT FUNCTION. Since EXTRACT FUNCTION depends on asking Agda for various pieces of information, we would either need a special show function for this data structure, or we end up doing a lot of conversion backwards and forwards. On the other hand, taking the tree as base data structure and rewriting it as a list for the expression-changing refactorings, we find we do not need to do any extra work anywhere else.

This leads us to the conclusion that in a free-standing refactoring engine, the tree option is preferable. However, since a lot of the drawbacks with the list option involves Agda communication, the list structure may be easier in the case of an integration with the compiler. In any case, from a code clarity perspective converting between representations seems preferable to trying to write an algorithm for a less suitable representation.

2.3 The pretty-printer

RefactorAgda's pretty-printer is written using the wl-pprint-text library. Apart from pretty-printing, it is in charge of deciding which parts of the original code need to be replaced. It does this by comparing each parse tree in the refactored program with the corresponding tree in the original. If they differ, it will replace the section of text in the code dedicated to this parse tree with a pretty-printed version of the new one. This needs to be done starting from the end so that the text positions stored in the old parse tree match the code string. Newly added elements in the refactored program are specially marked, and will be pretty-printed between the two parse trees it appears between.

2.4 Testing and verification of refactorings

While the data on actual refactoring behaviour may be sketchy on this point, we suggest that the theoretical arguments against a fully-correct refactoring engine do not apply in this case. As for speed, Agda and Baby-Agda have the concept of holes which can replace any element which it is too challenging to compute correctly in that situation. Input size is also not particularly relevant since the current implementation only handles single files. Finally, as regards programmer time, we suggest that since this project is supposed to focus on investigating the refactorings themselves more than on producing a useable tool, spending time on testing and verification is actually the right thing to do. This is supported by the fact that some of the bugs mentioned above involved the tool throwing an exception[8], which can hardly be the result of a deliberate trade-off and may very well be caused by an error of thought rather than a simple implementation mistake.

So how do we test RefactorAgda adequately and effectively? The safest way would be to prove each refactoring correct using some kind of proof engine, but this is rather too ambitious for a master's thesis, because, as mentioned above, we would need a very detailed parse tree to prove any useful properties. Therefore, we will have to stick to giving the tool an input file and a refactoring command and evaluating the output using some suitable oracle. The following is a list of oracles which may be useful for all or some refactorings.

- **Compile:** Compilation is the most obvious oracle – check that the tool produces output which the real Agda type-checker will accept.
- **Import:** Another option is to have another Agda file which imports and uses the one altered by the refactoring engine, and type-check that.
- **Import and test:** A variant on the oracle above, where the importing file consists of tests of the imported file, which are then run.
- **Change:** One may also compare the input and output files to check that the code has only been changed in the expected locations. This could be done with the actual files but it would be more convenient to do the check on the parse

tree because then the same input file could be easily reused to check multiple refactorings.

- **Compare to expected:** We can add the code we expected to see to an extra file and compare that with the output. This is a fair amount of work but can be made a little easier if we parse the reference file and compare the parse tree to the one produced by the refactoring engine. This way the reference files do not need to be continually updated with each trivial change in the pretty-printer.
- **Invert:** Some refactorings are invertible: For example, when you rename a variable and then rename it again to the old name you should get the same program again [5].
- **Reapply:** Some refactorings have the property that applying them multiple times yields the same result as applying them just once.
- **Re-parse:** Run a random refactoring (such as REINDENT) to make sure that the output is something the refactoring tool will accept. This does not technically check the correctness of the refactoring, but it is a useful way of generating extra tests for the parser and pretty-printer.

As a final note, there is another approach which theoretically would be a considerable improvement over using hand-written tests only: automatic testing. This could be done by generating test files, which has been done with a great deal of effort for a subset of Java [19], or by generating refactoring commands involving a particular input file. This could be done systematically on all elements where the refactoring can be applied, as has been done successfully when testing Java refactoring engines [8], or randomly, to simulate a user trying out the engine. While unfortunately there was not enough time to try this on this engine, we believe that generating refactoring commands may be useful.

3 Refactorings

From the above discussions, and from our own experiments, we can conclude that when planning a refactoring we need to have at least the following information for each one.

- **A comprehensible name:** This is necessary to allow the non-initiated to understand what the refactoring does.
- **A comprehensible and concise description:** In case a name is not enough. Here also we need to keep the normal programmer in mind, and avoid terminology incomprehensible to anyone who has not studied refactoring engines in depth.
- **Testing and verification methods** Any applicable testing methods, as well as any invariants which Agda might be persuaded to check for us.
- **A description of the user interface:** This can be far from trivial, as seen below, and is closely related to the problem with incomprehensible refactoring wizards described above. A practical consideration is that the translation from a cursor position or selection to a command the engine can process is rather tedious, and it is best to minimize the number of necessary algorithms as much as possible.

