,

REFACTORING HASKELL PROGRAMS

a thesis submitted to

The University of Kent

in the subject of computer science

for the degree

of doctor of philosophy.

By

Huiqing Li

September 2006

Contents

List of Tables

List of Figures

Abstract

Refactoring is the process of improving the design of existing programs without changing their external behaviour. Behaviour preservation guarantees that refactoring does not introduce (nor remove) any bugs. If applied properly, refactoring can make a program easier to understand or modify. While it is possible to refactor a program by hand, tool support is considered invaluable as it is more reliable and allows refactorings to be done (and undone) easily.

Taking the Haskell programming language as a case study, this research investigates the prospects of refactoring in the context of functional programming languages, both to complement the existing work on refactoring within other programming paradigms, such as OO, and to make refactoring techniques and tools available to functional programmers.

By building a tool for refactoring Haskell programs, we addressed the key issues involved in tool support of refactorings, including the supporting technologies for building refactoring tools, analysing the side-conditions and transformation rules of individual refactorings for behaviour preservation purpose, and the preservation of the layout and comments of the refactored programs.

Along with the development of the Haskell refactorer, we also investigated the specification and verification of validity of refactorings. This helped us to clarify the definition of refactorings, to improve our confidence in the behaviour-preservation of refactorings, and to reduce the need for testing.

Acknowledgements

First and foremost, my deepest thanks to my supervisor, Professor Simon Thompson, for his invaluable advice and support. His guidance is always insightful, and his patience is highly appreciated. As a supervisor, he shares with me his knowledge in this field, but his encouragement goes far beyond it. Secondly, my grateful thanks to EPSRC's support on the *Refactoring Functional Programs* project, which made it possible for me to work and study here at the University of Kent.

Many thanks to Dr. Claus Reinke for his comments on the thesis. As a great colleague and friend, his profound knowledge and insight into the language design area, his wisdom and humour have always impressed me. I learnt much from our usual discussions, weekly project meetings, and from other feedbacks he gave me.

I thank the members of my PhD supervision panel, Dr. Eerke Boiten and Dr. Stefan Kahrs, for the valuable feedbacks and suggestions they given me during the panel meetings. I also thank the function programming interest group here at Kent for providing a very pleasant environment for discussing research ideas, experiments, and learning new stuff.

I thank the Programatica and Strafunski development teams for allowing us to use their tools to build HaRe, and for their continuous support during the development of HaRe.

I wish to especially thank my husband, Shaomin, for his continuous love, support and encouragement, and my lovely daughter, Siqi, for all the happiness she has brought to me and our family. Without them, all my accomplishment would mean nothing.

Finally, thanks to my parents and parents-in-law for everything.

Chapter 1

Introduction

1.1 Program Refactoring

Refactoring [34] is a disciplined technique for transforming program source in such a way that it changes the program's internal structure and organisation, but not external behaviour. The key characteristic that distinguishes refactoring from general code manipulation is its focus on structural changes, strictly separated from changes in functionality. Functionality-preservation guarantees that refactorings do not introduce (nor remove) any bugs or invalidate any existing tests that do not rely on program's internal structure. While each refactoring performs a small-scale program transformation, a sequence of refactorings can produce a significant restructuring. Refactoring can be used for improving code design and quality, and for increasing code reuse and productivity [35].

Refactoring can occur at any level of a program. For example, renaming a local variable only has impact in the entity where the variable is declared; whereas, renaming a global variable might potentially affect the whole program.

There are both elementary refactorings and composite refactorings. An elementary refactoring can not be decomposed into simpler refactorings; whereas, a composite refactoring can be decomposed into a series of elementary refactorings.

Refactoring is a practice that programmers do very often as they modify existing code, even without using the name *refactoring*. After establishing a working

piece of code, they may refactor it to improve its design; before changing the functionality of an existing software system, they may refactor it to make it more manageable. The term *refactoring* was first introduced in the work of W. Opdyke and R. Johnson [78] in 1990.

In the past decade, software development approaches experienced a shift from the classical waterfall model [95], where analysis is fully completed before design and design is fully completed before implementation, to evolutionary approaches [14], where a simple version of what is required is extended iteratively to build a more complex system and program restructuring is an important stage during each evolution phase. As a consequence, refactoring has been identified as central to software development and maintenance, especially within the Software Engineering (SE) and Object-Oriented (OO) communities [39, 79, 14]

Until recent years, refactoring had been typically done manually or with the help of text editors with search and replace facilities. Manual refactoring is tedious, error-prone and costly. It depends on extensive testing to ensure that functionalities are preserved [35]. While testing can show up bugs and improve confidence, it can not prove correctness. Therefore, refactoring tools for various programming languages, which can help programmers perform refactorings automatically, and be proven functionality-preserving, are highly desirable. Two major activities involved in performing a refactoring are *program analysis* and *program transformation*. Program analysis checks whether certain side-conditions are met by the program under refactoring in order for the refactoring to preserve behaviour, and collects information needed by the program transformation phase; program transformation carries out the program restructuring step of a refactoring. Both program analysis and program transformation are amenable to automation, as manifested by the rich collection of existing work in many areas of software engineering including compiler construction [5], program understanding [114], debugging and testing [88], program slicing [117], program maintenance and reverse engineering [47], etc.

Since the first successful and most notable refactoring tool the *Refactoring*

Browser [93], which supports Smalltalk refactoring, there has been a growing number of refactoring tools [34] for a variety of languages such as Java, C, C++, C#, Python and UML. For example, the most recent release of the IntelliJ IDEA [48] claims to support 35 Java refactorings.

However, tool support for refactoring functional programs has been explored much less, and there was no such a practically usable tool for functional programmers at the time we started our *Refactoring Functional Programs* project.

1.2 Functional Programming Languages

Functional programming languages are a class of languages designed to reflect the way people think abstractly rather than the underlying machine [44]. They were originally based on Lambda Calculus [20, 10], which embodies a simple model of computation that provides a formal way to describe function and expression evaluation. This kind of programming languages view computation as the evaluation of mathematical functions. Each function takes zero or more parameters as input, and returns a single value as the function's output. In many functional languages, such as Haskell [53], functions are treated as first-class citizens, which means that functions can be parameters to other functions and can be the return values of other functions, and functions of this sort are called *higher order functions*. Another powerful feature of some functional programming languages such as Haskell is *lazy evaluation*, which delays the evaluation of an expression until the result of the evaluation is needed. Higher order functions and lazy evaluation together provide good support for modular programming, and make functional programs smaller and easier to write than programs in conventional programming languages [45]. In contrast to traditional imperative programming languages such as C or Pascal, there is no explicit memory allocation and no explicit variable assignment operation in pure functional languages, and the `=` operator is treated as an expression of definitional equality, i.e. the left-hand side is defined to be the right-hand side in all circumstances. The result of applying a function to a

given set of parameters will always be the same, no matter when, or where, the function is evaluated. This *referential transparency* feature makes verification of programs written in functional programming languages easier.

Apart from higher order functions and lazy evaluation, modern functional languages such as Haskell [53], ML [86] and Miranda [108] also manifest features including equations and pattern matching, type systems with Hindley-Milner type inference[42, 75] and type classes, etc. Furthermore, monadic programming [116] adds features such as hidden state and explicit sequencing like I/O to Haskell without violating its pure functional semantics.

1.3 Refactoring Functional Programs

Refactoring Functional Programs is a project carried out at the Computing Laboratory of University of Kent. The aim of this project is to investigate the prospects of refactoring in the context of functional programming languages, both to complement existing work on refactoring with a functional programming perspective, and to make refactoring techniques and tools available to functional programmers. We take Haskell as a concrete case-study, and explore the application of its results to other functional languages.

Functional programming languages differ from imperative and object-oriented languages in both theory and practice. While some functional refactorings, such as *renaming*, *deleting an unused parameter*, etc, have direct OO counterparts and involve similar program analysis and manipulation, for many other functional refactorings, the correspondence to their OO counterparts is either less obvious, or the involved program analysis and manipulation are quite different. For instance, the correspondence between *replacing a multiple-equation definition by a single-equation definition containing a* `case` *statement* in the functional programming diagram and *inlining a virtual method using a* `case` *statement* in the OO context is less obvious, and these two refactorings also involve quite different program manipulation. Furthermore, there are refactorings which are unique to the

functional programming paradigm, such as *introducing a monadic computation of a particular expression* in a Haskell program. The implementation of a refactoring tool for a real world functional programming language will therefore not be a simple re-implementation of an existing refactoring tool for a different language.

Similar to other program transformations [25, 83], refactoring is ultimately based on language semantics and program equivalence. The clean semantics of functional programming languages and the rich theoretical foundation for reasoning about programs make it practicable to rigorously specify and prove the validity of refactorings, while this is not often practical for existing imperative and OO languages because of the side effects and lack of formal semantics. From this point of view, functional programming languages should be more suitable for refactoring. A case study of refactoring functional programs by S. J. Thompson and C. Reinke also shows that the 'first code, then revise' style of programming approach is natural for writing functional programs in practice [98].

Program transformation in functional programming has been studied extensively, in the context of program optimisation and efficient program derivation [84, 11, 12, 73, 8, 90]. In the above context, a functional program transformation system usually takes a specification as a starting point, then transforms the specification into a program of acceptable efficiency by applying a sequence of behaviour-preserving transformations. This kind of program transformation is *vertical* in the sense that it addresses a program's control or data flow. The program transformation inherent in refactoring is different in that it operates on the structure of the program, therefore is often *horizontal* and *non-localised*.

While there are already a few refactoring tools available for a variety of languages such as Java and C#, many of them are commercial products, and there is not much literature (to our knowledge) about the implementation framework of refactoring tools except that for the Refactoring Browser [93, 94], and most recently, for C/C++ source program transformation from the Proteus project [115].

However, implementing a useful refactoring tool for a real world programming

language is by no means a trivial task. A refactoring tool needs to get access to the program's syntax, semantics (possibly including type information), to implement different kinds of program analysis and transformations, and to preserve the layout and comments of the transformed program. Apart from that, there are many criteria for a refactoring tool to be successful, such as efficiency, usability, completeness and so on. To implement such a complex tool to be used in practice, a naïve implementation without making use of proper existing frontends and advanced programming techniques may take years to finish. One aim of this project is to investigate what kind of implementation framework and technique is more suitable for implementing a refactoring tool.

1.4 Contributions of this Research

The study of this thesis was carried out as part of the *Refactoring Functional Programs* project. This study focuses on the implementation of a refactoring tool for Haskell programs and the specification and verification of Haskell refactorings. The main contributions of this research are:

- The study of a set of Haskell refactorings. A collection of Haskell refactorings have been analysed in terms of their side-conditions and transformation rules. Side-conditions, together with the transformation rules, guarantee that a refactoring does not change the program's external behaviour.

- The design and implementation of the *Ha*skell *Re*factorer, HaRe. HaRe is built on top of Programatica [81]'s Haskell frontend and Strafunski [65]'s generic programming technique. It covers the full Haskell 98 programming language, and preserves program appearance. By the third release of HaRe, it supports 23 primitive refactorings and one composite refactoring.

- An approach to program appearance preservation. A novel approach to program *appearance preservation* [49] has been proposed and implemented in HaRe. This approach allows us to preserve the comments and layout of

the refactored program without modifying Programatica's Haskell frontend, on which HaRe is built.

- An API for implementing refactorings or general program transformations in HaRe. Apart from the supported refactorings, HaRe also exposes an API which allows the users to implement their own refactorings or more general program transformations in a compact and high-level way.

- A simple language, λ_M, for the specification and verification of refactorings. A λ-calculus augmented with letrec-expressions and a module system has been defined, and the simple language serves as the vehicle of the specification and verification of refactorings.

- The specification and verification of a couple of representative refactorings.

1.5 Thesis Outline

This thesis is structured as follows:

Chapter 1 (this chapter) introduces the topic of refactoring, and places it in the functional programming paradigm.

Chapter 2 clarifies the basic issues involved in tool-support of refactorings, introduces the Haskell refactoring tool, HaRe, and the list of refactorings we have examined in this research, as well as other candidate refactorings.

Chapter 3 describes the software artefacts on which HaRe is built, and discusses our experiences from making these tools work together.

Chapter 4 presents the design of HaRe, including the basic issues involved in building a refactoring tool, how these issues are addressed in HaRe, and the architecture of HaRe.

Chapter 5 describes the implementation of refactorings through two examples: *rename a variable name* and *from concrete to abstract data type*.

Chapter 6 describes the API provided by HaRe, and how it can be used to implement refactorings in a compact and transparent way.

Chapter 7 presents the simple language used for the specification and verification of refactorings, and examines the specification of two representative refactorings: *generalise a definition*, and *move a definition from one module to another*, as well as the verification that show these two refactorings are behaviour-preserving.

Chapter 8 reviews the related work in both refactoring and closely related areas.

Chapter 9 summarizes our overall conclusions, and presents the possible directions for future work.

Appendix A contains the implementation of the program layout adjustment algorithm, which is used for preserving program layout.

Appendix B lists HaRe's API.

Appendix C contains the data type representing identifiers defined by Programatica [81].

Appendix D contains some generic strategy and recursive traversal combinators from Strafunski [65].

Appendix E contains the implementation of the *renaming a variable* refactoring.

Appendix F contains the implementation of the *from concrete to abstract data type* refactoring.

Chapter 2

A Model of Refactoring

Behaviour preservation and program appearance preservation in the context of refactoring Haskell programs form the basis of this thesis. Behaviour preservation is the fundamental requirement for refactoring. Generally speaking, given the same input values, the main function of the program should produce the same output values before and after a refactoring. However, depending on the application area, more behaviour constraints, such as efficiency, memory consumption, etc, can be added to this requirement. Programming languages vary in their semantics and syntactical rules, therefore expose different contexts for behaviour preservation. For a tool automating the refactoring process, program appearance preservation is also essential for the tool to be accepted in practice, though this is challenging when the concerned programming language does not have a set of widely used standard layout rules.

This chapter aims to set up the context of this study by clarifying the basic issues involved. Apart from the meaning of behaviour preservation and program appearance preservation, this chapter also introduces the Haskell 98 programming language and some of its properties, gives an overview of the tool we have implemented for refactoring Haskell programs, and describes both the refactorings we have examined and the refactorings worth examination in the future.

2.1 The Haskell Programming Language

The Haskell [53] programming language is typical of many modern functional languages. Haskell manifests features such as higher-order functions, lazy evaluation, equations and pattern matching, a type system with Hindley-Milner type inference [42, 75] and type classes [80], monadic programming [116], and a module system [28]. Haskell has evolved continuously since its first publication. The current standard version is Haskell 98, and defined in *Haskell 98 Language and Libraries: the Revised Report* [53].

A Haskell program is a collection of modules. A module defines a collection of values, data types, type synonyms, classes, etc [53]. A Haskell module may import definitions from other modules, and re-exports some of them and its own definitions, making them available to other modules. One of the modules contained in a program must be called $Main$, by convention, and exports the value *main*. The value of the program is the value of the identifier *main* defined at the top level of the *Main* module, which must be a computation of type IO τ for some type τ. When the program is executed, the computation of *main* is performed and its result (of type τ) is discarded [53].

In this research, we examine the application of refactoring techniques to Haskell programs, and also use Haskell as the implementation language to build the tool for refactoring Haskell programs. Choosing Haskell as the implementation language allows us to explore the usability of Haskell as a language for implementing refactoring tools, also allows us to find out how refactoring can help us during the development of a non-trivial Haskell software system. In the rest of this thesis, we assume a basic familiarity with Haskell 98.

2.2 The Meaning of Behaviour Preservation

The most essential criterion for behaviour-preservation is: given the same input value(s), the program should produce the same output before and after (the refactoring). However, this basic criterion might not be sufficient in some application areas, in which case more constraints such as execution time or memory consumption could be added to the definition of behaviour preservation.

Haskell is a general purpose programming language, and most Haskell programs don't have constraints on execution time, memory consumption, etc. Hence it is reasonable to consider only the basic criterion for behaviour preservation when refactoring general Haskell programs. That is, given the same input values, the top-level identifier *main* defined in the *Main* module of the program under refactoring should produce the same output before and after the refactoring. The semantics of other functions defined in the program could be changed after a refactoring, as long as the change does not affect the value of *main*.

2.3 Some Haskell 98 Program Properties

Given a syntactically correct program, the program after a refactoring must still be syntactically correct, and shows the same observable behaviour as the original program does. Systematically, a refactoring has three different aspects: a set of *side-conditions* that should be met by the program under refactoring in order for the refactoring to preserve behaviour; a set of *transformation rules* which specify how to transform the program in a disciplined way; and a *proof* showing that the transformation preserves the program's behaviour given that the side-conditions are satisfied.

While refactorings differ in their side-conditions and transformation rules, there is a set of Haskell 98 program properties that should be taken into account by the specification of each refactoring. These properties could easily be

violated by inadequate side-condition checking or improper program transformations. Violating any of these properties could produce a program that either fails to compile or compiles but is semantically different from the original program. These properties include:

- *Distinct entity names.* The entity names declared in the same scope and name space must be distinct from each other, otherwise a *name conflict* error will be incurred. Nevertheless, the same name can be declared in inner or outer scopes.

- *Unique binding.* At each use-site of an identifier, it must be possible to unambiguously resolve which entity is thereby referred to, that is there must be only one binding for the identifier. This use-site could be in either the body or the export list of a module. An *undefined identifier* error will be incurred if no definition is bound to the identifier, and an *ambiguous reference* error would be given if more than one definition is bound to this identifier. *Ambiguous reference* could only happen to top-level identifiers, and can be avoided using *qualified names.* In Haskell 98, a qualified name is a name prefixed with a qualifier (a module name or a module name alias), which is used to resolve conflicts between entities defined in different modules but with the same name.

- *No name clashes in the export list.* The unqualified names of the entities exported by a module must all be distinct to avoid name clashes.

- *No unexported entities in import declarations.* The entities explicitly stated in an import declaration that imports a module, say M, must be exported by M.

- *Compatible type signature.* After a refactoring, the type signature should be compatible with the type inferred by the compilers for the related entity.

- *No name capture. Name capture* must not happen during the refactoring process. *Name capture* occurs when an identifier that should be free in a

scope becomes bound because of the declaration of the same name across nested scopes. For instance, in the example:

$$g\ y\ =\ f\ y\ +\ 17\ \ \textbf{where}\ h\ z\ =\ z\ +\ 34$$

renaming h to f will capture the use of the free variable f in the definition of g. Unlike the above listed properties, name capture is an error which could not, in general, be detected by the compiler, and can only be avoided by proper side-condition checking and transformation rules.

2.4 Program Appearance Preservation

Like manual refactoring, a tool that automates the refactoring process also needs to address the problem of program appearance preservation. By program appearance preservation, we mean that the refactored program should preserve the original program's layout and comment information as much as possible. Program layouts reflect programming habits, which are normally different from person to person, especially when a standard layout is not enforced by the program editor. Comment information is valuable for program understanding and long-term maintenance, hence should not be discarded by the refactorer.

A similar opinion regarding to program appearance preservation was also pointed out by J.R. Cordy in a keynote paper [23], where he stated: *"... Recognizability of the source therefore becomes an important issue. Even if our automated maintenance systems do a wonderful job of renovating or updating an application, if the source code comes back reformatted, even just by changing the indentation or comment placement, the recognizability and hence the deep understanding is disturbed. It just doesn't look like their old friend any more, and they want their old code back."*.

Program appearance preservation is a challenging task given the existing programming language processing frameworks. A refactoring tool normally carries

out program analysis and transformation on some internal representation of programs, such as the abstract syntax tree (AST), the control flow graph (CFG) [5], the program dependency graph (PDG) [97], etc. Naturally, the refactoring tool needs to reproduce the program source from the internal representation after the refactoring process. However, most programming language processors discard layout or comment information or even both during the transformation from program source to the internal representation, and most pretty-printing tools for producing program source from ASTs just pretty-print the layout and completely ignore the original one. Together, this makes program appearance preservation a hard task.

A novel approach to program appearance preservation has been proposed in our implementation of the Haskell refactorer. In this approach, we use both the AST and the token stream as the internal representations of the source code. Both layout and comment information are kept in the token stream, and only some layout information is kept in the AST. After a refactoring, instead of pretty-printing the AST, we extract the source code from the transformed token stream. The detailed description of this approach is presented in Chapter 4.

2.5 HaRe: The *Ha*skell *Re*factorer

HaRe is the tool we have built to support refactoring Haskell programs. It covers the full Haskell 98 standard language, and is integrated with the two most widely used Haskell development environments (according to our survey [2]): Vim [3] and (X)Emacs [1, 4]. Apart from preserving behaviour, HaRe aims to preserve both the comments and layout of the refactored programs as much as possible. HaRe is implemented in Haskell using the Programatica [81] frontend for Haskell and the Strafunski library [65, 67, 66] for generic AST traversals and transformations. The first version of HaRe, containing a collection of scope-related single-module refactorings, was released in October 2003; multiple-module versions of these refactorings were added in HaRe 0.2, released in January 2004; various data-oriented refactorings were added in HaRe 0.3, released in November 2004, and this version

also restructures HaRe to expose an API for implementing refactorings and more general transformations of Haskell programs using HaRe's framework. HaRe operates on a project, which is a collection of files containing Haskell modules that are closed under the import relations between them. A multi-module refactoring could potentially affect every module contained in a project, but has no effect beyond the project.

2.6 The Refactorings in HaRe

This section describes the refactorings we have examined and implemented in HaRe. These refactorings fall into three categories: structural refactorings, module refactorings and data-oriented refactorings. The refactorings implemented in HaRe is only a subset of the still evolving refactoring catalogue [91] maintained by Simon Thompson. Due to time limit, we could not implement all the refactorings listed in the catalogue. Nevertheless, we chose to implement those refactorings which are basic, but useful, and most importantly can give us insight into the basic problems involved in setting up the framework of implementing a refactoring tool. With a properly established framework, implementing more refactorings is just a time issue. What follows is a brief description of each refactoring and its side-conditions.

2.6.1 Structural Refactorings

These refactorings mainly concern the name and scope of the entities defined in a program and the structure of definitions.

Rename a variable, type variable, data constructor, type constructor or a type class name, and update all the references to it. Renaming is possibly the most basic, but very useful, refactoring. It allows the name of an identifier yo keep reflecting the identifier's meaning. Suppose the old and new names are *bar* and *foo* respectively, then the side-conditions on *renaming* are:

- No binding for *foo* may exist in the same scope. This condition avoids *name conflict* in the scope where *bar* is defined.
- No binding for *foo* may intervene between the binding of *bar* and any of its uses, and the binding to be renamed must not intervene between existing bindings and the uses of *foo*. This condition avoids *name capture*.
- If the *bar* to be renamed is a top-level identifier, then *foo* (either qualified or unqualified) should not be exported by a module by which the *bar* to be renamed is also exported. This condition avoids *name clash* in the export list.

The possible *ambiguous reference* problems caused by *renaming* can be avoided using qualified names.

Delete a definition. The definition must not be used, or explicitly exported, by the module that contains it. If the definition is implicitly exported, it should not be used by any module that imports it. We say an entity is explicitly exported by a module if it occurs in the export list of the module; and an entity is implicitly exported by a module if it is exported by the module, but does not occur in the module's export list. This refactoring helps to clean up the program.

Duplicate a function under a user-provided new name. For conditions on the new name see the *renaming* refactoring. This refactoring is usually used as a precursor to making a modified version of the duplicated definition.

Promote a definition from a local scope to a wider scope, or directly to the top level of the module, say M. Widening the scope of an identifier allows the identifier to be used by a wider range of entities in the program. Lifting a definition to the top level of a program also allows easier testing of the definition's functionality. Suppose the name bound by the definition to be promoted is *foo*, then the side-conditions on *promoting* are:

- Promoting the definition must not intervene between any binding of *foo* and its uses in the outer scope.

- No binding for *foo* may exist in the binding group to which the definition is promoted.

- Free variables that are used by the definition but unbound at the outer scope are converted to parameters of the lifted definition. This requires: a) the free variables must not be used polymorphically if it is defined by a pattern binding, and b) the definition *foo* must be a function definition or a *simple pattern binding* (i.e. a pattern binding is which the pattern consists of only a single variable), rather than a *complex pattern binding* (i.e. a pattern binding which is not simple), so that parameters can be added to the definition if necessary.

- If *foo* is exported by the module containing the definition to be promoted (this *foo* could be defined by some other module, and imported by M), then the definition can not be promoted to the top level of the module if it will be exported by the module automatically.

Demote a definition which is only used within one definition to be local to that definition. This refactoring helps to group the related definitions together and to clean up the program. In contrast to *promoting a definition*, *demoting a definition* tries to remove parameters using free variables if possible. The side-conditions on demoting a definition are:

- All uses of the definition must be within the inner scope to which the definition is moved.

- Variables used by the definition must not be captured when the definition is moved over nested scopes.

- The definition can not be demoted if it is explicitly exported by the module containing it.

Unfold a definition by replacing an identified occurrence of the left-hand side(LHS) of a definition with the instantiated right-hand side (RHS). This refactoring helps to reveal the actual definition of the unfolded identifier. The unfolded definition should be a function definition or a simple pattern binding. Unfolding a complex pattern binding might involve extracting the part of the definition related to a particular identifier defined by the pattern binding, and this is not supported by the current implementation of HaRe. Unfolding a definition with guards and/or multiple equations requires the guards to be removed and the multiple equations to be transformed into a single-equation at the side of unfolding. In the implementation of HaRe, we use conditional expressions to remove guards, and use *case* expressions to transform a multi-equation definition into a single-equation definition. The side-conditions on *unfolding* are:

- The bindings for the free variables of the RHS of the unfolded definition must be accessible at the site of unfolding.
- The free variables (parameters) at the site of unfolding should not be captured by the bound variables of the instantiated RHS of the unfolded definition.

Introduce a new definition to name a user-identified expression. The introduced definition is added as a local definition of the definition whose RHS sub-expression has been identified. This can be followed by a *promoting* refactoring to take the definition to the top-level, so that the definition can be used by other definitions in the program. The conditions on the new definition name are:

- No binding for the new name may exist in the binding group where the new definition is added to.
- The new binding should not intervene between the existing bindings of the new name and any of its uses.

Generalise a definition by making an identified expression of its right-hand side into a value passed into the function via a new formal parameter, therefore

improve the usability of this definition. The new parameter is added as the first parameter of the generalised definition, so that we do not need to worry about partial applications. When the generalised definition has multiple equations, the new parameter is added to all the equations. The type signature (if there is any) of the generalised definition needs to be amended to reflect the change to parameters. The side-conditions on *generalisation* are:

- The new formal parameter must not conflict with the definition's existing formal parameters. i.e. the new parameter name should not be the same as any of the definition's existing formal parameter names.
- The new formal parameter must not capture any existing uses of free variables.
- After *generalisation*, the identified expression (or alternatively, a newly introduced identifier denoting this expression) becomes the first argument of the generalised function at every call-site of it. For each new occurrence of this expression, it is required that the bindings of all the free variables within the expression are resolved in the same way as they are in the original occurrence.

Add an argument to a function definition/simple pattern binding as its first argument. *undefined* is added to the call-sites of the definition as its default first argument. This refactoring is normally followed by future manipulations to the definition to make use of the newly added argument. The side-conditions on the new argument name are:

- The new name should not be the same as any of existing arguments of the function.
- The new name should not be used as a free variable in the function definition/simple pattern binding.

Remove an argument from a function definition. This refactoring also helps to clean up the definition. The only condition is that the argument is not used

by the definition. However removing the last parameter of a function should not trigger the monomorphic restriction [53] of Haskell 98.

2.6.2 Module Refactorings

Module refactorings concern the imports and exports of individual modules, and the relocation of definitions among modules.

Clean an import list to remove redundant import declarations and entities. This refactoring does not have side-conditions. Corresponding to this refactoring is *clean an export list*, which removes the unused entities from the export list. The latter is not supported by the current release of HaRe, but could be implemented without difficulty.

Add to an import declaration an explicit list of all the imported entities which are actually used in the module containing the declaration. This refactoring is useful when only a few of the entities brought into scope by the import declaration are actually used by the module. Corresponding to this refactoring is *adding an explicit list to a module*, which is another candidate refactoring for HaRe.

Add an entity to the export list of a module. It is required that the entity is in scope at the top-level of the module, and the same entity name is not already exported by this module.

Remove an entity from the export list. This refactoring requires that the entity is not used by other modules in the project. The use-site could be in either the import/export list of a module or the body of the module.

Move a definition from one module, A say, to a user-specified target module, B say. Relocating definitions between modules allows the user to put the

closely-related definitions together in a single module, hence to improve the module structure of the program regarding to import relations. The side-conditions on this refactoring are:

- No binding for the same name may exist at the top-level of module B.
- The free variables used in the definition to be moved should be in scope in module B, and refer to the same bindings as they do in module A.
- If the name defined by this definition is used as a free variable in module B, it should refer to this definition.
- Moving the definition should not introduce mutually-recursive modules. That is, the definition should not be used by any module which is imported by module B either directly or indirectly. We try to avoid introducing mutually-recursive modules during refactoring due to the fact that transparent compilation of mutually recursive modules are not yet supported by the current working Haskell compilers/interpreters.

A variety of this refactoring is moving a group of definitions from one module to another. This refactoring is useful when the user-identified definition depends on some other definitions in the same module, and these definitions could be moved together.

2.6.3 Data-oriented Refactorings

Data-oriented refactorings are associated with data type definitions. One large-scale, data-oriented refactoring implemented in HaRe is the ***from concrete to abstract data type*** refactoring, which turns a user-specified concrete data type into an abstract data type (ADT). A concrete data type exposes the representation of the data type, and allows the user to get access to the data constructors defined in the data type, as shown in the example in Figure 1; whereas an abstract data type [43] hides the data constructors from the users. Making a data type abstract allows changing the representation of the data type without affecting the client

functions that use this data type. ***From concrete to abstract data type*** is a composite refactoring built from a number of elementary ones in a sequential way. What follows is a description of these elementary refactorings, accompanied with illustrations showing how the data type *Tree* defined in Figure 1 can be refactored to an abstract data type.

```
  — Tree.hs
module Tree where

data Tree a
  = Leaf a
  | Node a (Tree a) (Tree a)

  —  Main.hs
module Main where
import Tree

flatten :: Tree a → [a]
flatten (Leaf x ) = [x]
flatten (Node x l r ) = x : (flatten l ++ flatten r)

main = print $ flatten (Node 1 (Leaf 2) (Leaf 3))
```

Figure 1: *Tree* as a concrete data type.

Add field names to the data type if field names do not exist. These field names can be used as *selector functions* to extract a component from a structure. Applying this refactoring to the data type *Tree* in HaRe will turn its definition into:

```
data Tree a
  = Leaf {leaf1 :: a}
  | Node {node1 :: a, node2 :: Tree a, node3 :: Tree a}
```

The fresh new names are chosen by the refactorer automatically, however the user

can *rename* them. This also applies to the following two refactorings. This refactoring assumes that the datatype under refactoring is not an existential datatype (a Haskell extension).

Add a discriminator function for each data constructor defined in the data type, if a discriminator for this data constructor does not exist in the current module. A discriminator function indicates whether or not a value is constructed using a particular constructor. Applying this refactoring to the data type *Tree* will add the following discriminator functions to module *Tree*:

```
isLeaf :: Tree a → Bool
isLeaf (Leaf _) = True
isLeaf _ = False

isNode :: Tree a → Bool
isNode (Node _ _ _) = True
isNode _ = False
```

Add a constructor function for each data constructor defined in the data type if such a constructor function does not exist in the current module. A constructor function for a data constructor builds a data structure from its components, and has the same signature as the constructor. Applying this refactoring to the data type *Tree* will add the following functions to module *Tree*:

```
mkLeaf :: a → Tree a
mkLeaf = Leaf

mkNode :: a → Tree a → Tree a → Tree a
mkNode = Node
```

Remove nested patterns. A nested pattern is a pattern containing constructors from other data types. Consider the following example, which uses the data constructor *Leaf*:

$$f\ (Leaf\ [x])\ =\ x\ +\ 17$$

In order to make *Tree* an abstract data type, we need to replace the pattern *Leaf* $[x]$ by a variable. However, *Leaf* $[x]$ is a nested pattern as the list constructor is used within it. Before removing the *Leaf* pattern, we need to remove the *List* pattern first, otherwise we will lose access to the variable x. Note that only patterns that are nested inside the data type we are trying to make abstract will be removed, and others will not be affected by this refactoring. The *List* pattern can be removed by using guards and *case* expressions. This refactoring does not have a side-condition. By removing nested patterns, the above function will become:

$$\begin{array}{l} f\ (Leaf\ p) \\ \quad |\ \textbf{case}\ p\ \textbf{of} \\ \qquad [x]\ \rightarrow\ True \\ \qquad \_\ \rightarrow\ False\ =\ (\backslash [x] -> \ x\ +\ 17)\ p \end{array}$$

The guard is necessary in order for the computation to "fall-through" to the next equation (if there is one) in case that this pattern matching fails.

Eliminate the explicit uses of data constructors. This refactoring eliminates the explicit uses of data constructors declared in the data type throughout the system except in the discriminator/constructor functions. The refactoring requires the following conditions:

- Discriminator, constructor and selector functions exist for each involved data constructor.
- The patterns using the involved data constructors are not nested patterns in the sense above.

Back to the example shown in Figure 1, the definitions of *flatten* and *main* will be affected by applying this refactoring to the data type *Tree*, and after the refactoring, they become:

$$
\begin{aligned}
&\mathit{flatten} :: \mathit{Tree}\ a \rightarrow [a] \\
&\mathit{flatten}\ p \\
&\quad |\mathit{isLeaf}\ p = [\mathit{leaf}1\ p] \\
&\mathit{flatten}\ p \\
&\quad |\mathit{isNode}\ p = (\mathit{node}1\ p) : (\mathit{flatten}\ (\mathit{node}2\ p) +\!\!+ \mathit{flatten}\ (\mathit{node}3\ p)) \\
&\mathit{main} = \mathit{print}\ \$\ \mathit{flatten}\ (\mathit{mkNode}\ 1\ (\mathit{mkLeaf}\ 2)\ (\mathit{mkLeaf}\ 3))
\end{aligned}
$$

Create an ADT interface. This refactoring creates the module interface so that the definition of the specified data type is invisible from outside the module, whereas other definitions which should be visible to other modules are exported. This refactoring requires that the data constructors declared in the data type are not used by other modules on either the right-hand side or left-hand side of any definitions. This refactoring does not move any definition out of the ADT module, however, if necessary, moving a definition out of the ADT module can be achieved using the *move a definition from one module to another* refactoring. The program shown in Figure 1 becomes what is shown in Figure 2 after applying the above sequence of elementary refactorings to the data type *Tree*.

```
  — Tree.hs
module Tree(Tree, isLeaf, isNode, leaf1, mkLeaf, mkNode, node1, node2, node3) where

data Tree a
  = Leaf {leaf1 :: a}
  | Node {node1 :: a, node2 :: Tree a, node3 :: Tree a}
mkLeaf :: a → Tree a
mkLeaf = Leaf

mkNode :: a → Tree a → Tree a → Tree a
mkNode = Node

isLeaf :: Tree a → Bool
isLeaf (Leaf _) = True
isLeaf _ = False

isNode :: Tree a → Bool
isNode (Node _ _ _) = True
isNode _ = False

  — Main.hs
module Main where
import Tree

flatten :: Tree a → [a]
flatten p
  |isLeaf p = [leaf1 p]
flatten p
  |isNode p = (node1 p) : (flatten (node2 p) ++ flatten (node3 p))

main = print $ flatten (mkNode 1 (mkLeaf 2) (mkLeaf 3))
```

Figure 2: *Tree* as an abstract data type.

2.7 Some Refactorings Which are not in HaRe

Apart from the refactorings we have examined implemented in HaRe, there are some other common Haskell refactorings we would like to add to HaRe. A more extensive catalogue of these refactorings, maintained by Simon Thompson, is available from our project website [91]. Some representative refactorings from this

catalogue are:

2.7.1 Structural Refactorings

- ***Eliminating duplicated code*** by extracting a common function. Duplicated code arises naturally during the development of software systems for a number of reasons such as copy and paste, however duplicated code have a negative effect on the maintenance of such software systems. This refactoring involves both detecting and removing the duplicated code. While eliminating those exactly duplicated code might be relatively straightforward, eliminating those code which is not exactly the same but very similar is more challenging. Chris Brown from the HaRe group has been looking into this refactoring.

- ***Swapping*** the position of two arguments of a function. A simplified version of swapping the first two arguments of a function has been implemented as an example illustrating the uses of HaRe API (see Chapter 6). A complete implementation of this refactoring needs to take partial application and type information into account.

- ***Converting*** between curried and uncurried arguments. A function which takes its arguments one at a time is said to be a curried function, and a function which takes all its arguments together as a tuple is said to an uncurried function. From curried functions to uncurried functions needs to handle partial application.

- ***Converting*** between *let* expressions and *where* clauses, which could potentially narrow/widen the scope of those involved bindings.

- ***Introducing pattern matching over an argument position*** by replacing the variable argument at this position with an exhaustive set of patterns over the type of the variable. This refactoring is suitable when

the right-hand side of the definition is a *case* expression switching over the variable.

- ***Replacing a multiple-equation definition*** with a single-equation definition using *case* expressions. This transformation has been used in the implementation of *unfolding a definition*, but is worth of being a separate refactoring.

- ***Splitting a function*** doing two things into two separate ones. Separating the loosely-related functionalities in one definition makes the function easier to understand and reuse. One special case of this refactoring is to split a function returning a tuple into two functions returning each part of the tuple separately. This refactoring can be a composite refactoring, and some elementary refactorings such as *introduce a definition*, *promoting a definition*, *remove an argument* can be the building blocks for this refactoring. Program slicing techniques [117, 118] could also help the implementation of this refactoring.

2.7.2 Module Refactorings

Apart from the refactorings mentioned in Section 2.6.2, some other useful module refactorings include:

- ***Splitting a single module*** into two so as to seperate loosely related definitions into different modules.

- ***Combining a number of modules*** into one module to group together closely related definitions. Both this refactoring and the above refactoring need to amend the import and export lists of the affected modules.

2.7.3 Data-Oriented Refactorings

- ***Naming a type using*** `type`. Proper type synonyms make a program easier to understand. Uses of a type should be identified and made instances of

the type synonym.

- ***Naming a type using*** `data` ***or*** `newtype`. This changes the meaning of the data type as new data values have to be introduced (an extra $\perp$ is also added to the value domain when `data` is used), and hence might change the meaning of the program, therefore could be called "not quite a refactoring".

- ***Introducing*** `class` ***and*** `instance` by identifying a type and a collection of functions over that type.

- ***Monadification***. This refactoring turns non-monadic programs into monadic form. For example, a non-monadic program can be 'sequentialized' to make it monadic, or alternatively, a program with explicit actions, but without explicit uses of monads, can be turned into a program which explicitly uses the monadic operations. Existing work on describing modification includes M. Erwig and D. Ren's *monadification* [33] and R. Lämmel's *monad introduction* [61].

A suite of basic operators for datatype transformation, including *permutation of type parameters and constructor components*, *introduction and elimination of type declarations*, *folding and unfolding of type declarations*, etc, has been studied and implemented in Haskell by J. Kort and R. Lämmel [60]. Integration of these datatype transformations into HaRe would be desirable.