The following list contains refactorings which may be relevant to Agda. They have been obtained partly by looking at refactoring engines for other languages and partly by asking available Agda programmers. A number of them have been implemented wholly or partially for Baby-Agda, and any useful information from this experiment will also be found here: REINDENT FILE, RENAME, REORDER FUNCTION ARGUMENTS, EXTRACT FUNCTION, CONVERT BETWEEN EXPLICIT AND IMPLICIT ARGUMENTS and RE-CASE-SPLIT.

3.1 Reindent file

This type of refactoring is strangely enough not found in either HaRe or Haskell-tools, even though it is not uncommon in other languages: Xcode provides it for Swift and Visual Studio for multiple languages (where it is called “Format document”). It is however likely to be just as useful for Agda as for any other language, for instance when copy-pasting code from elsewhere or incorporating a file into a project, where it is desirable to use the same layout rules everywhere. Effectively, we get this refactoring for free because we need a parser and pretty-printer anyway.

Suitable oracles are **reapply**, to check that nothing vanishes or is added during application, and **compile**, to make sure that the code is formatted properly.

3.2 Rename

Studies show that RENAME is by far the most commonly used refactoring [15] [11], earning it a place in any refactoring engine. Its most interesting feature in terms of implementation is that it can be seen as either one refactoring or several depending on perspective. We see this easily when looking at possible testing methods. Some tests are common to all forms of renaming: for example, the parse tree should be structurally unchanged by the refactoring. Also, all renamings should be amenable to randomized testing by exhaustively enumerating all entities in each test file which are renameable and renaming them to a randomly generated identifier. The output can be tested with the **compile** oracle.

There are, however, ways in which the result differs depending on what kind of identity is being renamed. For example, renaming a function argument should make no difference whatsoever to either the rest of the module or to any importing module, but renaming a function may.

The intuitive way to implement renaming is to treat it as separate refactorings with a set of preconditions for each possible renaming situation which checks whether the renaming is sound. However, even an informal attempt shows that this makes for complicated and error-prone code which may become outdated with each change or extension of the input language without the compiler noticing. Therefore, we have adapted the algorithm developed by Schäfer et al. [18], which treats all renaming as one single refactoring, to use Agda as input and implementation language.

Schäfer et al.'s algorithm was implemented to preserve the following correctness criterion: "Rename refactorings should preserve the invariant that only names are affected by the refactoring, and that each name refers to the same declared entity before and after the transformation." To do this, two functions called *lookup* and *access* are required.

lookup computes for each access of a function name, argument or other things the declaration to which it refers – that is, it tells the compiler which entity is meant by a certain name in a certain scope. The *lookup* function's (partial) inverse is called *access*, which computes for a given entity and a given scope a name which refers to the entity. The inverse is partial because it may happen that there is no way to refer to an entity in a particular scope, for example if the entity is shadowed. It is implemented in the original algorithm by manually coding a reverse to each lookup rule.

These two functions have the invariant that

```
isDefined (access scope declaration) =>
  lookup scope (access scope declaration) == declaration
```

With these two functions in place, renaming can be implemented by changing the name in the declaration, and then adjusting all accesses which might be affected by the change so that they still point to the same declaration as before. If an adjustment fails, the whole procedure is aborted.

For this project, we use the same algorithm but with a different approach to writing the *access* function. First we define a helper function which computes for the declaration in question all possible ways of referring to an entity, independently of any scope. For example, the declaration *true* in listing 3 could be referred to as either *true* or *Bool.true*.

```
data Bool : Set where
  true : Bool
  false : Bool
```

Listing 3: text

Then we define *access* as in listing 4. This type of implementation has the advantage that the invariant automatically becomes true and remains true no matter how we change *lookup*. Like *lookup*, the helper function essentially returns information which Agda needs to keep track of anyway, so if this refactoring is integrated into the compiler, version mismatch bugs are unlikely.

```
access scope declaration =
  allPossibleNames = run helper function with declaration
  for (name in allPossibleNames)
      x = lookup scope name
      if x == declaration then return name
  return error
```

Listing 4: An alternative implementation of *access*

3.3 Extract function

Extract function moves code from an existing function to a new function, so that it can be re-used in other functions. Implementing it has been called refactoring's rubicon, because it requires in-depth analysis of the parse tree [6]. It is considered a commonly used refactoring [7]. However, implementations show substantial usability problems. Two studies come to the conclusion that this type of tool is significantly underused [15] [20]. Several researchers highlight problems with the selection of code to be extracted as problematic [12][20]. Although these studies involve non-functional languages, the problem exists in Agda also. Particularly, a user may accidentally select two arguments to a function without also selecting the function call itself. Our current solution to this problem is to automatically extend the selection to the smallest selectable part of the parse tree which completely covers the selected part.