2.8 The Design Space Problem

During the specification and implementation of refactorings, quite often we are in a situation where there is a choice of the specification/implementation, and need to decide which solution we should choose. There are basically three kinds of scenarios where we might need to make a design decision:

- Interpreting the definition of a refactoring. Given the name of the refactoring, there may be more than one interpretation. For example, by *generalising a definition*, one could mean to generalise either the definition whose right-hand side directly contains the identified expression, or the outmost definition containing the identified expression, as shown in the following function definition:

  $$sumSqs\ x\ y\ =\ x\ \hat{}\ pow\ +\ y\ \hat{}\ pow\quad \textbf{where}\ pow\ =\ 2$$

  where by identifying the expression 2, we could generalise either the function *pow* or the function *sumSqs*.

  Another example is the *introduce a new definition* refactoring, in which case one could mean to replace *only* the highlighted expression, or *all* or *some* of the occurrences of the identified expression within the module by the instantiation of the newly-created definition.

- Balancing between side-conditions and transformation rules. In general, the combination of side-conditions and transformation rules guarantees the behaviour preservation of refactorings. However, it is not always clear where a line should be drawn between the side-conditions and transformation rules. On one hand, stronger side-conditions could simplify the transformation rules, but may allow fewer refactoring opportunities or increase the number of necessary refactoring steps to achieve a target state of the program. On the other hand, weaker side-conditions could complicate the transformation rules, but may allow more refactoring opportunities or reduce the number of necessary refactoring steps to get to a target program state. For instance, the *move a definition from one module to another* refactoring moves an identified definition from its current module to a user-specified target module. One situation comes along with this refactoring is that some variables that are free in the definition to be moved are not in scope in the target module, as shown in the following simple example:

— M1.hs

```
module M1 where
sq x = x ^ pow where pow = 2
foo x y = sq x + sq y
```

— Main.hs

```
module Main where
import M1
main x y = print $ foo x y
```

where *sq* is a free variable used in the definition of *foo*, and moving the definition of *foo* to module *Main* requires *sq* to be in-scope in module *Main*. In this case, one solution is to invalidate the refactoring by requiring, as part of the side-conditions, that all these free variables must be in-scope in the target module; the other solution is to proceed with the refactoring by bringing those free variables into scope during the transformation phase. The second solution is more powerful, but involves more complex program analysis and transformation.

- **Implementation considerations**. Even if the side-conditions have been fixed, there are still decisions needed to be made during the program transformation phase. Again with the *move a definition from one module to another* refactoring, suppose the definition to be moved is exported by its current module, then we need to decide, in the refactored program, whether the current module should still export the definition, and which module (the current module or the target module) should a client module import the definition from. While behaviour preservation can be guaranteed in both case, the transformation rules regarding to the imports/exports of involved modules will be different.

The combination of different design decisions at different stages could produce a number of variants under the same refactoring name, and it is not possible

to implement all of them in practice. Therefore we try to implement the most reasonable decision from both the user's point of view and the implementation considerations. One of our design principles is to make the definition and implementation of a single refactoring as clean and simple as possible. We try to avoid doing too many things in one single refactoring by decomposing a complex refactoring into simple ones. This allows the implementation of the basic refactorings easier to understand, maintain and reuse, and also allows the user to have better control over the refactoring process and to compose the basic refactoring steps in new ways.

Chapter 3

Technology Background

Implementing a tool for refactoring programs written in a non-trivial programming language, Haskell, needs tool support itself. Briefly, a complete Haskell-in-Haskell (we are using Haskell to write the Haskell refactorer) frontend, including a lexer, a parser, a type-checker, a module analysis system and a pretty-printer, is indispensable. Haskell generic programming techniques, especially for abstract syntax tree (AST) traversals, which can significantly reduce the implementation time and the amount of code are highly desirable, as both program analysis and transformation involve huge amount of AST traversals. As an interactive program manipulation tool, it is important for the tool to be integrated with the mostly commonly used Haskell editors/IDEs, so that it is easily accessible, and people do not need to give up their favourite development environments in order to use the refactoring tool.

Instead of developing these supporting tools from scratch, we tried to make use of the existing Haskell frontends and generic programming techniques, so that we could concentrate on the most important part of implementing a refactoring tool, i.e. analysing the inherent logic of refactorings. One problem with reusing the existing Haskell frontends is that some information we have expected to be available may not be provided by the tool, or is provided in a way different from what we have expected. This, to some extent, affects the design of the refactoring tool as explained in chapter 4.

This chapter describes the software artefacts on which our HaRe is built, and discusses our experiences from making these tools work together.

3.1 Introduction

Like text editors, refactoring tools support interactive program manipulation; but unlike text editors, refactoring tools operate on the level of program syntax and semantics instead of character strings.

Syntactically, a Haskell refactoring tool needs to get access to the different kinds of syntax phrases in a program, such as declarations, expressions, identifiers, etc. This can be achieved by using the AST representation of programs. ASTs, typically generated by a parser, capture the essential structure of the program by using a collection of mutually recursive data types, while omitting the unnecessary syntax details, such as brackets, keywords, etc. Some examples will be given in Section 3.2.

Semantically, a Haskell refactoring tool needs to have certain kinds of static semantics information, including scope information, type information and module information, of the program under refactoring for the purpose of side-condition checking and program transformation. Scope information reflects the name space of identifiers and the binding structure of the program (binding structure refers to the association of uses of identifiers with their definitions in a program, and is determined by the scope of the identifiers); type information tells the type and kind information of identifiers; and module information reveals the module graph of the program and the interfaces of individual modules contained in the program. Static semantic analysis, type analysis and module system analysis are needed to obtain this information.

Comments and layout information are essential for a refactorer to preserve program appearance. Hence, it wound be ideal if this information is kept in the AST, and the pretty-printer, which produces program sources from ASTs, could make use of it during the pretty-printing process. Unfortunately, most parsers or

lexers discard this information, and almost all pretty-printers ignore it even if the information is kept in the AST. Therefore, efforts have had to be made in order to preserve program appearance while still using the existing Haskell frontends.

After having examined and compared the Haskell frontends provided by GHC [38], Haddock [69], Hatchet [70], and Programatica [81], we decided to use Programatica's frontend for our Haskell refactorer based on the fact that it best supported the full Haskell 98 and provides the most complete information compared with the other systems available. This was done at the project start, i.e. autumn/winter 2002.

Although, in theory, program analysis and transformation functionalities can be written over ASTs without further tool support, the programmer will soon realise the huge amount of boilerplate code he, or she, has to write for AST traversals. This is due to the large size of Haskell 98 abstract syntax, whose representation normally contains a large number of mutual recursive algebraic data types, each being a sum of a large number of data constructors. While some generalised higher order functions, such as *map*, *fold*, can be written in plain Haskell, it is still not convenient to program over the complex, recursive, nested abstract syntax of Haskell. To attack this problem, a generic programming technique which allows a high-level, succinct specification of program analyses and transformations is needed.

At the time we started our refactoring project, the Strafunski [65, 67, 66] tool had just begun to be stabilised. Strafunski is a Haskell-centred software bundle developed for supporting generic programming in application areas that involve term traversals over large abstract syntaxes. After having experimented with Strafunski on some program analysis/transformation examples, we felt that Strafunski is the right tool for our AST traversal purposes. Our later experience also showed that using Strafunski was a correct decision. In late 2003, another generic programming approach, i.e. the *scrap your boilerplate* approach[63, 64], emerged. This approach is also a lightweight generic programming approach focusing on

term traversal as the prime idiom of generic programming. Originated from Strafunski, the *scrap your boilerplate* approach is now fully supported by GHC 6.4, and provides another choice of generic programming (in Haskell) support for AST traversals.

Finally, as the refactoring tool supports interactive program manipulation, it has to be integrated with Haskell development environments. Choosing the proper development environments is also crucial for the refactorer to be accepted in practice, as people may be unwilling to use the refactoring tool if it is not supported by their favourite Haskell development environment(s). For this purpose, we launched a Haskell editing survey[2] in July 2002, which showed that Vim [3] and (X)Emacs [1, 4] cover the vast majority of Haskell programmers' development environments. Hence, we decided to choose Vim and (X)Emacs as the environments to host our Haskell refactoring tool.

3.2 Programatica's Haskell Frontend

Programatica [81] is a project carried out at the OGI School of Science & Engineering, Oregon Health & Science University. Programatica is a system implemented in Haskell for the development of high-assurance software in Haskell. It supports an interactive development environment in which the program, its properties, and evidence are simultaneously developed and improved. To this purpose, the Programatica team have developed a very expressive logic, called P-logic, and they have extended Haskell to support property definitions and assertions in P-logic. Source code written in Programatica's extended Haskell can include both definitions of executable code and assertions of properties, and an assertion of properties can be accompanied by a certificate which encapsulates the evidence for this assertion.

Programatica supports the full Haskell 98 standard language and a number of Haskell extensions to varying degree. Components of Programatica's frontend include a lexer, a parser, a type-checking system, a module analysis system, and a pretty printer. As all these components contribute to the implementation of

HaRe, a brief description of each part of the frontend is given in what follows.

3.2.1 The Lexer

Instead of being hand-written in Haskell, the main part of Programatica's lexer [41], the token recognition, is generated from the lexical syntax specification in the Haskell report [53] using a regular expression compiler. The lexer takes program source as input and produces a list of tokens, which is called the token stream. The whole lexer is split up into several passes, and the first passes of the lexer preserve both comments and white spaces (' ', '\t', '\n', '\f' and '\r') are also tokens in the token stream, and consecutive whitespace characters are put in one single token. The type of the lexer is defined as:

$$\textbf{type}\ \mathit{Lexer}\ =\ \mathit{String}\ \rightarrow\ [(\mathit{Token}, (\mathit{Pos}, \mathit{String}))]$$

where the first *String* represents the program source, *Token* is a data type classifying different kinds of tokens, *Pos* represents the token's position in the source in terms of row and column numbers, and the second *String* contains the content of the token.

3.2.2 The Abstract Syntax

Programatica represents the ordinary Haskell abstract syntax [53] with a parameterised syntax. The definition of parameterised syntax is split into two levels: a structure defining level, and a recursive knot-tying level [100]. For example, the data type defining the structure of an expression is as shown in Figure 3 , where the parameter i represents the type of identifiers, e represents the type of expressions, p represents the type of patterns, ds represents the type of declarations, t represents the type of types and c represents the type of type context. The definition of these parameter types can also be recursive. The definition which ties the recursive knots of the expression type is:

$$\textbf{newtype}\ \mathit{HsExpI}\ i$$

```
data  EI i e p ds t c
        = HsId (HsIdentI i)
        | HsLit HsLiteral
        | HsInfixApp e (HsIdentI i) e
        | HsApp e e
        | HsNegApp SrcLoc e
        | HsLambda [p] e
        | HsLet ds e
        | HsIf e e e
        | HsCase e [HsAlt e p ds]
        | HsDo (HsStmt e p ds)
        | HsTuple [e]
        | HsList [e]
        | HsParen e
        | HsLeftSection e (HsIdentI i)
        | HsRightSection (HsIdentI i) e
        | HsRecConstr i (HsFieldsI i e)
        | HsRecUpdate e (HsFieldsI i e)
        | HsEnumFrom e
        | HsEnumFromTo e e
        | HsEnumFromThen e e
        | HsEnumFromThenTo e e e
        | HsListComp (HsStmt e p ds)
        | HsExpTypeSig SrcLoc e c t
        | HsAsPat i e
        | HsWildCard
        | HsIrrPat e
```

Figure 3: The data type defining the structure of an expression

$$= Exp\ (EI\ i\ (HsExpI\ i)\ (HsPatI\ i)\ [HsDeclI\ i]\ (HsTypeI\ i)\ [HsTypeI\ i])$$

where i represents the type of identifiers, and a new layer of data constructor, *Exp*, has been introduced.

Parameterised syntax provides support for syntax variants and extensions. For example, the type defining a Haskell module is defined as:

```
data HsModuleI m i ds
  = HsModule { hsModSrcLoc :: SrcLoc,
    hsModName :: m,
    hsModExports :: Maybe [HsExportSpecI m i],
```

$$
\begin{aligned}
&\mathit{hsModImports} :: [\mathit{HsImportDeclI}\ m\ i],\\
&\mathit{hsModDecls} :: \mathit{ds}\ \}
\end{aligned}
$$

where the parameter m represents the type of module name, i represents the type of identifiers, and ds represents the type of a declaration list. Different instantiations of these three parameters will result in different abstract syntaxes. The disadvantage of this two-level approach is that it introduces an extra layer of tagging in the data structures.

The Haskell 98 abstract syntax defined by Programatica consists of 20 data types and 110 data constructors in total. Even the data type defining expressions contains 26 data constructors itself as shown in Figure 3. Naïvely writing AST traversals on this non-trivial sized mutually recursive abstract syntax without proper generic programming support would produce huge amount of boilerplate code, and negatively impact the maintenance and reusability of the produced code.

3.2.3 The Parser

Programatica's parser is based on HsParser, a Haskell 98 parser now in the haskell-src package of the Haskell hierarchical libraries, but using Programatica's lexer rather than the lexer in HsParser, and the parameterised abstract syntax. The parser produces a variant of the abstract syntax where every identifier is paired with its actual source location in the file. Source location is represented by the combination of the file name, the actual character position, the row number and the column number. The type for identifiers with source location is *SN HsName*, where *SN* and *HsName* are defined as:

$$
\begin{aligned}
&\textbf{data}\ \mathit{SN}\ i\ =\ \mathit{SN}\ i\ \mathit{SrcLoc}\\
&\textbf{data}\ \mathit{SrcLoc}\ =\ \mathit{SrcLoc}\ \{\ \mathit{srcPath} :: \mathit{FilePath},\\
&\qquad\qquad \mathit{srcChar},\ \mathit{srcLine},\ \mathit{srcColumn} :: !\mathit{Int}\ \}\\
&\textbf{data}\ \mathit{HsName}\ =\ \mathit{Qual}\ \mathit{ModuleName}\ \mathit{String}\\
&\qquad\qquad \mid \mathit{UnQual}\ \mathit{Id}
\end{aligned}
$$

$$\textbf{data}\ \mathit{ModuleName} = \mathit{PlainModule}\ \mathit{String} \mid \mathit{MainModule}\ \mathit{FilePath}$$

A further static scoping process on the AST adds more information to each identifier, and produces another variant of the AST, which is called the scoped AST. The scoped AST representation of a Haskell module is defined as:

$$\textbf{type}\ \mathit{HsModuleP} = \mathit{HsModuleI}\ \mathit{ModuleName}\ \mathit{PNT}\ [\mathit{HsDeclI}\ \mathit{PNT}]$$

In the scoped AST, each identifier is associated with not only its actual source location, but also the location of its defining occurrence and name space information. The type for this kind of identifiers is called PNT and is defined as:

$$\textbf{data}\ \mathit{PNT} = \mathit{PNT}\ (\mathit{PN}\ \mathit{HsName}\ \mathit{Orig})\ (\mathit{IdTy}\ \mathit{Pid})\ \mathit{OptSrcLoc}$$

where

- *HsName* contains the name of the identifier;
- *Orig* specifies the identifier's origin information (which usually contains the identifier's defining module and position);
- *IdTy Pid* specifies the category (i.e. variable, field name, type constructor, data constructor, class name, etc) of the identifier with information of relevant type if the identifier is a field name, type constructor or a data constructor (see Appendix A).
- *OptSrcLoc* contains the identifier's source location information unless the identifier is generated internally by Programatica itself.

The complete definition of *PNT* and its component data types are given in appendix A. Compared with normal ASTs, the scoped AST makes life easier for the user in several aspects:

- Source position information makes the mapping from a fragment of code in the source to its corresponding representation in the scoped AST easier.

- Identifiers can be distinguished by just looking at their *PNT* representations. Two identifiers are semantically the same if and only if they have the same origin.
- Given an identifier, the scoped AST makes it easy to find the identifier's definition and use sites.

The source location information for identifiers in the AST reveals the program's layout to some extent. More information, including comments and their locations, locations for keywords and some special characters, is needed in order to record the complete layout of a program. Unfortunately, this information is not kept in the AST, and we have to get it from the token stream. The scoped AST is the version of AST used in HaRe's implementation.

3.2.4 The Module System

As part of the Programatica project, a formal specification of the Haskell 98 module system has been developed by Iavor S. Diatchki, *et al.* [28]. The specification is written in Haskell, and is executable as a program. In this approach, the semantics of a Haskell program with respect to the module system is a mapping from a collection of modules to their corresponding in-scope and export relations. Given a list of modules, the analysis program either reports a list of errors found in each module, or returns the in-scope and export relations of the respective modules.

3.2.5 The Type Checker

Programatica's type checking system [82] is influenced by a number of earlier pieces of work. Mark P. Jones' *Typing Haskell in Haskell* has provided general guidance, some naming conventions, the class *Types* and operations on substitutions [50]; Johan Nordlander's O'Haskell type checker [77] has suggested ways of avoiding both threading an accumulating substitution and applying substitutions to the environments; HBC's [7] type checker has also been relevant. The type checker is structured to allow large parts to be reusable in extended versions of

the language. The type checking system generates another variant of the abstract syntax tree where all declaration lists have been decorated with the kinds and types of the entities defined in the list, and all applications of polymorphic functions have been decorated with the instantiation of the type variables. The top level of the type checked AST can is defined by:

$$\textbf{type}\ \mathit{TiModule}\ =\ \mathit{HsModuleI}\ \mathit{ModuleName}\ \mathit{PNT}\ (\mathit{TiDecls}\ \mathit{PNT})$$

The advantage of AST annotated with type information is that it allows the user to extract the type information of the entities while traversing the AST. The disadvantage is that the type information makes the AST several times larger than the scoped AST, which has a potential to slow down the traversal and therefore any refactoring using the typed AST. Apart from that, the type inference engine itself is relatively slow.

3.2.6 The Pretty Printer

Programatica's pretty printer is based on the modified version of John Hughes' and Simon Peyton Jones' Pretty Printer Combinators [46, 51]. Instances in the pretty printing class *Printable* have been defined for each data type in the abstract syntax of Haskell. The pretty printer does not use any of the source location information in the AST.

3.3 The Strafunski Library

Strafunski [65, 67, 66] is a Haskell-centred software bundle, developed by R. Lämmel and J. Visser, for supporting generic programming in application areas that involve term traversals over large abstract syntaxes. The key idea underlying the Strafunski style of generic programming is to view traversals as a kind of generic function that can traverse into terms while mixing uniform and type-specific behaviour. Strafunski is based on the notion of *functional strategy*. A

functional strategy is a first-class generic function, which can be applied to arguments of *any* type, can exhibit *type-specific* behaviour, and can perform *generic* traversal to subterms. The advantage of Strafunski is that it allows the user to write concise, type-safe, generic functions for AST traversals, in which only the strictly relevant constructors need to be mentioned. Code written using Strafunski is normally substantially shorter than the code written using plain Haskell, and it is much easier to see that code is *correct by inspection*, as there is no irrelevant information.

Strafunski can be implemented in two different ways. One implementation is based on a specific universal term representation, and supported by a generative tool based on *DrIFT* [72] for the automated derivation of Haskell class instances; the other implementation is based on GHC's support for the *Typeable* and *Data* classes. The classes *Typeable* and *Data* comprise members for type-safe cast and processing constructor applications [64]. To use the second implementation, a clause for deriving the *Typeable* and *Data* class instances needs to be added to the definition of every relevant data type.

The Strafunski library is organised so: *StrategyLib* is the top-level module of the library; under this module is a collection of modules covering a range of generic programming 'themes', such as traversal, name analyses, refactoring, metrics, etc; at the bottom of the library are two modules: the *StrategyPrimitives* module, which defines the strategy types and a suite of basic strategy combinators, and the *Term* module defining a type class *Term* as the generic term interface used by the DrIFT-based approach. A more detailed description of the strategy types, basic combinators, and the commonly used recursive traversal strategies is given next.

3.3.1 Strategy Types and Combinators

Two strategy types are distinguished in Strafunski: ***TP*** for type-preserving strategies and ***TU*** for type-unifying strategies. The result of applying a type-preserving strategy to a term of type t is of type t in a monadic form, whereas the result

of applying a type-unifying strategy is always of a specific type (in a monadic form), regardless of the type of the input term. Monads are used to handle effects such as state, environment, IO, failure, etc. Normally, type-preserving strategies are used for program transformation purposes, and type-unifying strategies are used for program analysis. The exact definition of ***TP*** and ***TU*** depends on the underlying models, and should be treated as abstract data types.

Functional strategies are composed and updated in a combinator style. A suite of basic strategy combinators has been defined in *StrategyLib*. These combinators, as listed in Section B.1 in Appendix B, cover strategy application, strategy update, strategy composition, term traversal, and monad transformation. Strafunski also provides a collection of high-level combinators defined on top of these basic combinators. These high-level combinators are grouped into a number of generic programming 'themes', including a traversal theme, an overloading theme, control and data flow themes, a fixpoint theme, a keyhole theme, a name theme, a path theme, an effect theme, a refactoring theme and a metric theme.

Among the pre-defined themes, the recursive traversal combinators defined in the *traversal theme* are the most heavily used strategies in our refactoring tool implementation. These traversal combinators are listed in Section B.2 of Appendix B, where the definitions of the first and third combinator are also given. Four kinds of recursive traversals have been defined, and they are:

- **Full traversals.** A full traversal visits every node in the term.
- **Traversals with stop conditions.** A traversal with stop conditions cuts off the below nodes where the argument strategy succeeds, but it proceeds the traversal with the sibling nodes.
- **Single hit traversals.** A single hit traversal terminates at the first node where its argument strategy succeeds.
- **Traversals with environment propagation.** A traversal with environment propagation starts the traversal with an initial environment, and modifies the environment during the traversal process.

Some of the traversals, for example full traversals, can proceed in either top-down or bottom-up order, and in this case two versions are defined for each traversal as reflected in the combinator list in Section B.2 in Appendix B.

3.3.2 Some Examples of Using StrategyLib

This section shows how the Strafunski combinators can be used to write generic program analysis/transformation functions over Programatica's abstract syntax by two examples. The first example is a type-unifying function which collects all the data constructors in a fragment of Haskell code; the second example is a type-preserving function which renames all occurrences of a specified identifier to a new name. For comparison, an incomplete implementation of the renaming functionality without using Strafunski is also given, and it is very obvious how the amount of code can be reduced by using Strafunski's combinators.

The first example, shown in Figure 4, collects all the data constructors in a fragment of Haskell code. Here, the functions *applyTU*, *stop_tdTU*, *failTU* and *adhocTU* are type-unifying variants of strategy combinators from *StrategyLib*: *applyTU* applies a type-unifying strategy to a term; *stop_tdTU* traverses the AST in a top-down order, and cuts off below the nodes where its argument strategy succeeds and the result is collected; the polymorphic strategy *failTU* always fails (by using *mzero* from the *Monadplus* class) independent of the incoming term; and *adhocTU* updates a strategy to add a type-specific behaviour so that it behaves just like the function on the left except when the type is such that the function on the right can be applied. We don't use full traversal in this example as terms blow the *PNT* node do not contribute to the result.

When the strategy represented by *strategy* is applied to an AST, it performs a top-town traversal of the AST to the terms of type PNT, where it calls *pntSite*. This latter function returns the data constructor name in a list nested in the *Maybe* monad (*Just* [*pname*]) if the current identifier is a data constructor, otherwise, it returns *Nothing*. The List data type is used to collect together the results when there are several data constructor names. The default strategy *failTU* indicates

```
hsDataConstrs :: (Term t) ⇒ t → Maybe [PName]
hsDataConstrs = applyTU strategy
  where
    strategy = stop_tdTU (failTU 'adhocTU' pntSite)
    pntSite :: PNT → Maybe [PName]
    pntSite (PNT pname (ConstrOf _ _) _)
        = Just [pname]
    pntSite _ = Nothing
```

Figure 4: Collecting data constructors

that this function always fails when faced with terms of any other types than *PNT*. In combination with *stop_tdTU*, this means that only applications of *pntSite* to terms of type *PNT* contribute to the result of applying *strategy* to an AST.

The second example, shown in Figure 5, renames all occurrences of a specified identifier to a new name in an AST. 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 identifier name contained in current subterm (with type *PName*) by *newName* if this identifier is the same as the identifier to rename. Otherwise, it returns the subterm unchanged.

```
rename :: (Term t) ⇒ PName → HsName → t → t
rename oldPName newName = runIdentity.applyTP strategy
  where
    strategy = 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 5: Renaming an identifier

In the above examples, only the strictly relevant data types are mentioned.

```
instance Rename HsExp
  where
    rename oldName newName (Exp (HsId id))
        = Exp (HsId (rename oldName newName id))
    rename oldName newName (Exp (HsLit x))
        = Exp(HsLit x)
    rename oldName newName (Exp (HsInfixApp e1 op e2))
        = Exp (HsInfixApp (rename oldName newName e1)
                          (rename oldName newName op)
                          (rename oldName newName e2))
    rename oldName newName (Exp (HsApp e1 e2))
        = Exp (HsApp (rename oldName newName e1)
                     (rename oldName newName e2))
    rename oldName newName (Exp(HsNegApp e))
        = Exp (HsNegApp (rename oldName newName e))
    rename oldName newName (Exp(HsLambda ps e))
        = Exp (HsLambda (rename oldName newName ps)
                        (rename oldName newName e))
    rename oldName newName (Exp (HsLet ds e))
        = Exp (HsLet (rename oldName newName ds)
                     (rename oldName newName e)
      ...
    rename oldName newName (Exp (HsExpTypeSig Loc e c t))
        = Exp (HsExpTypeSig Loc (rename oldName newName e)
                                (rename oldName newName e)
                                (rename oldName newName t))
```

Figure 6: Rename the occurrences of an identifier in expressions

This liberates us from the complexity of the abstract syntax, and reduces the boilerplate code we need to write. The same renaming functionality has been implemented without using the Strafunski library. Figure 6 shows the class instance renaming the occurrences of a specified identifier in an expression. There would be one clause for each data constructor, therefore 26 clauses for only this class instance. Comparing the implementations in Figure 5 and 6, the advantage of using Strafunski is immediately clear.

3.4 Integrating Strafunski with Programatica

As mentioned in Section 3.3, there are two variants of Strafunski: the DrIFT-based model and the GHC-deriving model. We chose to use the DrIFT-based model for our refactoring project, as it began to stabilise and the other model did not exist at the time we started our project (2002). To use this version, *Term* and *Typeable* class instances need to be derived for Programatica's AST-related data types. This is automated by the instance-deriving tool DrIFT [72]. What we needed to do was to extract those AST-related data types from Programatica into a single file, add our instance deriving commands, and feed the file to DrIFT. This way, we hoped that we needed to make no further modifications to Programatica. Nevertheless, during the automatic deriving process, DrIFT run into various problems with Programatica's more complex data types and efforts were made to solve these problems. For example, initially DrIFT could not cope with Programatica's parameterised two-level types, and several patches were needed to handle it; DrIFT also had a problem with deriving *Term* class instances with data type of the form $data\ T\ a\ =\ D\ [a]$: there was an ambiguous overlap of the String and [a] instances of *Term*, and we had to add $Term\ [a]$ to the instance context so as to delay the decision between the two instances to the point of usage; the unguarded calls to the function *head* made it difficult to trace an error, etc.

Our own refactoring-implementing modules are built on top of Strafunski's traversal library and Programatica's Haskell frontend. One major problem with linking these two libraries together was caused by the Haskell module system's inadequate control over the import/export of class instances. Conflicts occur when a different class instance has been defined by both libraries or their support libraries, and both the modules defining the instances are imported by some other module in the system. Class conflicts had to be solved by renaming data types and/or factoring out common instances during the integration process.

3.5 Some Pitfalls with Using Strafunski

Strafunski allows us to write concise and robust AST traversal functions, but learning how to use Strafunski takes some time, especially for those who are not very familiar with functional programming and/or AST traversals. From our experience, we found there are several points where Strafunski learners can easily get confused. What follows is a summary of our experience, which might be useful for other Strafunski learners.

- Choosing the correct default strategy. It is crucial to choose the proper default strategy for different traversals. For example, *idTP*, which returns the incoming term without change, can be used as the default strategy for *full_tdTP* and *full_buTP*, whereas *failTP* cannot; for type-preserving traversals with stop conditions, such as *stop_tdTP*, *once_tdTP*, and *once_buTP*, *failTP* (which always fails) is the proper default strategy, whereas *idTP* is not; for the type-unifying full traversal *full_tdTU*, *constTU* [], which always returns the empty list, can be used as the default strategy; and for type-unifying traversals with stop conditions, such as *stop_tdTU*, *once_tdTU*, *once_buTU* and *once_peTU*, *failTU* is always the proper default strategy.

- Top-down or bottom-up? For full traversals, choosing top-down or bottom-up normally does not make much difference. However, there is one special case with *full_buTP* and *full_tdTP*. When a full traversal tries to extend a syntax phrase to a larger syntax phrase of the same type like in the example shown in Figure 7, where a single identifier expression is replaced by this expression applied to *undefined*, we found that *full_buTP* can do the job correctly, but *full_tdTP* causes stack overflow because it keeps expanding the AST by adding the expression *undefined*.

- The two-layer monads when using ***TU*** traversals. Strafunski uses monads to manage the backtracking behaviour of its traversal combinators, therefore the traversal is always in a monad form. When using a type-unifying

```
addParam pn = applyTP strategy
  where
    strategy = full_buTP (idTP 'adhocTP' inExp)
    inExp (exp@(Exp (HsId (HsVar (PNT pn' _ _))))) :: HsExpP)
      |pn == pn'
      = return $ Exp (HsParen (Exp (HsApp exp (nameToExp "undefined"))))
    inExp x = return x
```

Figure 7: Add *undefined* as the first argument of the parameter identifier.

traversal for collecting information, one might want to use a list to collect the results. In this case, there are two nested monads involved in the traversal: one for backtracking, and one for collecting data. Consequently, in the returned result, there should be two layers of monads as shown in the example in Figure 4, where *Maybe* monad is used for backtracking by Strafunski, *List* monad is used for collecting data, and the control monad is outside the collecting monad. Omitting one of the monads would lead to wrong results.

Chapter 4

The Design of HaRe

This chapter describes the design of HaRe, including the basic issues involved in building a refactoring tool, how these issues are addressed in HaRe, and the implementation architecture of HaRe. We start from examining the basic steps involved in manually refactoring a trivial Haskell 98 program, then point out the problems we need to address during the refactoring process, and the design of HaRe is explained after that.

As explained in the previous chapter, HaRe is built on top of Programatica's Haskell frontend and Strafunski's traversal library. One of our design principles of HaRe is to try to make good use of the information provided by Programatica, but try to modify the library as little as possible. Programatica is currently a work-in-progress, and modifications have been made to the system by the Programatica team from time to time. Minimising our own modifications makes HaRe's upgrading to new Programatica versions easier. As a result, to some extent, the design of HaRe was influenced by how information is provided by Programatica's frontend, which is especially reflected by HaRe's handling of program appearance preservation.

4.1 An Example Refactoring

We take the *rename an identifier* refactoring as an example. This refactoring renames a specified identifier as a user-supplied new name as shown in Figure 8, where the identifier f defined in module $Test1$ is renamed to sq at both its define-site and use-sites. In Figure 8, the program on the left-hand column is the program before the renaming, and the program after the renaming is shown on the right-hand column.

```
  — Test1.hs
module Test1(f, sumSqs) where

f x = x^ pow
  where pow = 2

sumSqs x y = f x + f y

  — Main.hs
module Main where
import Test1(f)

sq x = x + x

main = print $ sq 10 + f 30
```

```
  — Test1.hs
module Test1(sq, sumSqs) where

sq x = x ^ pow
  where pow = 2

sumSqs x y = sq x + sq y

  — Main.hs
module Main where
import Test1(sq)

sq x = x + x

main = print $ Main.sq 10 + Test1.sq 30
```

Figure 8: Rename the identifier f defined in module *Test1* as sq

To perform this refactoring by hand, the following steps are necessary:

- Locate the definition of identifier f, and infer its scope and name space. In this example, f is a variable defined at the top-level of module $Test1$, and explicitly exported by $Test1$.
- At the define-site of f, check whether the new name, sq, would conflict with any of the existing bindings in the same binding group. The answer is *no* in this case, so we can continue the refactoring. If the answer is *yes*, we would have to abort the refactoring, and ask to choose another new name.

- At the use-sites of f, check whether the new name would cause *name capture* of f, *ambiguous reference* or *name clash* (*name clash* is only possible in the export list). In this example, f is used not only in its defining module *Test*1, but also in the module *Main*, which imports f. A top-level identifier could be referred to in the right-hand side of declarations, in the import declarations or in the export lists. Normally, a *name capture* or *ambiguous reference* involving top-level identifiers can be resolved by using qualified names, whereas a *name clash* error is only resolvable by changing the name. For local variables, none of the above errors can be resolved without changing the name, as local variables cannot be qualified.

  In this example, the new name *sq* does not cause any problem in module *Test*1, but it will cause *ambiguous reference* in the *main* function of module *Main*, since two *sq*s would be in scope in module *Main* after the renaming, and the compiler will not be able to resolve which *sq* refers to which binding. At this stage, we can choose to use qualified names to solve the problem and proceed with the refactoring, or abort the refactoring and ask for another new name. We follow the first option in this example.

- After the previous two steps of checking, it is clear that f can be renamed to *sq* without affecting the program's behaviour. So we rename f to *sq* at both its define-site and use-sites throughout the program, qualify the uses of *sq* in the *main* function in the *Main* module, and finish the refactoring process.

In the above process, the first step locates the focus of the refactoring; the next two steps check whether the transformation will change the behaviour of the program; and the last step carries out the source-to-source program transformation. Apart from renaming the identifier, this step also shows an auxiliary transformation, i.e. qualifying the uses of *sq*, to enable behaviour preservation. All these steps of program analysis and transformation activities are good candidates for automation. Obviously, a tool that automates this refactoring process needs to

automate all these steps. However, unlike human beings who can infer certain semantics by examining the program source and manipulate the program source in a sensible way, a refactoring tool needs to work on some internal representation of the program, such as the abstract syntax tree or the concrete syntax tree, as the string representation of the programs does not provide enough information for a refactoring tool to carry out program analysis and transformation. Consequently, two steps need to be added to the above process: before the refactoring, the source representation needs to be parsed into an internal representation; after the refactoring, the representation needs to be transformed back to program source. Apart from the representation construction, the refactoring tool needs a more formal way to infer module information and type information. Module information is normally necessary when refactoring multi-module programs, and type information is needed by some refactorings for inferring the type of a syntax phrase, such as an expression, and also for keeping the refactored program well-typed. While manual refactoring can preserve the program appearance without being noticed, this problem could become a big issue when the new program source has to be reconstructed from the internal representation.

So far, we can summarise that a refactoring tool needs to address at least the following basic problems:

- transforming program source into some internal representation, and derive the necessary information, such as scope information, module information and type information;
- analysing the program to validate the side-conditions of a refactoring;
- transforming the program according to the transformation rules specified by a refactoring;
- preserving program appearance as much as possible;
- transforming internal representations back to program source, and presenting the refactored program to the user.

The design of HaRe is explained in the next section, with an emphasis on how these problems are solved.

4.2 The Framework of HaRe

4.2.1 Information Gathering

As discussed in Section 3.1 and revealed by the example in the previous section, a Haskell refactoring tool needs the following information to carry out a refactoring:

- The internal representation of programs (we choose to use AST).
- Scope information.
- Type information.
- Module information.
- Layout and comment information.

All these information can be derived from Programatica's frontend in its own particular way. Table 1 summaries the information we need (the left-hand column) and how it is provided by Programatica's frontend (the right-hand column). All

AST	Generated by parser
Scope information	Annotated in the AST after scope analysis
Type information	Annotated in the AST after type checking
Module information	Generated by the module analysis system
Comments	In the token stream generated by the Lexer
Layout information	In the token stream and partially in the AST

Table 1: How information is provided by Programatica's frontend

this information is easily accessible by invoking the relevant frontend functions. The only problem is that the complete layout and comment information is kept in the token stream instead of the AST. This is fairly understandable, as it is much easier to keep layout and comments in the token stream than in the AST, due to the fact that comments can appear everywhere (except inside the identifiers or

literals) in a Haskell program and there is no standard rules for associating comments with the commented code. However, the separation of layout and comment information from the AST imposed a challenge on the design of HaRe, and finally led to a framework where both the AST and token stream are manipulated during the transformation phase, as will be described in Section 4.2.3

4.2.2 Program Analysis and Transformation

Program analysis and transformation form the essence of the refactoring process. Program analysis serves to check whether the side-conditions of a refactoring are satisfied by the program under refactoring, and to collect information that is needed by the program transformation phrase; program transformation transforms the program according to the refactoring's transformation rules. The refactoring process fails if the side-conditions are not satisfied by the program.

For a functional programming language like Haskell 98, the AST annotated with scope information (and type information if needed), together with the module system information, provides enough semantic information for most refactorings. Having scope information in the AST is one of Programatica's advantages over other Haskell frontends. Many refactorings involve names and their manipulation, and it is crucial that these manipulation do not unintentionally disrupt the binding structure of the program. Scope information gives a clear view of the binding structure of the program, and makes the program analysis much easier.

Using Strafunski's AST traversal library, various program analysis and transformation functions can be written over the Programatica-based AST without too much difficulty. Using Strafunski also allows the exposure of the AST representation of programs being minimised, which makes porting HaRe from one Haskell frontend to another easier.

4.2.3 Program Appearance Preservation

If the refactoring tool is designed to demonstrate the variety of ideas (it is a *proof of concept*), then it would be sufficient to pretty-print the transformed AST. However, for a refactoring tool to be used in real world, program appearance preservation is unavoidable, and the above program transformation process has to be adapted to take this problem into account.

With Programatica's frontend, we have comments and layout information in the token stream, and location information for identifiers and literals in the AST. Further examination revealed that given a syntax phrase from the AST, we can roughly locate it in the token stream without much difficulty, as long as the given syntax phrase contains some identifiers in it (this constraints could be removed by adding location information for expressions/patterns in the AST). For example, Figure 9 shows the AST and token stream representations of the same syntax phrase: *if a then b else c*, which is extracted from some example code. For each identifier, there is a natural correspondence between its AST representation and its token representation because of the source location information.

Inspired by this mapping between AST and token stream, HaRe's solution to preserving program appearance is to make use of both the AST and the token stream. In this approach, the refactorer still carries out program analysis with the AST, but it performs program transformation with both the AST and the token stream: whenever the AST is modified, the token stream will be modified to reflect the same change in the program source. After a refactoring, instead of pretty-printing the AST to get the refactored program, we extract the program source from the token stream, which is fairly straightforward. We look at this process in more detail in the remainder of this section.

From AST to Token Stream

HaRe takes two views of the program: one in the AST format and the other in the token stream format. The source location information attached to the identifiers

```
  — The AST representation of 'if a then b else c'.
  — Note: in the PNT representation of an identifier, the
  — first location is the identifier's defining location,
  — and the second location is its source location.
(Exp (HsIf
  (Exp (HsId (HsVar
       (PNT (PN (UnQual "a")(S ("D.hs", 37, 5, 5))) Value
            (N (Just ("D.hs", 48, 5, 16)))))))
  (Exp (HsId (HsVar
       (PNT (PN (UnQual "b")(S ("D.hs", 39, 5, 7))) Value
            (N (Just ("D.hs", 55, 5, 23)))))))
  (Exp (HsId (HsVar
       (PNT (PN (UnQual "c")(S ("D.hs", 41, 5, 9))) Value
            (N (Just ("D.hs", 62, 5, 30)))))))))

  — The token stream representation of 'if a then b else c'
(Reservedid, (Pos char = 45, line = 5, column = 13, "if")),
(Whitespace, (Pos char = 47, line = 5, column = 15, " ")),
(Varid, (Pos char = 48, line = 5, column = 16, "a")),
(Whitespace, (Pos char = 49, line = 5, column = 17, " ")),
(Reservedid, (Pos char = 50, line = 5, column = 18, "then")),
(Whitespace, (Pos char = 54, line = 5, column = 22, " ")),
(Varid, (Pos char = 55, line = 5, column = 23, "b")),
(Whitespace, (Pos char = 56, line = 5, column = 24, " ")),
(Reservedid, (Pos char = 57, line = 5, column = 25, "else")),
(Whitespace, (Pos char = 61, line = 5, column = 29, " ")),
(Varid, (Pos char = 62, line = 5, column = 30, "c")),
(Whitespace, (Pos char = 63, line = 5, column = 31, " "))
```

Figure 9: Mapping from AST to token stream

serves as a connection between the AST and the token stream. For an identifier from the AST, its token representation can be found by simply searching the token stream for the token with the same source location. Mapping other syntax phrases from the AST to token stream is slightly less straightforward because keywords and special characters, such as brackets and comma, are not part of the abstract syntax, and there is no span information, i.e. the begin and end position of a syntax phrase in the source, stored in the AST.

Under this approach, we first find the tokens for the first and last identifier

in the abstract syntax phrase, then look to the left of the first token and/or to the right of the last token to include the tokens for those keywords or special characters preceding/following the first/last identifier in the concrete syntax phrase. For example, in order to map the AST representation of the expression *if a then b else c*, shown in Figure 9, to its token stream representation, we fetch the locations of the identifiers a and c from their *PNT* representations in the AST, and find their corresponding tokens in the token stream based on the locations, then search backwards from the token representing the identifier a to find the token for the keyword *if*. For this purpose, the class *StartEndLoc*, as shown below, with a single method called *startEndLoc* has been defined to fetch the start and end location of the corresponding token sequence of arbitrary abstract syntax phrases.