We will now look at what the refactoring should do when a suitable piece of code has been selected. First, a new function should be added to the module. The precise placement of the new function is something of an issue because of mutual recursion. One very simple way of solving this is to place the new function immediately above the old function definition (and therefore below the old definition's signature). However, this would very often lead to the definition being separated from the signature unnecessarily, so a more advanced implementation would check for mutual recursion and place the new function above or below the old signature accordingly.

```
newFunction : newArguments -> newResultType
newFunction necessaryVariables = right-hand side
```

Listing 5: The function parts we need to find

Listing 5 shows a skeleton of the desired new function. This function will then be called in place of the extracted code in the old function. While some of the necessary components are straightforward, others are difficult to obtain because we need to do considerable type analysis. By far the trickiest part of this refactoring is how to find all necessary arguments to the new function. Happily, we can get some help from the Agda compiler by replacing the part to be extracted with a hole and feeding the result to Agda. There are two kinds of information we can get in this way. The first is the type of the expression to be extracted, which is equal to the result type of the new function and to the type of the expression which needs to go into the hole. The other is the hole's context, which consists of a list of all variables with their associated types in the order in which they appear in the type signature of the old function.

```
 func : (a : Set) -> {b : Set} -> a -> b -> a
 func b c d = {!   !}

 --The hole's context:
b : Set
.b : Set
c : b
d : .b
```

Listing 6: A function with a hole and the hole's context

In listing 6 we can see that the context has all the relevant variables listed in order of their appearance in the type signature. We can further observe that if we construct the new function with a hole for a right-hand side like *func*, the context of that hole must be a subset of this context. Also, the order of this subset should be undisturbed, because some of the variables will depend on others before them.

So how do we find the right subset? There are three reasons we might want to keep a variable: it appears in the expression to be extracted (which will later make up the right-hand side of the new function), in the type of that expression, or in the type of any variable which has already been shown to be necessary. Consequently, we filter the context according to these criteria starting from the end, because no variable can be needed by the types of variables preceding it.

```
recover : Context -> Context
recover ((name : type) :: restOfContext) = do
  result <- recover restOfContext
  if name does not have a dot before it
  then
    return ((name : type) :: result)
  else
    if name (without the dot) appears anywhere in result
    then
      return ((newName : type) :: replace all occurrences
        of name-with-dot with newName in result)
    else
      return ((name-minus-dot : type) :: replace all occurrences
        of name-with-dot with name-minus-dot in result)
```

Listing 7: Recovering types and variables from a context

The next and more complex part of the transformation is to recover the new function's type and variables from the context. As a matter of fact, since Baby-Agda does not have where or let statements, a context is effectively a description of the arguments in the type signature. There are just a few little things we need to fix, which we will show here by using *func* in listing 6 as an example. For the general algorithm, see listing 7. First, we observe that one of the variables $.b$ is not in scope of the hole, as shown by the prefixed dot. It is a variable used only in the type signature. Since this dot will not be accepted by the compiler, we need to remove it – but we have another variable b which also appears in one of the types, and we cannot be sure that the two will not clash. Therefore, we will need to rename one of them.

When picking a variable to rename, we observe first that there can be at most two variables with the same name, of which one is in scope and the other is not. This is because the Agda compiler will give unique names to any variables sharing the name of another in the type signature, and enforces unique variable names in the definition. We also observe that only the variable which is in scope can potentially turn up in the extracted expression. Since we do not want to do the extra work of checking the expression, we will choose to rename the variable which is not in scope to a unique identifier.

Now there are only three things left to do. First, we need to remove all the dots, which is straightforward now we no longer risk clashes. Then we need to decide whether to make the arguments explicit or implicit in the type signature. In this case, we simply decide that whenever an argument has a name which was in scope originally, we make it explicit; otherwise, it will be implicit. Finally, we need to get the list of variables on the left-hand side of the function definition, which we can get easily by reusing the names of all the explicit arguments.

In listing 8, we give some pseudocode summarizing how to get hold of all the necessary function parts first mentioned in listing 5.

```
newFunction = automatically generated new identifier
newArguments = recover (filter context)
newResultType = type of extracted expression, given by Agda and
   with any renamings applied during recover applied to it
necessaryVariables = map fst newArguments
right-hand side = the selected expression, unchanged
```

Listing 8: How to get hold of the function parts from listing 5

The very last step is to put the entire left-hand side of the new function in the place where the expression to extract once was. This requires no extra changes because we made sure not to rename anything that was in scope in the body of the old function.

A point to note is that since we only interact with very small parts of the code, and most of that involves writing new code, we do not actually need a specialized parser or anything else. The information Agda extracts from the code, including line numbers, should be sufficient. This does presuppose, though, that we always want to send functions defined in where-clauses as arguments, which might not be the expected behaviour if the function is only used once.

Extract function can be tested with the **import** and **import and test** oracles, to make sure that the code still behaves as before.

3.4 Expression-changing refactorings

In this section, we will examine a certain group of refactorings which includes among others adding, removing or reordering arguments and converting between explicit and implicit arguments. Although the refactorings are fairly different when looking at what they do to the type signature being refactored, they all have one thing in common: they need to find all expressions where the function (or constructor, or similar construct) is being used and transform them in a particular way, which may be different for each refactoring.

Obviously, we do not wish to write code for this multiple times over, so an important task is to define an algorithm which can do this for any such refactoring. Such an algorithm needs to go through all expressions where the entity being refactored is in scope, figure out which ones need to be changed, and apply the change. For simplicity, we will ignore the fact that some kinds of expressions may not turn up in certain places: for example, there is really no point looking for an identifier pointing to a function in places reserved for pattern matching.

Although the idea is easy in theory, it is not so in practice. For example, let us consider the code below and imagine that we want to switch the order of the two inputs *a* and *x*. In *use1*, the usage is obvious. We have the function identifier (let us pretend that all names are unique; in reality, we would use engine-generated unique labels) at the very left of the expression, and then we have a sub-expression for each argument. Incidentally, while it may sound as if this is a simplification due to Baby-Agda, which does not have mixfix notation, in reality of course the engine must rewrite the code to prefix notation before refactoring.

However, in the other cases we need to think a little more. In *use2*, things are still fairly straightforward, because we still have the function name at the start of the expression and slots for the arguments we are actually manipulating. *use3* is a more difficult case, because we do not know what we should write in the first slot here. This situation is particularly treacherous because the arguments to be switched have the same type, so the compiler will not alert the programmer. Therefore, we need to replace this portion of code with something which cannot be ignored – a hole. At the same time, we would prefer that the original code does not vanish in case the programmer needs it to refresh their memory, so we should put the original code in the hole. At this point we can note that in full Agda, we would be able to insert a lambda expression instead.

Then, we have example *use4*. This is different from all the others in that one of the subexpressions also needs to be refactored, which shows that we cannot rely on the identifying function name to be at the front of the main expression.

Finally, there is another consideration not shown here: implicit arguments. Implicit arguments are arguments which may or may not appear in an expression. From the perspective of the refactoring engine, their appearing or not appearing is completely random. Therefore, the algorithm needs to know what to do in the case of an omitted argument.

```
func : (a : Bool) -> (x : Bool) -> (b : Bool) -> Bool
func a x b = a

use1 : Bool
use1 = func true false false

use2 : Bool -> Bool
use2 = func true false

use3 : Bool -> Bool -> Bool
use3 = func true

use4 : Bool
use4 = func true false (func false true false)
```

Listing 9: An example function with some calls

We have already mentioned that expression-changing refactorings benefit from a data type for expressions which resembles a list, because then we can see immediately if we need to refactor the expression. What we want to do now is to produce a refactoring-specific function of the type *List Expression -> List Expression*, which can then be applied to all uses which need to be refactored without regard to what kind of refactoring we are doing.

```
funcToApply : List ArgumentTypes -> Nat ->
        (List Expression -> List Expression)
funcToApply (x : y : cs) 0 (a : b : es) = handle cases use1 and use2
funcToApply (x : y : cs) 0 (a : []) = handle use3
funcToApply (x : cs) (suc n) (a : es) = a : (funcToApply cs n es)
...
```

Listing 10: A simplified part of the function to apply for our argument-switching example

In listing 10 we give a partial and highly simplified version of a function which can be partially applied to produce a function of the desired type. The first argument here gives relevant information from the type signature of the function to be refactored, particularly whether the argument in question is implicit or explicit. We go through this list from the start, matching it up at each step with the list of expressions, which is the use we are trying to refactor. The second argument tells us when we have reached the spot where we are supposed to make a change. In reality, this function is fairly complicated because it needs to handle omitted arguments. On the positive side, this type of implementation makes it immediately obvious which line needs to be fixed in case of a bug. Also, Agda will tell us if we have forgot a use case as long as we refrain from using catch-all statements.

3.4.1 Convert between explicit and implicit parameters

Let us start with the simplest of all expression-changing refactorings, which refactors only one argument at a time with no relationship whatsoever with any other parts of the code. With Agda's syntax, any argument can be written either as explicit or implicit. In listing 11 we have an example of an explicit argument.

```
id : (A : Set) → A → A
id A a = a

idZero : ℕ
idZero = id _ 0

idOne : ℕ
idOne = id ℕ 1
```

Listing 11: Code with an explicit argument

In *idZero*, the compiler is instructed to figure out the explicit argument for itself, whereas in *idOne* the argument is passed explicitly. This argument can be neatly converted to an implicit one (see listing 12). The explicit instruction in *idZero* has been removed entirely, whereas in *idOne* the syntax for explicitly specifying an implicit argument is used.

```
id : {A : Set} → A → A
id {A} a = a

idZero : ℕ
idZero = id 0

idOne : ℕ
idOne = id {ℕ} 1
```

Listing 12: Listing 11 after *convert to implicit argument*

The same operation also works in reverse and should produce the same code again. In this case we could the **invert** oracle, which states that any application of *toExplicit (toImplicit x)* or *toImplicit (toExplicit x)* will have a result identical to the input.