```
class StartEndLoc t where
  — PosToken: token with position information;
  — type SimpPos = (Int, Int)
startEndLoc :: [PosToken] → t → (SimpPos, SimpPos)
```

startEndLoc takes the token stream of the program and the syntax phrase as arguments, and returns the start and end location of the syntax phrase in the program source in terms of row and column numbers. White space before/after the syntax phrase is not covered by the start/end location.

How To Handle Newly Created Code

For the mapping from AST to token stream to work properly, we require: a) the same occurrence of an identifier has the same source location information in both the AST and the token stream; and b) the locations are unique. While this is natural for a freshly lexed and parsed program, it may not always be true without proper care of the newly created code during the program transformation phase.

A newly created piece of code (the AST representation) could be derived from the existing code, or composed from scratch by the refactorer. Whenever it is

possible, we try to get the new code's string representation from the existing token stream, otherwise the refactorer will pretty-print the AST of the newly created piece of code to create its string representation, as the case for the *add a discriminator* refactoring.

In both cases, the new code's string representation will be lexed (tokenised) with a fresh start source location, and the location for each identifier in the newly created tokens will be injected back to the new code's AST representation, so as to keep the source locations between the token stream and the AST consistent. For example, suppose the expression *if a then b else c* is composed by the refactorer from a case expression, and this conditional expression will be pretty-printed by the refactorer and replace the case expression in the source. The composed AST representation of the conditional expression (extracted from one of our experiment results) is:

$$
\begin{aligned}
&(Exp\ (HsIf\ (Exp\ (HsId\ (HsVar\\
&\quad (PNT\ (PN\ (UnQual\ \text{“}a\text{”})(S\ (\text{“}D.hs\text{”}, 21, 3, 5)))\ Value\\
&\qquad (N\ (Just\ (\text{“}D.hs\text{”}, 34, 3, 18)))))))\\
&\quad (Exp\ (HsId\ (HsVar\\
&\qquad (PNT\ (PN\ (UnQual\ \text{“}b\text{”})(S\ (\text{“}D.hs\text{”}, 23, 3, 7)))\ Value\\
&\qquad (N\ (Just\ (\text{“}D.hs\text{”}, 62, 4, 24)))))))\\
&\qquad (Exp\ (HsId\ (HsVar\\
&\quad (PNT\ (PN\ (UnQual\ \text{“}c\text{”})(S\ (\text{“}D.hs\text{”}, 25, 3, 9)))\ Value\\
&\qquad (N\ (Just\ (\text{“}D.hs\text{”}, 75, 5, 24)))))))))
\end{aligned}
$$

where the source locations no longer reflect the identifiers' actual locations.

By tokenising the pretty-printed expression "*if a then b else e*" with a fresh start location, let's use (-1000, 1) in this example, we get a sequence of tokens as shown below:

$$
\begin{aligned}
&(Reservedid, (Pos\ char\ =\ 0,\ line\ =\ -1000,\ column\ =\ 1, \text{“}if\text{”})),\\
&(Whitespace, (Pos\ char\ =\ 0,\ line\ =\ -1000,\ column\ =\ 3, \text{“ ”})),\\
&(Varid,\ (Pos\ char\ =\ 0,\ line\ =\ -1000,\ column\ =\ 4, \text{“}a\text{”})),\\
&(Whitespace, (Pos\ char\ =\ 0,\ line\ =\ -1000,\ column\ =\ 5, \text{“ ”})),\\
&(Reservedid, (Pos\ char\ =\ 0,\ line\ =\ -1000,\ column\ =\ 6, \text{“}then\text{”})),
\end{aligned}
$$

$$
\begin{aligned}
&(\mathit{Whitespace}, (\mathit{Pos}\ \mathit{char}\ =\ 0,\ \mathit{line}\ =\ -1000,\ \mathit{column}\ =\ 10, \text{“ ”})),\\
&(\mathit{Varid},\ (\mathit{Pos}\ \mathit{char}\ =\ 0,\ \mathit{line}\ =\ -1000,\ \mathit{column}\ =\ 11, \text{“}b\text{”})),\\
&(\mathit{Whitespace}, (\mathit{Pos}\ \mathit{char}\ =\ 0,\ \mathit{line}\ =\ -1000,\ \mathit{column}\ =\ 12, \text{“ ”})),\\
&(\mathit{Reservedid}, (\mathit{Pos}\ \mathit{char}\ =\ 0,\ \mathit{line}\ =\ -1000,\ \mathit{column}\ =\ 13, \text{“}\mathit{else}\text{”})),\\
&(\mathit{Whitespace}, (\mathit{Pos}\ \mathit{char}\ =\ 0,\ \mathit{line}\ =\ -1000,\ \mathit{column}\ =\ 17, \text{“}c\text{”}))
\end{aligned}
$$

In the above token stream, the value for the *char* field is always zero. This field is not used by the refactorer, and we set it to zero to indicate that the token is a newly created token.

By updating the AST of “*if a then b else e*” with the new locations, we have:

$$
\begin{aligned}
&(\mathit{Exp}\ (\mathit{HsIf}\ (\mathit{Exp}\ (\mathit{HsId}\ (\mathit{HsVar}\\
&\quad (\mathit{PNT}\ (\mathit{PN}\ (\mathit{UnQual}\ \text{“}a\text{”})(S\ (\text{“}D.hs\text{”}, 21, 3, 5)))\ \mathit{Value}\\
&\qquad (N\ (\mathit{Just}\ (\text{“}D.hs\text{”}, 0, -1000, 4)))))))\\
&(\mathit{Exp}\ (\mathit{HsId}\ (\mathit{HsVar}\\
&\quad (\mathit{PNT}\ (\mathit{PN}\ (\mathit{UnQual}\ \text{“}b\text{”})(S\ (\text{“}D.hs\text{”}, 23, 3, 7)))\ \mathit{Value}\\
&\qquad (N\ (\mathit{Just}\ (\text{“}D.hs\text{”}, 0, -1000, 11)))))))\\
&(\mathit{Exp}\ (\mathit{HsId}\ (\mathit{HsVar}\\
&\quad (\mathit{PNT}\ (\mathit{PN}\ (\mathit{UnQual}\ \text{“}c\text{”})(S\ (\text{“}D.hs\text{”}, 25, 3, 9)))\ \mathit{Value}\\
&\qquad (N\ (\mathit{Just}\ (\text{“}D.hs\text{”}, 0, -1000, 17)))))))))
\end{aligned}
$$

The fresh start source location is created by the refactorer automatically, and this aims to guarantee the uniqueness of source locations. The fresh location does not have to be the new code's actual start location in the program source, as the concrete value of the location has no effect on the mapping between AST and token stream.

Tokenising the newly created code not only allows us to keep the mapping from AST to token stream correct, but also allows the refactorer to manipulate the new code in both the AST and the token stream correctly, which is necessary for some refactorings, such as *swapping the arguments of a function definition.*

Basic Token Stream Operations

Apart from the class *StartEndLoc*, a collection of token stream manipulation functions have been defined in HaRe, and among which are:

- Functions for classifying tokens based on the token's content and category. For example the function $isWhere :: PosToken \rightarrow Bool$ checks whether a token represents the reserved identifier *where*.
- The function $getToks :: (SimpPos, SimpPos) \rightarrow [PosToken] \rightarrow [PosToken]$, which takes an abstract syntax phrase as input and extracts its token stream representation.
- Functions for modifying the token stream. They are:
  $replaceToks :: [PosToken] \rightarrow SimpPos \rightarrow SimpPos \rightarrow [PosToken] \rightarrow [PosToken]$,
  where *replaceToks tokens start end new* replaces the subsequence of *tokens* specified by the *start* and *end* locations by the *new* sequence;
  $addToks :: [PosToken] \rightarrow [PosToken] \rightarrow PosToken \rightarrow [PosToken]$,
  where *addToks tokens new t* adds the sequence of tokens, *new*, after the specific token t in the token stream *tokens*; and
  $deleteToks :: [PosToken] \rightarrow SimpPos \rightarrow SimpPos \rightarrow [PosToken]$,
  where *deleteToks tokens start end* deletes the subsequence of *tokens* specified by the *start* and *end* locations from the token stream.

  The above three functions should be used together with AST transformations, and the replaced/added/deleted tokens should correspond to proper syntax phrase(s) in the program.
- A core function $adjustLayout :: [PosToken] \rightarrow Int \rightarrow Int \rightarrow [PosToken]$, which encodes a layout adjustment algorithm (see next section for details) and serves to adjust the token stream to compensate for the layout changes because of replacing/adding/deleting tokens. This function is called by *replaceToks*, *addToks* and *deleteToks*. for adjusting the program layout.

The Layout Adjustment Algorithm

There are two issues to do with layout preservation for Haskell programs. One issue is to keep the layout correct so that the refactored program does not violate any layout rules; the other issue is to ensure that the code is as much as possible like the original one in appearance. The layout adjustment algorithm mainly addresses the first issue. The second issue is relatively straightforward, as the token stream for the code which is irrelevant to the refactoring is not touched by the refactorer, and their layout can be preserved naturally when extracting and concatenating the contents of the tokens. For the code which is affected by the refactoring, there is no standard algorithm, but we try to use the existing layout information as much as possible in the implementation of individual refactorings. We discuss the layout adjustment algorithm next.

Haskell programs can be written in either layout-sensitive or layout-insensitive style as described in the Haskell 98 report [53]. Most Haskell programmers tend to use the layout-sensitive style. A layout-sensitive program uses *program layout* (or *the layout rule*, see Section 9.3 of the Haskell 98 report [53]) to convey the information which is otherwise provided by braces and semicolons. Informally stated, the layout rule takes effect whenever the open brace is omitted after the keyword *where*, *let*, *do* or *of*. When this happens, the indentation of the new lexeme is remembered and the omitted open brace is inserted. For each subsequent line, if it contains only whitespace or is indented more, then the previous item is continued; if it is indented the same amount, then a new item begins; and if it is indented less, then the layout list ends and a close brace is inserted. As the meaning of a Haskell program may depend on its layout, it is essential for a refactorer not to violate the layout rules when transforming a program. However, naïve updating of the token stream can destroy the existing layout very easily as shown in the *output* 1 part of Figure 10, where the identifier f is renamed to *sumSq*. The correct result in shown in the *output* 2 part of that Figure, where white space has been added to shift the affected line(s) to the right.

As only the layout of the source which is lexcially after the point where modification has been made could possibly be affected by a token stream manipulation, the function *adjustLayout* starts by checking whether the offset of the token, say t, which is right after the modified token sequence, has been changed, and whether a layout rule applies at some point between t and the end of the line to which t belongs. If neither of the checks is positive, then the layout has not been violated and the token stream will remain unchanged; otherwise, the following lines will need to be shifted to the left/right by removing/adding a number of whitespace tokens until a line has been reached whose indentation is less than t's original indention. The number of whitespace added/removed is decided by the change to t's indention. The implementation of *adjustLayout* is given in Appendix C.

— The original program.

module *Test* **where**

$f\ x\ y\ =\ sq\ x\ +\ sq\ y$ **where** $sq\ x\ =\ x^{\wedge}\ pow$
$pow\ =\ 2$
$test\ =\ f\ 10\ 20$

— Output 1: only update the token containing 'f'.

module *Test* **where**

$sumSq\ x\ y\ =\ sq\ x\ +\ sq\ y$ **where** $sq\ x\ =\ x^{\wedge}\ pow$
$pow\ =\ 2$
$test\ =\ sumSq\ 10\ 20$

— Output 2: update the token containing 'f', and adjust the layout.

module *Test* **where**

$sumSq\ x\ y\ =\ sq\ x\ +\ sq\ y$ **where** $sq\ x\ =\ x^{\wedge}\ pow$
$pow\ =\ 2$
$test\ =\ sumSq\ 10\ 20$

Figure 10: A layout sensitive Haskell program

Association of Comments and Program Entities

Comments usually rely on the commented program entities to exist. During the refactoring process, when a program entity has been moved/removed, the comments for that program entity should be moved/removed as well. Even more, in some cases the comments need also to be refactored to be consistent with the refactored program entity. HaRe does not support refactoring the content of comments so far, but it tries to move/remove comments together with the commented program entities if possible.

Like most programming languages, there is no standard for associating comments with program entities in Haskell 98. In principle, a programmer can put the comments for a program entity randomly as long as this does not violate the syntax rules. In practice, people tend to put comments next to the commented program entity. Based on this fact, we use some heuristics to decide when to move/remove comments together with definitions. For example, when moving a definition, we also move the last consecutive comments right before this definition (with at most one empty line between the comments and the definition); we also move the comment whose start location is in the same line as the definition's last line of code. This was also made possible by the token stream manipulation.

Another idea to keep the consistence between comments and the commented code is to highlight those comments which are most likely to need examination after a refactoring has taken place. This is not supported by the current implementation of HaRe, but worth further exploration.

Discussion

Program appearance preservation is a general problem when tool support for interactive source-to-source program transformation is concerned. The approach taken for HaRe is substantially affected by the chosen frontend, i.e., Programatica, and the design decision to minimise the modifications to Programatica. However, whatever approach is taken, keeping the correct program layout and the proper

association of comments and program entities are unavoidable problems.

Program appearance preservation is not trivial, especially for programming languages that do not have an editor-enforced standard layout, and previous research effort has already been spent on it. Extending abstract syntax trees, or parse trees, with layout and comment information is currently the most popular approach to recording program appearance information. For example, this approach has been used by the Proteus project [115] in the implementation of a high-fidelity C transformation system. This system effectively uses a special form of AST which contains layout and comment information, and automatically propagates layout and comment information during transformations. A more detailed description of Proteus is given in Section 8.1.3. ASF+SDF [56, 109] is a meta-environment supporting source code analysis and transformation using term rewriting. SDF (Syntax Definition Formalism) is used for describing the syntax of programming languages, and ASF (Algebraic Specification Formalism) is for describing their semantics. The system has recently been extended so that layout can be parsed and rewritten like any other program structure [110]; support for this in the infrastructure of the system has required substantial changes to the system.

4.2.4 The Interface of HaRe

HaRe can be invoked from either of the two program editors: Vim and (X)Emacs, or from the command line.

The Editor Interface

HaRe is integrated with two program editors: Vim and (X)Emacs. The integration was mainly designed and implemented by Claus Reinke. To make the explanation of HaRe complete, a sequence of snapshots of HaRe embedded in Emacs are given in Figures 11-13. This sequence of figures also show the process of generalising the function f over the subexpression 2. In Figure 11, the subexpression 2 has been

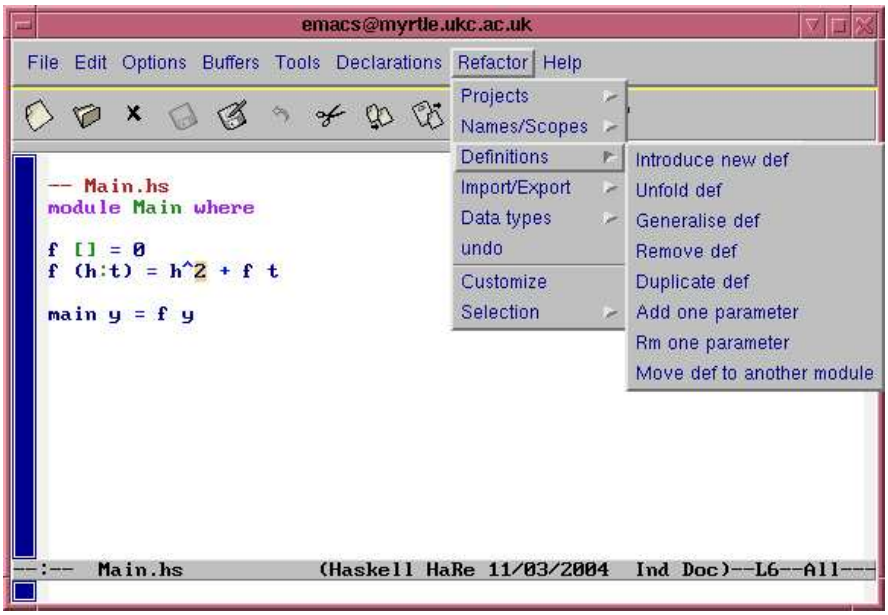

Figure 11: A snapshot showing the application of *generalising a definition*

highlighted, and the command *Generalise def* is being chosen from the *Definitions* submenu, which is in the *Refactor* menu. After that, the refactorer would prompt the the user to input the name of the new parameter in the mini-buffer, as shown in Figure 12. After having input the parameter name as m, the user would press the *Enter* key. The result of the refactoring is shown in Figure 13: the function f has been generalised over the subexpression 2 with a new parameter m, and 2 is now an argument of f at its call-site(s) outside the definition of f, and within the definition of f, m is supplied to the recursive call(s) of f as an argument.

The Command line Interface

pfe is Programatica's front-end tool, and can be invoked from the command line. It supports a host of commands covering from basic project management to type checking, simple program transformations, metrics, etc. We extended Programatica's command set with our own refactoring commands, and these refactoring commands can be invoked from the command line just like those from Programatica itself. For example, the following command line snapshot shows how the *generalising a definition* refactoring can be invoked from the *pfe* environment.

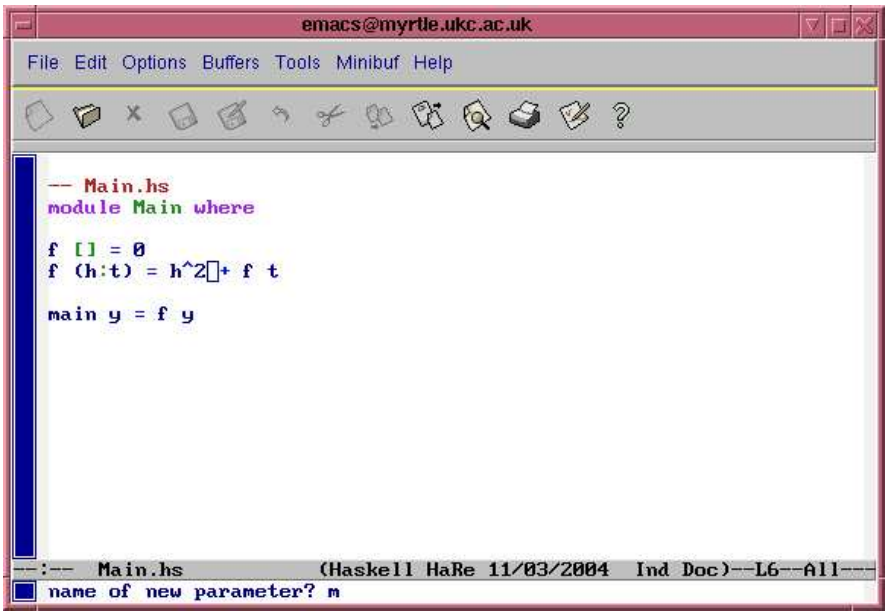

Figure 12: A snapshot asking for the parameter name during *generalisation*