3.4.2 Reorder function arguments, constructor arguments, data type parameters or indices

Though this section will use the reordering of function arguments as an example, reordering constructor arguments, data type parameters or data type indices works in the same manner. The order of arguments in functional languages has a substantial impact on legibility, particularly because of partial application. In Agda, this problem is trickier than usual because of dependent types. Consider for example the implementation of the identity function in listing 13. In this case, reordering is not possible because the type of the second argument depends on that of the first. This kind of situation cannot appear in non-dependently-typed languages.

```
id : (A : Set) -> (a : A) -> A
id A a = a
```

Listing 13: The identity function

One simple way of implementing this refactoring is a simple *push argument* refactoring. This is a useable user interface all by itself, which could be made even more useful if the cursor moves with the argument so that several push commands can be easily done in a row. Fancier user interfaces would also be possible, because more complex reorderings can always be broken down into a series of pushes.

This refactoring can also be tested with **invert**, by pushing an argument in a particular position twice.

3.4.3 Add argument or index to data type, or add argument to a function

Let us look at the list datatype shown in listing 14, and let us say that we would like to add an index type for length to this list type, and accordingly change the code to what is shown in listing 15. This listing also shows that all expressions starting with *Vec* are to be transformed in a regular fashion by inserting a hole in the last position.

```
data Vec A : Set where
  [] : Vec A
  cons : A -> Vec A -> Vec A
```

Listing 14: A list datatype

```
data Vec A : ℕ -> Set where
  [] : Vec A ?
  cons : A -> Vec A ? -> Vec A ?
```

Listing 15: The list from listing 14, transformed to a vector

The refactoring is called by placing the cursor in the declaration or a use of it and calling the refactoring, which then asks for the type to be added. Suitable oracles are **import** and **import and test**, to check that the desired argument has indeed been added. Further, since these refactorings are the reverse of the *remove* refactorings below, implementing them makes it possible to use the **invert** oracle on several refactorings.

Adding an argument to a function works in exactly the same way, except that functions cannot be case split on and therefore we need not check quite as many places.

3.4.4 Add data type parameter

Seeing that we have been able to use our expression-refactoring algorithm for adding data type indices, we would like to do the same thing with parameters. In fact, it would be possible to do that: we could refactor listing 16 into listing 17 using the same algorithm as above.

```
data Stuff (A : Set) : Set where
  makeStuff : Stuff A
  makeStuff2 : Stuff Bool -> Stuff A
```

Listing 16: A datatype wanting another parameter

```
data Stuff (A : Set) (B : Set) : Set where
  makeStuff : Stuff A ?
  makeStuff2 : Stuff Bool ? -> Stuff A ?
```

Listing 17: Datatype 16 with a parameter added using instructions from *add index*.

While there is nothing particularly wrong with this result, we could give the programmer a little more help, because we know that both places reading *Stuff A* must be changed to *Stuff A B*, because there are no other options for these positions. It would take a fair amount of extra writing but little mental effort to produce 18.

```
 data Stuff (A : Set) (B : Set) : Set where
   makeStuff : Stuff A B
   makeStuff2 : Stuff Bool ? -> Stuff A B
```

Listing 18: A more helpful version of listing 17

This refactoring is tested and used in the same manner as ADD DATA TYPE INDEX.

3.4.5 Extract argument

This refactoring works essentially like EXTRACT METHOD, except that the selected code is replaced by a new argument. Consider, for example, listing 19. The function *example* adds a list of length (f a b) constructed with one function to a list of the same length constructed with another function. Now suppose we want to extract *f m n* in this function to an argument, as shown in listing 20.

```
replicate : {A : Set} -> (n : Nat) -> A -> List A n
f : Nat -> Nat -> Nat
f-list : forall a b -> List Nat (f a b)

example : (a : Nat) -> (b : Nat) -> List Nat (f a b + f a b)
example m n = replicate (f m n) 0 ++ f-list m n
```

Listing 19: Some example code.

```
example m n o = replicate o 0 ++ f-list m n
```

Listing 20: Listing 19 after performing EXTRACT ARGUMENT on *f m n*

The difficult part about this refactoring is figuring out how to change the type signature. We have added a new argument with a known type, so we start by adding that to the function type, as shown in 21. This, however, will not compile, because Agda has no way of knowing that *c* should be the same as *f a b*.

```
example2 : (a b c : Nat) -> List Nat (f a b + f a b)
```

Listing 21: A suggested type signature after refactoring

One option would be to tell Agda this explicitly, as in listing 22. This code will type-check. We have simply added another argument which consists of a proof that *c* equals *f a b*. We have also added this argument to the function definition, and made sure to use the constructor *refl* rather than a random name because Agda will not understand the code otherwise. A point to note is that it would be useful for the refactoring to check if ≡ (the one from Agda.Builtin.Equality, not a custom one with the same name) is in scope and add an import statement if it is not.

```
example2 : (a b c : Nat) -> c ≡ f a b -> List Nat (f a b + f a b)
example2 m n o refl = replicate o 0 ++ f-list m n
```

Listing 22: A type signature which will compile

However, is this really the type we want? One could imagine a few other options. For example, we could remove the information that both parts of the list should have the same length, thereby avoiding the need for any extra possibly unwanted argument, as in listing 23. For example, we could refactor *example* to the following:

```
example3 : (a b c : Nat) -> List Nat (c + f a b)
```

Listing 23: Alternative to listing 22

At this point, we do not really have any idea of what would be the most useful to Agda programmers. There, is, however, one thing we can look at: the kind of code we are actually capable of producing. If we consider only the typing capability of the refactoring engine itself, we would get listing 24. If instead we choose to ask Agda what the type of the result should be, we would get either listing 23 or 25, depending on whether we have Agda deduce the type of the expression or ask it to fill in the hole in listing 24.

```
example3 : (a b c : Nat) -> {! !}
```

Listing 24: Another alternative to listing 22

```
 example3 : (a b c : Nat) -> List Nat (c + b)
```

Listing 25: Yet another alternative to listing 22

Finally, we point out that no matter which option is chosen here the refactoring should take care to find all usages of *example* and add something suitable in the place of the new argument. The simplest thing to put in would be a hole (which we can easily do with the instruction given for ADD ARGUMENT), but since the new argument is a known function of already given arguments, it might be possible to calculate the intended result directly, if that should prove to be desirable.

3.4.6 Remove argument or data type index

In its simplest definition, REMOVE ARGUMENT is just as simple as ADD ARGUMENT, and is handled in the same fashion. We can, however, in the spirit of the topic of this thesis add some bells and whistles to this refactoring in the form of checking whether the argument or index is safe to remove. In the case of both arguments and indices, the item cannot be removed if any later arguments or indices depend on it.

In the case of arguments particularly, the argument also cannot be removed if it is used in the body of the function or if it has been case split on. This is the case even if the case split results in only one constructor – if this is not immediately obvious consider listing 22 again.

In the case of indices, we also need to check that the index is, in fact, a parameter which happens to be in index position. This is the case when for each constructor, any value will work for the index. In listing 26 you will find an example of an index which can not be removed.

```
  data Eq {A : Set} (x : A) : A -> Set where
     refl : Eq x x

subst : {A : Set} -> (P : A -> Set) ->
        {x : A} -> {y : A} -> Eq x y -> P x -> P y
subst P refl px = px
```

Listing 26: A datatype with a non-removable index

This refactoring has a certain relationship to CHANGE OPERATOR FIXITY (a variant of renaming an operator), because removing an argument may cause one of the underscores in the name to vanish. The easiest way of dealing with that is to first perform a renaming operation and then remove the argument.

3.5 Inline function definition

This is essentially the opposite of EXTRACT FUNCTION. One of its uses is to get rid of functions which turn out to have been used only once. A more interesting use case is when attempting to switch the arguments in *f* in listing 27.

```
F : Set
F = Bool -> Bool

f : Bool -> F
f a b = {!   !}
```

Listing 27: Argument pushing which is not possible

Here we must inline the definition of *F* before making the change, even though in the definition of *f*, we get a clear impression of a function of two arguments.

3.6 Organize imports

This refactoring can remove unused imports and restrict imported entities to those actually used in the file in question. Another interesting functionality would be suggesting suitable imports when trying to use something that is not in scope.

The **compile** and **change** oracles should be sufficient for testing this refactoring, since the refactoring should only change import statements and any unsuitable change will cause the file to fail to compile.

3.7 Split a module

It may happen quite frequently that a module grows so large that it needs to be split into two. In that case, the two resulting modules must each import some but probably not all the modules imported by the original module, and the "lower" module may need to import its sister module.

The **import** oracle is useful here, among other things to check that the right parts of the original module end up in each new module.

3.8 Abstract to module parameters

At some point during development, it is possible that the programmer gets the idea to generalize the module by replacing a type or term by a module parameter. Consider, for example, listing 28, and imagine that we get the idea that we could very well combine other things than numbers in the same way, such as strings. So we try to change it to listing 29.

```
module AbstractModuleParameter where

combine : List ℕ -> ℕ
combine [] = 0
combine (x :: xs) = x + combine xs
```

Listing 28: Module to abstract

```
module AbstractModuleParameter (A : Set) where

combine : List A -> A
combine [] = 0
combine (x :: xs) = x + combine xs
```

Listing 29: Listing 28 after abstracting

There are two problems with this code: One is the numeric value 0, the other is the function _+ _ on the last line, both of which are specific to numbers. At this point there are two things we could do with these. The first option is to replace the offending entity with a hole, or better yet, with a function whose right-hand side is just a hole, to avoid getting a huge number of holes for the same reason. This has been done in listing 30 with 0. The other option is to abstract over the entity as well, as has been done with _+ _ in the same listing. As it is not obvious what should be done in each instance, it might be wise to investigate any possibility of adding interactive capabilities to the refactoring engine.

```
module AbstractModuleParameter (A : Set) (f : A -> A -> A) where

z : A
z = {! 0 !}

combine : List A -> A
combine [] = z
combine (x :: xs) = f x (combine xs)
```

Listing 30: Another abstracted version of listing 28

One point of particular importance is that finding the entities something needs to be done with requires considerable typing capabilities. This refactoring should therefore be somehow integrated into Agda.