```
myrtle$ pfe
generaliseDef /home/cur/hl/HaRe/test/Main.hs m 6 13 6 14
Analyzing: /home/cur/hl/HaRe/test/Main.hs
modified: /home/cur/hl/HaRe/test/Main.hs
```

In the above *generaliseDef* command, 6 and 13 specifies the start line number and column number of the identified expression respectively; 6 and 14 specified the end line and column number of the expression; and *m* is the new parameter name. This commands performs the same refactoring as shown in Figure 11. The two lines that follow the *generaliseDef* command is outputted by the refactorer.

4.2.5 The Implementation Architecture

To conclude this section, we summarise HaRe's implementation architecture as shown in Figure 14.

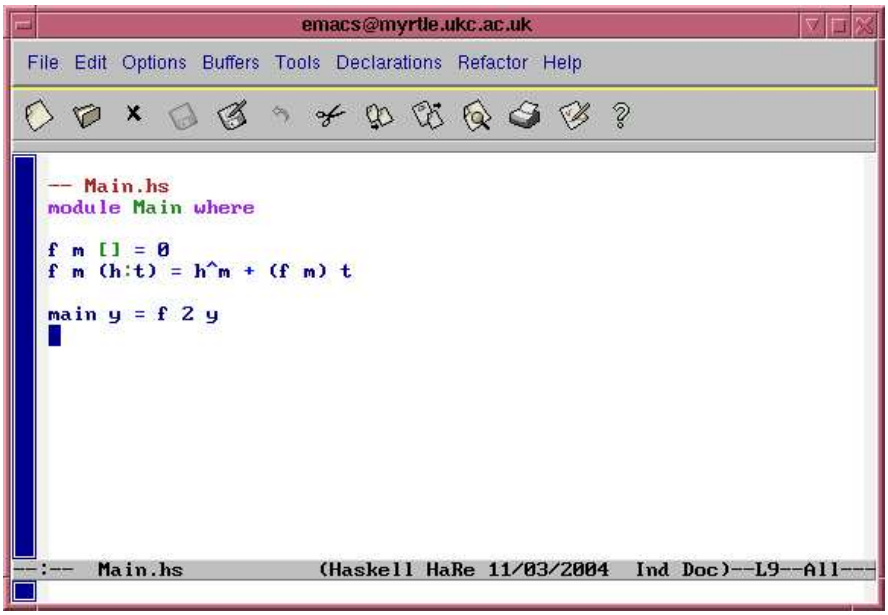

Figure 13: A snapshot showing the result of generalising a definition

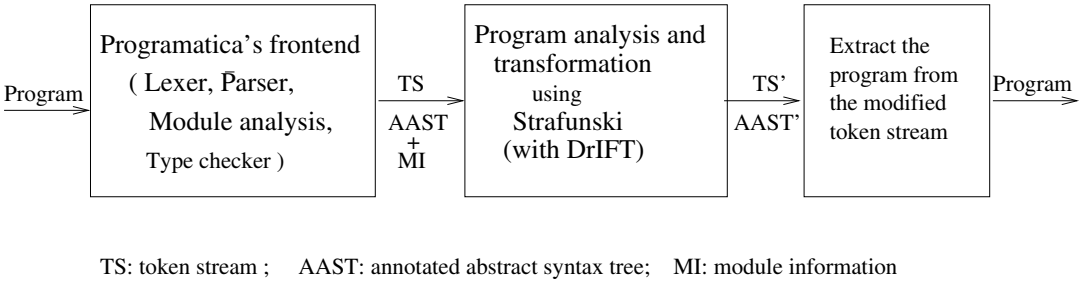

Figure 14: The implementation architecture of HaRe

Chapter 5

The Implementation of HaRe

The initial release of HaRe, in October 2003, supported a dozen of scope-related single-module refactorings. Since then, single-module refactorings have been extended to support multi-module programs; various data-oriented refactorings and module refactorings have been added; and an API has been exposed to HaRe's infrastructure for implementing refactorings and more general program transformations. Apart from these external evolutions, the internal implementation of refactorings has been restructured mainly in two aspects:

- Factoring out the common token stream and AST manipulations. In the previous implementation of HaRe, explicit token stream and AST manipulations were used heavily. An AST manipulation is normally accompanied with a token stream manipulation in order to keep the AST and token stream consistent. Keeping two views of the program in mind is hard, and the code produced looks complex. We therefore refactored our implementation, and extracted a number of functions to encapsulate the basic token stream and AST manipulations. These functions are now part of HaRe's API which will be discussed in Chapter 6. For instance, one of the representative functions we have extracted is *update*. This function replaces a syntax phrase with another of the same type in both the AST and the token stream. Using

these extracted functions, we were able to eliminate the explicit manipulations of token stream in the implementation of individual refactorings, and make the code much easier to understand and to write (enabling others to contribute the project).

- Separation of condition checking and program transformation. Another major refactoring to HaRe's implementation is that we separated the condition checking part from the program transformation part. The advantage of doing this is three-folded: first, it helps to make the implementation clearer, more readable and maintainable; second, this helps to extract the common side-conditions among different refactorings; and third, the separation makes it easier to skip the unnecessary side-condition checking when elementary refactorings are composed into composite ones.

This chapter presents the internal implementation of refactorings. We take two refactorings as examples: *rename a variable name* and *from concrete to abstract data type*. Using the first refactoring, we explain the implementation of elementary refactorings and how multi-module programs are handled; through the second refactoring, we explain how a composite refactoring can be composed from elementary refactorings. Before that, we give an overview of the module architecture of HaRe.

5.1 The Module Architecture of HaRe

Figure 15 illustrates the module hierarchy of HaRe. At the bottom of this structure are the infrastructure modules on which HaRe is built, including about 210 modules from Programatica's Haskell frontend, the module *RefacLocUtils* which contains a repertoire of token stream related functions, the *DriftStructUtils* module generated by running DrIFT over Programatica's data types defining the abstract syntax, and the Strafunski library's top-level module *StrategyLib*.

Built on top of these infrastructure modules is the module *RefacUtils*. This

module defines HaRe's API, as presented in Chapter 6, and also exports the abstract syntax defined by Programatica, module *StrategyLib* and *DriftStructUtils*.

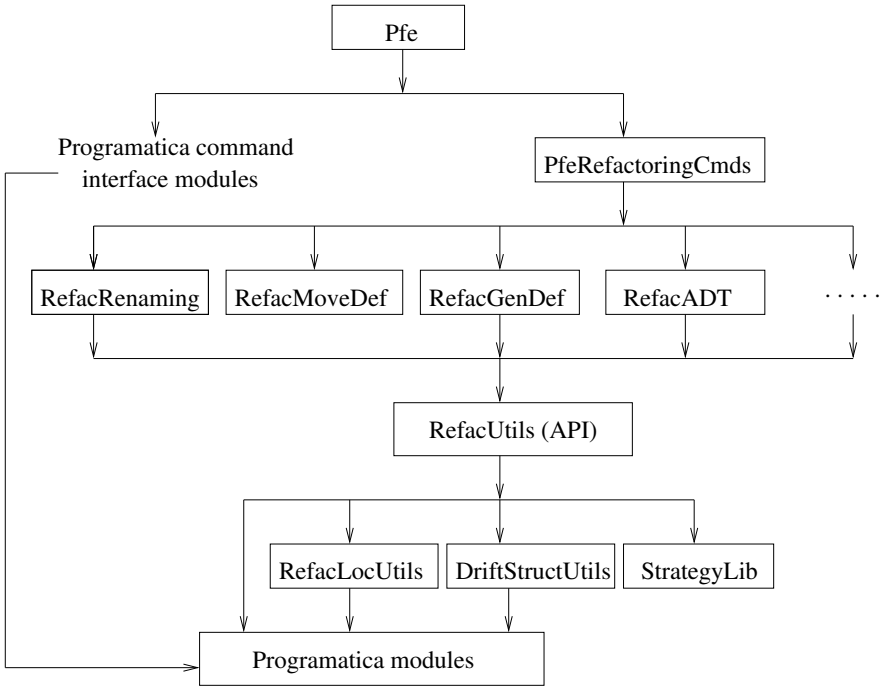

Figure 15: The module hierarchy of HaRe

At the centre of the hierarchy are those modules implementing individual refactorings. Each module contains one or more refactorings. Normally, only closely related refactorings are put together in a single module. For example, the module *RefacMoveDef* includes three refactorings: promote a definition one level, promote a definition to the top level, and demote a definition one level. All refactoring commands are assembled in the module *PfeRefactoringCmds*, as an extension to the frontend commands provided by Programatica. Finally at the top of the hierarchy is the *Main* module defined by Programatica, but modified by us, in the file *pfe.hs*.

5.2 Implementing Elementary Refactorings – *Renaming* as an Example

While refactorings differ from each other in their side-conditions and transformation rules, their implementation normally follows a similar pattern. In this section, we go through the implementation of *rename a variable*, with an aim to shed light on the implementation of other elementary refactorings. The variable to be renamed can be either a top-level identifier or a local identifier.

Renaming is one of the basic, but useful, refactorings supported by almost all the available refactoring tools. The fundamental requirement for *renaming* is not to violate the binding structure of the program. Basic as it is, the implementation is by no means a trivial task due to the many layers of nesting scopes that a Haskell program can have.

In a multi-module program, renaming a top-level exported identifier affects not only the module, say A, containing the identifier's definition, but also those modules importing A. A refactoring on a multi-module program succeeds if and only if the refactoring succeeds on all the affected modules. When refactoring multi-module programs, we first carry out side-condition validation over the program, then transform the module where the refactoring is initiated, and after that, transform those modules importing the refactored module. For convenience, we refer to the module where the refactoring is initiated as the *current module* (the *renaming* refactoring can only be initiated from the module where the identifier is defined, hence the *current module* is also the identifier's *defining module*), and those modules that import the current module as the *client modules*. Normally the *current module* and the *client modules* need different transformations, but all the *client modules* use the same transformation rules.

To explain the implementation of *rename a variable*, we start by assuming that the *current module* is not imported by any other modules, then extend the implementation to get rid of this assumption. The complete implementation of this refactoring is given in Appendix D, and we refer to that implementation when

we talk about the functions used in the implementation.

5.2.1 Refactoring the *Current Module*

As discussed in Section 4.1, a refactoring process normally involves the following steps: transforming program source into internal representations; locating the focus of the refactoring; validating the side-conditions of the refactoring; transforming the program according to the transformation rules; and presenting the refactored program to the user. These steps are reflected in the top-level function of individual refactorings. The top-level function of *rename a variable* is as shown in Figure 16. This function only refactors the current module, and it takes 4 arguments: the name of the file containing the declaration of the variable, say x, the new variable name, say y, and row and column number of one of the variable's occurrence.

Among the called functions, *locToPNT* (see Appendix F) turns the textual selection of the identifier into its *PNT* representation, one of the abstract representations of identifiers defined by Programatica [81].

```
  — The top-level function of renaming
rename fileName newName row col
= do
    modName ← fileNameToModName fileName
      — inscps: in-scope entities; exps: exported entities; mod: the AST;
    info@(inscps, exps, mod, _) ← parseSourceFile fileName
      — turns textual selection to the PNT represenation
    let pnt@(PNT pn _ _) = locToPNT fileName (row, col) mod
      — condition checking in the curremt module
    condChecking pn newName modName (inscps, exps, mod)
      — transformation: renaming in the current module
    r ← applyRefac (doRename pn newName modName) (Just info) fileName
    writeRefactoredFiles False [r]    —  output the result.
```

Figure 16: The top-level function of *rename a variable*.

The side-condition checking function *condChecking* consists of two parts. The first part is defined in the local function *condChecking*1, see Appendix D, which

does some trivial checking that do not need AST traversals, such as whether the new name is a lexically valid variable identifier, whether the name to be renamed is the *main* defined at the top-level of the *Main* module, whether the new name will cause *name clash* in the export list of the current module, etc. The second part is defined in the local function *condChecking*2. This function performs a top-down traversal of the AST until it reaches a syntax entity, say E, such that E contains the declaration of x, and all the references to the x in question. E could be the Haskell module, a declaration defining a function, a declaration defining a pattern binding, an expression, a branch in a case-expression, or a do statement. The syntax phrase E forms the context for condition checking, and at the place where it is reached, the function *condChecking*$'$ is called, and the traversal terminates.

Inside the function *condChecking*$'$, three conditions are checked. The first condition ensures that the new name does not exist in the same binding group, where the function *declaredVarsInSameGroup* (from the API) is used to fetch all the variable names declared in the same binding group where x is declared. The second condition checks whether the new name will intervene between the existing uses of y and its bindings, where function *hsFreeAndDeclaredNames* is used to fetch the free and declared variables in the argument syntax phrase. The third condition checks whether the new name is declared somewhere between the declaration of identifier to be renamed and one of its call-sites, and function *hsVisibleNames* is used to collect the names which are declared in the given syntax phrase and visible to one of the call-sites of the identifier. In the local functions, including *inMatch*, *inPattern*, and *inAlt*, the values *defaultPNT* and/or [ ] are used to shadow those variables declared in the same syntax phrase but in an outer scope.

In this implementation, *ambiguous references* are avoided using qualified names, therefore it is not checked during the side-condition validation phase.

doRename is the function that actually performs the renaming. This function does two things: it renames all the occurrences of the identified variable to the new name, and it qualifies the uses of the new name if otherwise an *ambiguous*

reference would happen. Function *renamePN′* is used to do renaming in the import list, and *renamePN* is called to do renaming in the export list and the body of the module. The only difference between *renamePN′* and *renamePN* is that *renamePN′* does not check whether the name needs to be qualified, as qualified names can not be used in the import list. Function *renamePN* traverses the AST in top-down order searching for the *PNT* representation of identifiers, where it calls the function *inPNT*. This latter function checks and renames the identifier according to several conditions:

- The identifier will be renamed if it has the same define location as x, and the new name y does not cause *ambiguous reference*. If the identifier is qualified/unqualified before the renaming, then it is still qualified/unqualified after the renaming.
- The identifier will be renamed and qualified if it has the same define location as x, but this occurrence of x will cause *ambiguous reference* after renaming.
- The identifier will be qualified if it has the same name as y, and is unqualifiedly used in a scope where the x in question is visible.
- The identifier will be left as it is in all other cases.

The function *update* is used to update the identifier in both the AST and the token stream.

5.2.2 Refactoring the *Client Modules*

For the *renaming* refactoring to work with multi-module programs, both the side-condition checking and the program transformation need to be extended.

In a module that imports the renamed identifier, the only possible non-resolvable violation is *name clashes* in the module's export list. So we have defined another condition checking function *clientModsCondChecking*, to ensure that none of the client modules will have conflicting exports after the renaming. Another function

doRenameInClientMod has been defined to perform renaming in the client modules. This function is similar to *doRename* in terms of the traversal strategy, but is different from *doRename* at two points:

- Qualified names are used to resolve both ambiguity and name capture.
- The qualifier for the renamed variable has to be inferred from the in-scope relation of the module.

Finally, the top-level function of the refactoring, *rename*, is extended to accommodate the side-condition checking and transformation for the client modules, and the implementation becomes complete.

5.2.3 Some Strategies for Refactoring Multi-module Programs

The general strategy is to minimise the amount of AST traversal, program analysis and transformation.

As to side-condition checking, we try to confine the scope of checking within the module where the refactoring is initiated, and resolve the possible problems caused in the client modules during the program transformation phase. For example, lifting a function definition to the top level of a module may expose this identifier to other modules. In this case, instead of checking each module for possible *ambiguous references* or *name clashes* in the export lists, we chose to hide the identifier in those import declarations which explicitly import the current module without explicitly specifying the imported entities, so that none of the client modules' in-scope/export relations will be changed after the refactoring. Qualified names can also be used to resolve some ambiguity and conflict problems. In the case that a violation can not be resolved using these techniques, condition checking over multiple modules is unavoidable.

As to the program transformation phase, the refactorer needs to go through every affected module to perform the transformation rules. Nevertheless, we still

try to keep the transformation as simple as possible. For example, when generalising an exported top-level binding, it is possible that some of the global variables contained in the identified expression are not visible to some of the use-sites of the generalised function in the client modules. In this case, instead of analysing and transforming each client module to make those free variables available, we choose simply to bind the identified expression to a newly created identifier in the current module, and to make the identifier visible to those involved modules by exporting/importing this identifier.

5.3 Implementing Complex Refactorings

A complex or large scale refactoring normally involves a number of transformation steps. When a number of separate transformations have been performed, it might become hard to maintain the correctness of the binding structure in the AST, and difficult to continue the transformation. In this case, one solution is to decompose the complex refactoring into a number of simpler elementary refactorings (some of these elementary refactorings might already exist), so that performing these elementary refactorings in a specific order (sequentially or iteratively) would achieve the same effect as the complex refactoring does.

Suppose a complex refactoring, R, is decomposed into a chain of elementary refactorings: R_1, R_2, ... , R_n, then each of these elementary refactorings will have its own side-conditions and transformation rules, therefore can be applied in its own right, and re-used in other refactoring scenarios.

Closely relevant to the decomposition of complex refactorings, a number of elementary refactorings could also be composed into a composite refactoring to perform a complex refactoring task. In this way, instead of invoking the involved elementary refactorings one after another, we only need to invoke the composite refactoring once to achieve the same effect. Moreover, composite refactorings could possibly need less side-condition checking, and be more efficient. For example, when the elementary refactorings, R_1, R_2, ..., R_n, are composed sequentially into

a composite refactoring, R, the side-condition of R may not be the sum of the elementary refactorings' side-conditions, as R_i may establish the side-conditions of $R_{i+j(j=0..(n-i))}$.

More about composite refactorings, especially the derivation of composite refactorings' side-conditions, is discussed by D. B. Roberts in [94], M. Cinnide, *et al.* in [22], and G. Kniesel, *et al.* in [57].

One complex refactoring implemented in HaRe is *from concrete to abstract data type*. This refactoring turns a user-identified data type into an abstract data type (ADT). In the implementation, we decomposed this refactoring into 6 elementary refactorings: *add field names*, *add discriminators*, *add constructors*, *remove nested patterns*, *eliminate the explicit uses of data constructors*, and *creating the ADT interface*, as described in Section 2.6.3. Implementing these elementary refactorings follows a similar pattern as the implementation of *renaming*, and each refactoring has its condition checking and transformation completely separated. These elementary refactorings have also been composed into the composite refactoring *from concrete to abstract data type*, in which all the elementary refactorings' side-condition checking have been skipped, as this composite refactoring does not have any side-conditions. The top-level function of this composite refactoring is given in Figure 17, where the function seqRefac performs the listed transformations one by one sequentially. The complete implementation of this complex refactoring and its supporting elementary refactorings are given in Appendix E.

In the current implementation, there is no reuse of ASTs between elementary refactorings, but this could be improved in the future research. The main challenge with reusing ASTs is that the refactorer needs to guarantee that no *dirty* information is introduced during the transformation phase and the information stored in the ASTs is up-to-date regarding to the new program. That is, the refactorer should ensure the correctness of existing static semantic information including the binding structure and type information annotated in the ASTs, and the module system information. Furthermore, the location information in the ASTs should also reflect the new status of the program. The reuse of AST is

discussed further in Section 9.2.