Import and test or plain **import** are the most suitable oracles for this refactoring, since it is probably safe to assume that errors of the type “failed to abstract enough stuff” are likely.

3.9 Change associativity

Consider the _*s*_ operator. Let us say that it is left-associative, that is *3 s 4 s 5 == (3 s 4) s 5*. However, it is possible that one finds that such an operator would be more useful if it was right-associative. In that case, in order to make the expression above keep its original meaning, the parentheses need to be changed. This is more easily and safely done by a refactoring engine than by hand, owing to the fact that the compiler is not going to be of the least help in finding places where the associativity change causes a change in meaning.

This refactoring should be tested using the **import and test** oracle, since it is extremely likely that the engine will produce compiling but buggy output.

3.10 Convert between constructor and record syntax

When a record has a constructor, it is possible to use either constructor or record syntax to make an element of the type. For example, either of the two definitions in listing 31 will produce the same result.

```
twice : {A : Set} -> (a : A)-> A × A
twice a = a , a

twice a = record {proj₁ = a ; proj₂ = a}
```

Listing 31: The same record in two flavours

As programmers are likely to consistently prefer one version over the other, this refactoring is really a pretty-printing matter. Both versions can be parsed to the same parse tree, and the pretty-printer is then instructed on how to print it.

The **compile** oracle is likely sufficient for this refactoring, since the parser and pretty-printer are automatically tested when all other tests are run.

3.11 Add or remove constructor from data type

Case splitting seems to be one of the most useful capabilities of Agda in terms of saving programmer time. However, it is frequently the case that a data type needs to be changed after it has been case split on multiple times, resulting in a lot of tedious typing. It would be very convenient if the refactoring engine could handle this for us.

There are two possible user interfaces for these refactorings. The first involves placing the cursor in the data type and giving the engine the name and type of the constructor to be added. For deletion, the user would put their cursor into the constructor to be deleted. The other interface is what we can call RE-CASE-SPLIT: we make changes to the datatype first, then call the refactoring.

The second interface can be partially implemented for addition simply by reading the error messages output by Agda. Error messages about incomplete pattern matching contain not only the line and column number of the start of the case split, but also the complete left-hand side. It is easy to parse these messages and make the necessary changes. However, this method can not easily be extended to pattern matching in lambdas or do-blocks, which can be in the middle of other code rather than neatly on a line of its own.

This interface cannot currently be extended to removing constructors, even though the error message given by Agda contains the necessary information. This is because Agda will stop parsing when it encounters such an error, rather than output a list of all errors it detects, as GHC does for Haskell.

The first suggested interface has the advantage that it can be made to work on lambdas and in do-blocks as well. However, it has the drawback of being a great deal more work, because you would need to use a full refactoring engine.

Compile should be a sufficiently good test, since the compiler will notice errors in the data type if it is used in sufficiently many situations in the test file.

3.12 Generate FFI bindings

In some projects, such as the current one, large data types are shared between Agda and Haskell code. Writing the necessary bindings by hand is rather mechanical and well suited for automatic refactoring. This refactoring would need to detect if there is already an FFI binding for the current datatype, and add or update as appropriate. If the data type contains dependent types, no bindings can be generated and an error message should be returned instead.

This is one refactoring which would be better placed in the Agda compiler than in a separate refactoring engine. Since it only creates or changes pragmas (which cannot contain comments anyway), there is no need for a special parser. Also, we need to detect data types which it is not possible to generate bindings for, for which we need Agda's typing capability.

4 Conclusions

So how close are we to having usable implementations of any or all of the above refactorings? To answer this question, we first note that the main use of a separate engine with a comment-handling parser and a syntactic-sugar-preserving scoping function is when we make changes to or move existing code. If we are simply adding or deleting lines, a normal compiler front-end will do. While a full refactoring engine seems unlikely in the foreseeable future, refactorings which do not need one could reasonably easily be added to the existing Agda compiler. Among these, we have already discussed GENERATE FFI BINDINGS and ADD OR REMOVE CONSTRUCTORS FROM DATA TYPE. EXTRACT FUNCTION is another refactoring which makes few changes to the code, and most of that is in adding newly generating code. The only part involving comments is in moving the selected code, and since we do not need to make any changes to the moved code, we can just copy-paste it comments and all. The same thing goes for its reverse INLINE FUNCTION DEFINITION, although it rather depends on whether you want the comments copied and if you want to do any simplifying. The part of ORGANIZE INPUT which deals with suggesting imports, as well as SPLIT A MODULE, can also be done with just a compiler parser. This is convenient since we can re-use Agda's mechanism for dealing with multiple modules.

Finally, we would like to discuss whether it was worth using Agda in this project, and if there is anything we could have improved in the interplay between the Agda and Haskell parts of the program. We note that it is generally difficult to estimate how many bugs were prevented by a particular approach without having tried any others, but we have successfully identified and implemented a few properties of the internal engine state which could not have codified in Haskell. For example, we can check that the internal state does not contain references to scopes or variables which do not exist. We have also had some useful debugging help from Agda's termination checker.

As regards the workload, there is a little bit of overhead involved in getting the connection between Haskell and Agda set up. On the other hand, since refactoring essentially involves multiple passes through an enormous data structure, Agda's case splitting functionality saves a considerable amount of typing. Without having made an exact study of how much time is saved, we estimate that using Agda actually ends up being less work than using Haskell.

Finally, there is one thing we might have done differently: the connection between Agda and Haskell. Currently, this is handled by an automatic translator from Haskell to Agda, but if the refactoring GENERATE FFI BINDINGS had occurred to us earlier we would have started from that end. There would not have been any extra starting overhead, we could have got a head start on a useful refactoring, and once some data-type-altering refactorings have been finished, we could use those to speed up development of the others, and get some free testing in. There is no such benefit in having the main parse tree – the one we make manual changes in – in Haskell because there are no functioning refactoring engines for Haskell.

Bibliography

[1] HaRe. https://github.com/RefactoringTools/HaRe.

[2] haskell-tools. https://github.com/haskell-tools/haskell-tools.

[3] Robert Bowdidge. Performance trade-offs implementing refactoring support for Objective-C. *Proc. 3rd WRT*, 2009.

[4] Dustin Campbell and Mark Miller. Designing refactoring tools for developers. In *Proceedings of the 2nd Workshop on Refactoring Tools*, page 9. ACM, 2008.

[5] Brett Daniel, Danny Dig, Kely Garcia, and Darko Marinov. Automated testing of refactoring engines. In *Proceedings of the the 6th joint meeting of the European software engineering conference and the ACM SIGSOFT symposium on The foundations of software engineering*, pages 185–194. ACM, 2007.

[6] Martin Fowler. Crossing refactoring's Rubicon. https://www.martinfowler.com/articles/refactoringRubicon.html.

[7] Martin Fowler and Kent Beck. *Refactoring: improving the design of existing code*. Addison-Wesley Professional, 1999.

[8] Milos Gligoric, Farnaz Behrang, Yilong Li, Jeffrey Overbey, Munawar Hafiz, and Darko Marinov. Systematic testing of refactoring engines on real software projects. In *European Conference on Object-Oriented Programming*, pages 629–653. Springer, 2013.

[9] Miryung Kim, Thomas Zimmermann, and Nachiappan Nagappan. A field study of refactoring challenges and benefits. In *Proceedings of the ACM SIGSOFT 20th International Symposium on the Foundations of Software Engineering*, page 50. ACM, 2012.

[10] Huiqing Li. *Refactoring Haskell Programs*. PhD thesis, University of Kent, 2006.

[11] Gail C Murphy, Mik Kersten, and Leah Findlater. How are Java software developers using the Elipse IDE? *IEEE software*, 23(4):76–83, 2006.

[12] Emerson Murphy-Hill and Andrew P Black. Breaking the barriers to successful refactoring: observations and tools for extract method. In *Proceedings of the 30th international conference on Software engineering*, pages 421–430. ACM, 2008.

[13] Emerson Murphy-Hill and Andrew P Black. Making refactoring tools part of the programming workflow. *ACM Transactions on Software Engineering and Methodologies*, 2008.

[14] Emerson Murphy-Hill and Andrew P Black. Refactoring tools: Fitness for purpose. *Computer Science Faculty Publications and Presentations*, 109, 2008.

[15] Emerson Murphy-Hill, Chris Parnin, and Andrew P Black. How we refactor, and how we know it. *IEEE Transactions on Software Engineering*, 38(1):5–18, 2012.

[16] Jeffrey Overbey. *A toolkit for constructing refactoring engines*. PhD thesis, University of Illinois at Urbana-Champaign, 2011.

[17] Gustavo H Pinto and Fernando Kamei. What programmers say about refactoring tools? an empirical investigation of stack overflow. In *Proceedings of the 2013 ACM workshop on Workshop on refactoring tools*, pages 33–36. ACM, 2013.

[18] Max Schäfer, Torbjörn Ekman, and Oege De Moor. Sound and extensible renaming for Java. *ACM Sigplan Notices*, 43(10):277–294, 2008.

[19] Gustavo Soares, Rohit Gheyi, Dalton Serey, and Tiago Massoni. Making program refactoring safer. *IEEE software*, 27(4):52–57, 2010.

[20] Mohsen Vakilian, Nicholas Chen, Stas Negara, Balaji Ambresh Rajkumar, Brian P Bailey, and Ralph E Johnson. Use, disuse, and misuse of automated refactorings (extended version). In *Software Engineering (ICSE), 2012 34th International Conference on*, pages 233–243. IEEE, 2012.