```
fromAlgebraicToADT fileName row col
 = do
      info@(_, _, mod, _) ← parseSourceFile fileName
      case locToTypeDecl fileName row col mod of
        Nothing → error "Invalid cursor position!"
        Just decl → do
                      let typeCon = pNtoName $ fromJust $ getTypeCon decl
                      seqRefac [doAddFieldLabels typeCon (Just info) fileName,
                                doAddDiscriminators typeCon Nothing fileName,
                                doAddConstructors typeCon Nothing fileName,
                                doElimNestedPatterns typeCon Nothing fileName,
                                doElimPatterns typeCon Nothing fileName,
                                doCreateADT typeCon Nothing fileName
                               ]
```

Figure 17: The top-level function of *from concrete to abstract data type*.

Figure 18 and 19 show the snapshots of before and after applying *from concrete to abstract data type* to the data type *Tree*.

5.4 Summary

This chapter discussed the implementation of refactorings. So far there are about 8000 lines of code in HaRe (outside of Programatica/Strafunski), about half of which is for the API implementation. The code for implementing a primitive refactoring appears to average out at about 200 lines. The balance between condition checking and transformation differs from refactoring to refactoring. For example condition checking is more complex than transformation in the *renaming* refactoring, whereas transformation is more complex than condition checking in the elementary refactorings of *from concrete to abstract data type*. *Undo* is supported by HaRe, and this allows the users to refactor their program in an exploratory way, since there is no cost in undoing a refactoring which proves not to be useful. There is still a long way for HaRe to progress before it becomes an indispensable part of Haskell programmers' daily used tools, due to the fact

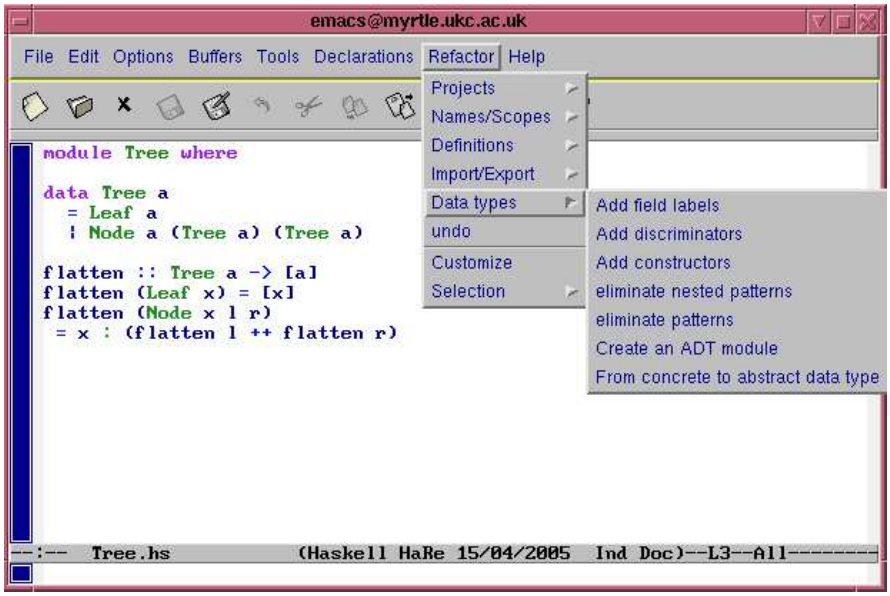

Figure 18: Before applying *from concrete to abstract data type*

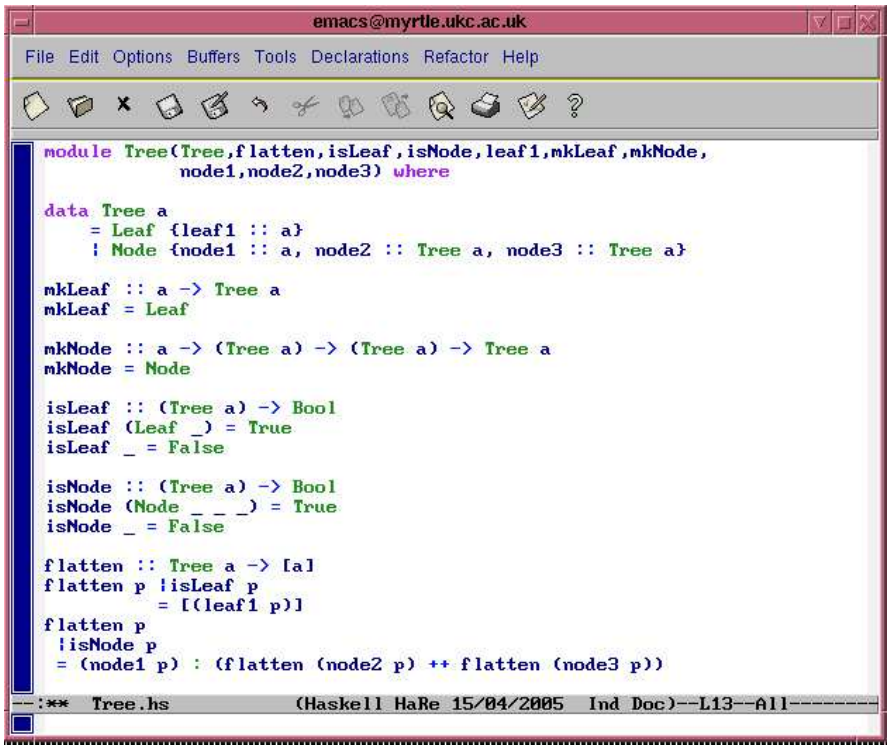

Figure 19: After applying *from concrete to abstract data type* to data type *Tree*

that it does not support most of the Haskell extensions, and that the number of supported refactorings is still limited. However, implementing HaRe provides us with a platform for exploring our refactoring and program transformation ideas, and with the established framework and our experience from implementing the existing refactorings, populating more refactorings into HaRe will not be a problem. As a matter of fact, the established framework is currently being used by Chris Brown, a PhD student in the Computing Lab at the University of Kent, to continue the investigation of tool support for refactoring functional programs, and one refactoring he has implemented using this framework is *duplicated code elimination*. Finally, the implementation of HaRe also makes it possible to integrate some of the existing stand alone Haskell program transformations into HaRe's system. For example, José Proença, from the University of Minho, visited the HaRe team during April-May 2005 and integrated a number of 'pointfree' transformations into HaRe.

Chapter 6

An API for Defining Refactorings

An API has been exposed from HaRe's infrastructure for implementing refactorings. It contains a collection of functions for program analysis and transformation, covering a wide range of syntax entities of Haskell 98. Moreover, the token stream manipulations, used to ensure that layout and comments are preserved, are hidden in the program transformation functions provided by this API. This chapter gives a description of the API and illustrates how it can be used to implement refactorings in a compact and transparent way.

6.1 The Origin of HaRe's API

HaRe's API was derived gradually during our implementation of more than a dozen of refactorings in HaRe. The functions included in the API are not specific to a particular refactoring, but rather common for a wide range of refactorings or general program analysis and transformation. Being well tested and documented, this API could save other developers of HaRe from having to re-implement the same or similar functionalities.

Before the API was written, a function which transforms the program normally contained two parts of code: one part for modifying the AST and the other part for modifying the token stream, and it was the developer's responsibility to coordinate the two kinds of modifications. Our experience showed that it was quite

difficult for other developers of HaRe to understand the token stream manipulation mechanism and they normally chose to ignore the token stream part to start with. By deriving this API, we were able to wrap the token stream manipulations inside the individual program transformation functions. With this API, the collection of token stream manipulation functions introduced in page 62 is no longer visible to the refactoring developers, and a program appearance-preserving transformation can be written without the underlying layout and comment preservation mechanism being noticed.

This domain-specific API is supposed to be used together with Programatica's abstract syntax for Haskell 98 and the Strafunski [65] library for AST traversals, and it serves as the basis for implementing primitive refactorings or general-purpose program transformations. This API alone does not guarantee the behaviour-preservation of refactorings, but it can give a high confidence that the implementation is faithful to its intention, because code written using the API is compact and high level.

6.2 What is in HaRe's API?

HaRe's API covers a wide range of syntax entities of Haskell 98, including identifiers, expression, patterns, declarations, imports, export lists and so forth, and provides functions such as free and declared variable analysis, simplification of multi-equation definitions, updating/adding/removing syntax phrase, etc. These functions fall into three categories: program analysis, program transformation and some miscellaneous functions. What follows is a summary of the API, and a full list of this API can be found in Appendix F.

The Program analysis API. A program analysis function returns some information about the program, but does not modify the program. The program analysis API covers the following aspects:

- Import and export analysis. A collection of functions for checking the imported/exported entities of a module.

- Variable analysis. A collection of functions for different kinds of free and bound variable analyses, and for collecting a specific kind of variable from a given syntax phrase.

- Property checking. Functions for checking whether a given syntax phrase satisfies a specific property, such as whether an identifier is a top-level identifier, whether a syntax phrase contains free variables, etc.

- Modules and files. Functions for module graph analysis.

The Program transformation API. A program transformation function transforms the program from one state to another. The program transformation API covers:

- Adding a given syntax phrase, such as a declaration, to a specified place in the program.

- Removing a syntax phrase, such as a function definition, a parameter, etc, from the program. A syntax phrase can be added/removed only if the result program is still syntactically correct.

- Updating an existing syntax phrase, such as an expression, in the program with a new phrase of the same type.

Some miscellaneous functions. These functions are not for program analysis or transformation purpose, but have also been used extensively in HaRe. They are:

- Functions for lexing/parsing Haskell programs, and for producing the refactored source code from token streams/ASTs.

- Functions for mapping a textual selection to an AST representation.

- Functions for applying a refactoring to a module or a set of modules.

- A function for creating fresh names based on a collection of existing names and a given prefix.

- Functions for converting from expression to patterns and vice versa, and functions for converting between the different abstract representations of identifiers, e.g. from *PNT* representation to *PName* representation.

- Functions for manipulating locations in the AST, such as removing location information, converting absolute locations to relative locations, etc. Converting absolute locations to relative locations allows us to identify a wider range of identity between syntax phrases using equality comparison. For example, after this transformation, we are able to treat the different occurrences of the lambda expression, for instance $\backslash x \rightarrow x + 1$, as the same. This is not possible using absolute locations as the variable x has different defining locations at different occurrences.

6.3 Implementing Refactorings Using HaRe's API

This section illustrates how refactorings can be implemented using the API within HaRe's infrastructure, with an emphasis on the compactness and high level nature of the code produced. For this purpose, we take the implementation of a simplified version of *swap the first two arguments (of a function)* as an example. This refactoring swaps the first two arguments of an identified function at both its definition site and call-sites. For simplicity reason, we assume that the first two arguments are always supplied to the function at its call-sites, and that the types of first two arguments are accessible in the function's type signature if there is one. However, neither of these is an essential restriction.

Figure 20 shows the implementation of *swap the first two arguments*. The top-level function implementing this refactoring is called *swapArgs*. This function takes 3 arguments: the name of the file in which the identified function is declared, the row and column number of one of the function name's occurrences in this file. Using *locToPNT*, the body of the function turns the textual selection of the identifier into its AST representation *pnt*, if possible, and then calls the *doSwap* function to effect the transformation on the current module as well as

those modules which could possibly import the identifier. The function *doSwap* takes the identifier *pnt* as an argument.

In *doSwap* itself, the functions *applyTP*, *full_buTP*, *idTP* and *adhocTP* are type-preserving strategy combinators from Strafunski [65]. *full_buTP* performs a bottom-up traversal of the AST, and its argument strategy searches for abstract syntax phrases of type *HsMatchP*, *HsExpP* and *HsDeclP*, on which it calls functions *inMatch*, *inExp* and *inDecl* respectively. Abstract syntax phrases of any other types are returned without any change. The functions *inMatch*, *inExp* and *inDecl* are the places where the arguments are actually swapped. Function *inMatch* swaps the first two formal arguments of the function definition equation if this equation defines the identifier *pnt*; function *inExp* swaps the first two arguments of *pnt* if the current expression is an application of *pnt* to its at least first two arguments; and function *inDecl* swaps the type signature for the two arguments if the type signature declaration defines *pnt*'s type.

The API function *update* replaces a syntax phrase with a new syntax phrase of the same type in both the AST and the token stream. The functions *locToPNT*, *parseSourceFile*, *isFunPNT*, *isExported*, *applyRefac*, *applyRefacToClientMods*, *update*, *expToPNT* and *isTypeSigOf* are all from the API, and their meaning can be found in Appendix F or in the API documentation available from the HaRe webpage [91].

6.4 Summary

Together with Strafunski and Programatica, the HaRe API reduces the size and complexity of the code implementing refactorings, and improves our confidence in the code produced. The program analysis API can serve as the basic building blocks for implementing side-condition checking. The program transformation API liberates us from modifying the AST and token stream, and allows us to focus on interpreting the transformation rules in the implementation. This way, the implementation can more closely reflect the inherent logic of the refactorings,

instead of the complex implementation details. The API also makes it feasible for the users to design and implement their own refactorings or general program transformations. For example, in summer 2004, Chau Ngyuen Viet worked on implementing *traditional* transformations, such as deforestration, using the API, and his work is reported in the technical report *Transformation in HaRe* [19].

```
module RefacSwapArgs(swapArgs) where
import RefacUtils

swapArgs fileName row col
= do
    modInfo@(_, exps, mod, toks) ← parseSourceFile fileName
    let pnt = locToPNT fileName (row, col) mod
    if isFunPNT pnt mod
    then do
            r ← applyRefac (doSwap pnt) (Just modInfo) fileName
            rs ← if isExported pnt exps
                    then applyRefacToClientMods (doSwap pnt) fileName
                    else return []
            writeRefactoredFiles False (r : rs)
    else error "Invalid cursor position!"
doSwap pnt (_, _, mod)
= applyTP (full_buTP (idTP `adhocTP` inMatch
                          `adhocTP` inExp
                          `adhocTP` inDecl)) mod
  where
    -- At the define site.
    inMatch ((HsMatch loc fun pats rhs ds) :: HsMatchP)
      |fun == pnt
    = case pats of
        (p1 : p2 : ps) → do
                           pats'' ← update p2 p1 =≪ update p1 p2 pats
                           return (HsMatch loc fun pats'' rhs ds)
        _ → error "Insufficient arguments to swap."
    inMatch m = return m

    inExp exp@((Exp (HsApp (Exp (HsApp e e1)) e2)) :: HsExpP)
      | expToPNT e == pnt
    = update e2 e1 =≪ update e1 e2 exp
    inExp e = return e

    inDecl (decl@(Dec (HsTypeSig loc is c tp)) :: HsDeclP)
      |isTypeSigOf pnt decl
    = if length is == 1
        then do
               let (t1 : t2 : _) = tyFunToList tp
               update t2 t1 =≪ update t1 t2 decl
        else error $ "This type signature defines the type of more than one identifier"
    inDecl d = return d

    tyFunToList (Typ (HsTyFun t1 t2)) = t1 : (tyFunToList t2)
    tyFunToList t = [t]
```

Figure 20: Implementation of *swap the first two arguments of a function.*

Chapter 7

Specification and Verification of Refactorings

This chapter focuses on the specification and verification of refactorings. The specification of refactorings aims to give an accurate description of what the refactoring is supposed to do, and in particular to give a clear description of the side-conditions on the refactoring, whereas one of the most important properties to verify is that the specified transformation does not change the behaviour of the program given that the side-conditions of the refactoring are satisfied. Therefore clear specification and sound verification of refactorings help to clarify the definition of refactorings, to guarantee program correctness and behaviour preservation, and so to minimise the need for testing.

As refactoring is ultimately rooted in the semantics of, and program equivalence for, the programming language in question, it is natural to base the specification and verification of refactorings on the definition of the language and its semantics. Compared to imperative languages, pure functional programming languages have a stronger theoretical basis, and reasoning about programs written in pure functional languages is less complicated due to the referential transparency [45] property. This is also manifested by the collection of related work in the

functional programming paradigm where functionality-preserving program transformations are used for reasoning about programs [85], for deriving efficient implementations from program specifications [18, 25, 83], and for compiler optimisation [52].

Two refactorings are examined in detail in this chapter, and they are *generalise a definition* and *move a definition from one module to another*. The former is a typical *structural* refactoring, and the later is a typical *module* refactoring. By examining these two refactorings, we hope to illustrate how other *structural* and *module* refactorings can be described in a similar way. For each of the two, we give its specification consisting of the representation of the program before the refactoring, the side-conditions for the refactoring and the representation of the program after the refactoring, and the verification that the programs before and after the refactoring are equivalent in functionality under the given side-conditions.

Although both *generalise a definition* and *move a definition from one module to another* are module-aware refactorings, we will examine *generalise a definition* without examining its effect on the module system; however, it should be straightforward to extend its specification and verification to cover the module system aspect after *move a definition from one module to another* has been examined. The specification and verification of *data-oriented* refactorings and how type information can be used need in verification further research, and are not discussed in this thesis.

While HaRe is targeted at Haskell 98, our first specification of refactorings is based on the simple λ-calculus augmented with letrec-expressions (denoted as λ_{Letrec}). By starting from λ_{Letrec}, we could keep our specifications and proofs simple and manageable, but still reflect the basic characteristics of refactorings. In the case that a refactoring involves features not covered by λ_{Letrec}, such as data constructors, the type system, etc, we could extend λ_{Letrec} accordingly. Another reason for choosing λ_{Letrec} is that although Haskell has been evolved to maturity in the last two decades, an officially defined, widely accepted semantics for this language does not exist yet.

The remainder of this chapter is organised as follows. Section 7.1 introduces λ_{Letrec}. Section 7.2 presents some definitions and lemmas needed for working with λ_{Letrec}. The formal specification and verification of *generalise a definition* are studied in Section 7.3 and 7.4 respectively. In Section 7.5 , we extend λ_{Letrec} to λ_M to accommodate a simple module system. Some fundamental definitions with the module system are given in Section 7.6. The formal specification of *move a definition from one module to another* is given in Section 7.7, and a rigorous but informal verification of this refactoring is given in Section 7.8. Some conclusions are drawn in Section 7.9.

7.1 The λ-calculus with letrec (λ_{Letrec})

The syntax of λ_{Letrec} terms is:

$$\begin{aligned} E ::= & \; x \\ & \mid \; \lambda x.E \\ & \mid \; E_1 \; E_2 \\ & \mid \; \text{letrec } D \text{ in } E \\ D ::= & \; \varepsilon \mid x_i = E_i \mid D, D \end{aligned}$$

where E represents expressions, and D is a sequence of bindings. Recursive definitions are allowed in a **letrec** expression, and the scope of the *recursion variables* $x_i, 1 \leq i \leq n$, in the expression, letrec $x_1 = E_1, ..., x_n = E_n$ in E, is E and all the E_is. For the same **letrec** expression, we also require that the *recursion variables* $x_i, 1 \leq i \leq n$, are distinct from each other. A **letrec** expression without bindings is allowed and written as: letrec ε in E. No ordering among the bindings in the **letrec** expression is assumed. By convention, we use $\equiv$ to represent syntactic equivalence, and $\doteq$ to represent semantic equivalence. If D_1 and D_2 are the list of declarations $x_1 = E_1, ..., x_m = E_n$ and $y_1 = E'_1, ..., y_n = E'_n$, respectively, such that $\forall i, j : x_i \not\equiv y_j$, then we denote the list of declarations

$x_1 = E_1, ..., x_m = E_n, y_1 = E'_1, ..., y_n = E'_n$ by $D_1 D_2$.

Type information is not reflected in λ_{Letrec}, the main reason for choosing this calculus to represent refactorings is that type information has not been used in the implementation of HaRe; the other reason is that we would like to keep the calculus as simple as possible at the starting point, whilst representing the essential features of Haskell.

As to the reduction strategy, one option for calculating lambda expressions with **letrec** is call-by-need [119, 68], which is an implementation technique for the call-by-name [85] semantics that avoids re-evaluating expressions multiple times by memoising the result of the first evaluation, whereas call-by-name is a normal order, leftmost and outmost reduction, in which argument expressions are passed unevaluated. In the case that behaviour-preservation does not care about introducing or removing sharing of computation, a call-by-need calculus might disallow many refactorings which preserve the observable behaviour, but change the sharing of computation. Therefore, in this study, we use call-by-name for reasoning about program transformations, so that sharing could be lost or gained during the transformation. However, comments about the change of sharing during a refactoring will be given when appropriate.

Instead of developing the call-by-name semantics for λ_{Letrec} from scratch, we make use of the research from the paper *Lambda Calculi plus Letrec* [120], in which Z. M. Ariola and S. Blom developed a call-by-name cyclic calculus ($\lambda\circ_{name}$), where *cyclic* implies that recursion is handled in this calculus. The goal of the paper is to develop a theory of cycles so that more source-to-source program transformation on recursive functions are expressible and reasonable. To this purpose, the authors developed a precise connection in the form of an axiom system between the terms of the lambda calculus extended with **letrec** (cyclic terms) and the class of well-formed cyclic lambda graphs. Among other extensions, the axiom system was extended to be sound and complete with respect to tree unwinding of graphs. The axiom system for tree unwinding combined with a notion of β-reduction constitutes the axiomization of the paper's cyclic lambda structures. The presence

of cycles and lambda-abstraction causes confluence to fail. So, instead of showing confluence, the authors shown that the cyclic calculus satisfies an approximate notion of confluence which guarantees uniqueness of the infinite normal form of a cyclic term. $\lambda\circ_{name}$ defines exactly the same set of terms as λ_{Letrec} does, only with slightly different syntax notation. To make the presentation clearer, we stick to our λ_{Letrec} notation in the remaining of this chapter. What follows are the axioms of $\lambda\circ_{name}$ expressed using the λ_{Letrec} notation of terms.

$\beta\circ$:

$(\lambda x.\ E)\ E_1 \ \doteq\ \text{letrec } x = E_1 \text{ in } E, \quad \text{if } x \notin FV(E_1).$

Substitution :

$\text{letrec } x = E, D \text{ in } C[x] \ \doteq\ \text{letrec } x = E, D \text{ in } C'[E]$

$\text{letrec } x = C[x_1], x_1 = E_1, D \text{ in } E \ \doteq\ \text{letrec } x = C'[E_1], x_1 = E_1, D \text{ in } E$

$Lift$:

$(\text{letrec } D \text{ in } E)\ E_1 \ \doteq\ \text{letrec } D' \text{ in } E'\ E_1$

$E\ (\text{letrec } D \text{ in } E_1) \ \doteq\ \text{letrec } D' \text{ in } E\ E_1'$

$\lambda x.(\text{letrec } D_1, D_2 \text{ in } E)$

$\doteq\ \text{letrec } D_1 \text{ in } \lambda x.(\text{letrec } D_2 \text{ in } E),$ if $D_1 \perp D_2$ and $x \notin FV(D_1)$.

Merge :

$\text{letrec } x = \text{letrec } D \text{ in } E_1, D_1 \text{ in } E \ \doteq\ \text{letrec } x = E_1', D', D_1 \text{ in } E$

$\text{letrec } D_1 \text{ in } (\text{ letrec } D \text{ in } E) \ \doteq\ \text{letrec } D_1,\ D' \text{ in } E'$

Garbage collection :

$\text{letrec } \varepsilon \text{ in } E \ \doteq\ E$

$\text{letrec } D, D_1 \text{ in } E \ \doteq\ \text{letrec } D \text{ in } E,$ if $D_1 \perp D$ and $D_1 \perp E$.

Copying :

$E \ \doteq\ E_1,$ if $\exists \sigma : \nu \to \nu, E_1^{\sigma} \equiv E.$

In the above rules, a $'$ attached to a term indicates that some bound variables

in the term might be renamed to avoid name capture during the transformation; A context $C[]$ is a term with a hole in the place of one subterm, as defined by definition 4 in section 7.2. The first substitution axiom requires that the x in the hole occurs free in $C[x]$, and similarly, in the second substitution axiom, x_1 should occur free in $C[x_1]$. $FV(E)$ means the set of free variables in term E as defined by definition 2 in section 7.2. $D_1 \perp D_2$ means that the set of variables that occur as the left-hand side of an equation in D_1 does not intersect with the set of free variables of D_2.

The *copying* axiom was introduced in order to prove equal every two representations of graphs with the same tree unwinding. In this axiom, σ is a function from recursion variables to recursion variables, and E^σ is the term obtained by replacing all occurrences of recursion variable x by $\sigma(x)$ (leaving the free variables of E unchanged), followed by a reduction to normal form with the unification rule: $x = E, x = E \rightarrow x = E$. The following equality is an example of the copying axiom:

letrec $y = \lambda z.w, w = \lambda x.y$ in y

$\doteq$ letrec $y = \lambda z.w', w' = \lambda x.y', y' = \lambda z.w'$ in y

where the mapping σ is: $w' \rightarrow w, y \rightarrow y$, and $y' \rightarrow y$. In this case, ${E_1}^\sigma$ is

letrec $y = \lambda z.w, w = \lambda x.y, y = \lambda z.w$ in y,

which reduces to letrec $y = \lambda z.w, w = \lambda x.y$ in y.

To make provable equality a congruence relation, we also assume the presence of the following axiom and inference rules.

$$
\begin{aligned}
E &\doteq E \\
E_1 \doteq E_2 &\Rightarrow E_2 \doteq E_1 \\
E_1 \doteq E_2 &\Rightarrow E_2 \doteq E_3 \Rightarrow E_1 \doteq E_3 \\
E_1 \doteq E_2 &\Rightarrow C[E_1] \doteq C[E_2]
\end{aligned}
$$

7.2 The Fundamentals of λ_{Letrec}

This section introduces some definitions and lemmas working with λ_{Letrec}. While these definitions and lemmas are introduced mainly for the specification and verification of *generalise a definition*, most of them should be useful for the specification and verification of other structural refactorings as well.

Definition 1. *Given two expressions E and E', E' is a sub-expression of E (notation $E' \subseteq E$), if $E' \in sub(E)$, where $sub(E)$, the collection of sub-expressions of E, is defined inductively as follows:*

$$sub(x) \doteq \{x\}$$
$$sub(\lambda x.E) \doteq \{\lambda x.E\} \cup sub(E)$$
$$sub(E_1\ E_2) \doteq \{E_1\ E_2\} \cup sub(E_1) \cup sub(E_2)$$
$$sub(\ \textbf{letrec}\ x_1 = E_1, ..., x_n = E_n\ \textbf{in}\ E)$$
$$\doteq \{\ \textbf{letrec}\ x_1 = E_1, ..., x_n = E_n\ \textbf{in}\ E\} \cup\ sub(E)\ \cup ...\ \cup\ sub(E_n)$$

Definition 2. *$FV(E)$ is the set of free variables in expression E, and can be defined as:*

$$FV(x) \doteq \{x\}$$
$$FV(\lambda x\ .\ E) \doteq FV(E) - \{x\}$$
$$FV(E_1\ E_2) \doteq FV(E_1) \cup FV(E_2)$$
$$FV(\ \textbf{letrec}\ x_1 = E_1, ..., x_n = E_n\ \textbf{in}\ E)$$
$$\doteq (FV(E_1) \cup ... \cup FV(E_n) \cup FV(E)) - \{x_1, ...x_n\}$$

Definition 3. *$TBV(E)$ is the set of variables which are bound at the top level of E and can be defined as:*

$$TBV(x) \doteq \{\ \}$$
$$TBV(\lambda x\ .\ E) \doteq \{x\}$$
$$TBV(E_1\ E_2) \doteq \{\ \}$$
$$TBV(\ \textbf{letrec}\ x_1 = E_1, ..., x_n = E_n\ \textbf{in}\ E) \doteq \{x_1, ..., x_n\}$$

Definition 4. *A Context $C[\ ]$ is an expression with a hole in the place of one*

subterm:

$$
\begin{array}{rcl}
C[\,] & \doteq & [\,] \\
& | & \lambda x.C[\,] \\
& | & C[\,]\ E \\
& | & E\ C[\,] \\
& | & \textbf{letrec}\ x_1 = E_1, ..., x_n = E_n\ \textbf{in}\ C[\,] \\
& | & \textbf{letrec}\ x_1 = C[\,], ..., x_n = E_n\ \textbf{in}\ E \\
& \ldots & \\
& | & \textbf{letrec}\ x_1 = E_1, ..., x_n = C[\,]\ \textbf{in}\ E
\end{array}
$$

Definition 5. *A multi-place context $M[\,]$ is an expression with none, one or more holes in the places of subterms:*

$$
\begin{array}{rcl}
M[\,] & \doteq & [\,] \\
& | & x \\
& | & \lambda x.M[\,] \\
& | & M_1[\,]\ M_2[\,] \\
& | & \textbf{letrec}\ x_1 = M_1[\,], ..., x_n = M_n[\,]\ \textbf{in}\ M[\,]
\end{array}
$$

Definition 6. *Given an expression E and a context $C[\,]$, we define $sub(E, C)$ as those sub-expressions of $C[E]$ which contain the hole filled with the expression E, that is:*
$e \in Sub(E, C)$ *iff* $\exists\ C_1[\,],\ C_2[\,],$ *such that* $e \equiv C_2[E] \wedge C[\,] \equiv C_1[C_2[\,]]$.

Definition 7. *The result of substituting N for the free occurrences of x in E with automatic renaming is defined as:*
$x[x := N] \doteq N$
$y[x := N] \doteq y;\ y \not\equiv x$
$(E_1E_2)[x := N] \doteq E_1[x := N]E_2[x := N]$

$(\lambda x.E)[x := N] \doteq \lambda x.E$

$(\lambda y.E)[x := N] \doteq \lambda z.E[y := z][x := N]$,

where $y \not\equiv x$, *and* $z \equiv y$ *if* $x \notin FV(E) \vee y \notin FV(N)$, *otherwise* z *is a fresh variable.*

$(\textbf{letrec } x_1 = E_1, ..., x_n = E_n \textbf{ in } E)[x := N]$

$\doteq$ $\textbf{letrec } z_1 = E_1[\vec{x}_i{:=}\vec{z}_i][x := N], ..., z_n = E_n[\vec{x}_i{:=}\vec{z}_i][x := N]$

$\textbf{in } E[\vec{x}_i{:=}\vec{z}_i][x := N]$,

where $z_i \equiv x_i$ *if* $x \notin FV(\textbf{ letrec } x_1 = E_1, ..., x_n = E_n \textbf{ in } E) \vee x_i \notin FV(N)$, *otherwise* z_i *is a fresh variable (i=1..n).*

Definition 8. *The result of substituting* N *for the free occurrences of* x *in* E *without automatic renaming is defined as:*

$x[x := N]_{nr} \doteq N$

$y[x := N]_{nr} \doteq y; \quad y \not\equiv x$

$(E_1E_2)[x := N]_{nr} \doteq E_1[x := N]_{nr}E_2[x := N]_{nr}$

$(\lambda x \ . \ E)[x := N]_{nr} \doteq \lambda x \ . \ E$

$(\lambda y \ . \ E)[x := N]_{nr} \ (y \not\equiv x) \doteq \lambda y \ .E[x := N]_{nr}$,

if $x \notin FV(E) \vee y \notin FV(N)$.

$(\textbf{ letrec } x_1 = E_1, ..., x_n = E_n \textbf{ in } E)[x := N]_{nr}$

$\doteq$ $\textbf{letrec } x_1 = E_1[x := N]_{nr}, ..., z_n = E_n[x := N]_{nr} \textbf{ in } E[x := N]_{nr}$,

if $x \notin FV($ *letrec* $x_1 = E_1, ..., x_n = E_n$ *in* $E) \vee x_{i(i=1..n)} \notin FV(N)$.

Definition 9. *Given* $x \in FV(E)$ *and a context* $C[\ ]$, *we say that* x *is free over* $C[E]$ *if and only if* $\forall e, e \in sub(E, C) \Rightarrow x \in FV(e)$. *Otherwise we say* x *becomes bound over* $C[E]$.

Lemma 1. *Let* E', E *be expressions, and* $E \equiv C[z]$, *where* z *is a free variable in* E *and does not occur free in* $C[\ ]$. *If none of the free variables in* E' *will become bound over* $C[E']$, *then* $E[z := E'] \equiv C[E']$.

Proof. By induction on the structure of E.

Case 1. E is a variable. Then $E \equiv z$, and $C[\ ] \equiv [\ ]$ by the condition.

Therefore $E[z := E'] \doteq E' \equiv C[E']$

Case 2: $E = \lambda x.E_1$. Then $x \not\equiv z$, x should not occur free in E', and $C[\,]$ is of the form $\lambda x.C_1[\,]$ by the condition. We have:

$$\begin{aligned} E[z := E'] &\doteq (\lambda x.E_1)[z := E'] \\ &\doteq \lambda x.(E_1[z := E']) \text{ since x is not free in } E' \\ &\doteq \lambda x.(C_1[E']) \text{ by induction hypothesis} \\ &\equiv C[E'] \end{aligned}$$

Case 3: $E = E_1E_2$.

Case 3.1. z occurs in E_1, then C[ ] is of the form $C_1[\,]E_2$, and $E_1 \equiv C_1[z]$.

$$\begin{aligned} E[z := E'] &\doteq (E_1E_2)[z := E'] \\ &\doteq E_1[z := E']E_2 \text{ since z does not occur in } E_2 \\ &\doteq C_1[E']E_2 \text{ by induction hypothesis} \\ &\equiv C[E'] \end{aligned}$$

Case 3.2. z occurs in E_2. Similar as Case 3.1.

Case 4: $E =$ **letrec** $x_1 = E_1, ..., x_n = E_n$ **in** E_0

Case 4.1. z occurs in E_i, then C[ ] is of the form:

letrec $x_1 = E_1, ..., x_i = C_i[\,], ..., x_n = E_n$ **in** E, and $E_i \equiv C_i[z]$, and we have:

$$\begin{aligned} & E\,[z := E'] \\ &\doteq (\textbf{letrec } x_1 = E_1, ..., x_i = E_i, ..., x_n = E_n \text{ in } E)[z := E'] \\ &\doteq \textbf{letrec } x_1 = E_1, ..., x_i = E_i[z := E'], ..., x_n = E_n \text{ in } E \\ &\doteq \textbf{letrec } x_1 = E_1, ..., x_i = C_i[E'], ..., x_n = E_n \text{ in } E \\ &\equiv C[E'] \end{aligned}$$

Case 4.2. z occurs in E_0. Similar as Case 4.1. □

Lemma 2. *Substitution Lemma: If $x \not\equiv y$, and $x \notin FV(L)$, then*

$E[x := E'][y := L] = E[y := L][x := E'[y := L]]$

Proof. Proof by induction on the structure of E. □

7.3 Specification of *Generalise a Definition*

The following definition specifies *generalise a definition*. A commentary on the definition and discussions about some variations to the given specification follow.

Definition 10. *Given an expression*

$$\textbf{letrec } x_1 = E_1, ..., x_i = E_i, ..., x_n = E_n \textbf{ in } E_0$$

Assume E is a sub-expression of E_i, and $E_i \equiv C[E]$. Then the condition for generalising the definition $x_i = E_i$ on E is:

$$x_i \notin FV(E) \wedge \forall x, e : (x \in FV(E) \wedge e \in sub(E_i, C) \Rightarrow x \in FV(e)).$$

After generalisation, the original expression becomes:

$$\begin{aligned}
\textbf{letrec } x_1 &= E_1[x_i := x_i E], \\
&\dots, \\
x_i &= \lambda z.(C[z][x_i := x_i z]), \\
&\dots, \\
x_n &= E_n[x_i := x_i E] \\
\textbf{in } E_0[x_i := x_i E],& \text{ where } z \text{ is a fresh variable}
\end{aligned}$$

What follows provides some explanation of the conditions in the definition above:

- The condition $x_i \notin FV(E)$ means that there should be no recursive calls to x_i within the identified sub-expression E. This is necessary in the case that the generalised definition $x_i = E_i$ is a directly recursive function. For instance, in the expression shown in Figure 21 (the syntax of this example is not supported by λ_{Letrec}), generalising the definition **f** on the sub-expression

f 10 will result in an expression in which the **f** in the sub-expression **f 10** has wrong number of arguments. While allowing recursive calls in the identified expression is possible but would need extra care to make sure that the generalised function has the correct number of parameters at its call-sites.

— The expression before generalising the definition of f on sub-expression $f10$.

$$\begin{array}{l}\textbf{let}\ f\ x\ =\ \textbf{if}\ x\ =\ 0\ \textbf{then}\ 1\\ \qquad\qquad\qquad\qquad\ \textbf{else}\ f\ (x-1)\ +\ f\ 10\\ \textbf{in}\ f\ 17\end{array}$$

— The expression after generalisation.

$$\begin{array}{l}\textbf{let}\ f\ y\ x\ =\ \textbf{if}\ x\ =\ 0\ \textbf{then}\ 1\\ \qquad\qquad\qquad\qquad\ \textbf{else}\ f\ y\ (x-1)\ +\ y\\ \textbf{in}\ f\ (f\ 10)\ 17\end{array}$$

Figure 21: Generalisation on a directly recursive definition

- The condition $\forall x, e : (x \in FV(E) \wedge e \in sub(E_i, C) \Rightarrow x \in FV(e))$ ensures that the none of the free variables in E is locally declared in the definition $x_i = E_i$.

Discussion. Some variations to the above specification are discussed next. These variations also reflect the general observation that under the same refactoring name, different people may mean different things, and there is no unique way of resolving this choice.

- The specification given in Section 7.3 allows automatic renaming, hence some bound variables might be renamed to avoid name capture during the substitution phase. Without automatic renaming, the refactoring would fail if any of the substitution fails. For the refactoring to succeed without automatic renaming, the following side-condition needs to be added:
  $\forall e, j : (E_j \equiv C[e] \wedge x_i \text{ is free over } C[e] \Rightarrow TBV(e) \cap FV(x_i E) = \phi)$,

which means that none of the free variables in the expression x_iE should be captured when x_iE is substituted for x_i at the call-sites of x_i.

- The specification given replaces only the identified occurrence of E in the definition $x_i = E_i$ by the formal parameter z. Another variant is to replace all the occurrences of E in $x_i = E_i$ by z. This does not change the side-conditions for the refactoring, but it does change the transformation within the definition $x_i = E_i$. With all the occurrences of E being replaced in $x_i = E_i$, the resulting program would be:

$$\begin{aligned}
\textbf{letrec } x_1 &= E_1[x_i := x_iE], \\
&\ldots, \\
x_i &= \lambda z.(M[z][x_i := x_iz]), \\
&\ldots, \\
x_n &= E_n[x_i := x_iE] \\
\textbf{in } E_0[x_i := x_iE], &\text{ where z is a fresh variable}
\end{aligned}$$

  where $M[\ ]$ is the resulting context by replacing each occurrence of E in E with a hole, and all the replaced occurrences of E should be syntactically equivalent (modulo α-renaming of bound variables) and semantically equivalent.

- According to the specification given, this refactoring could introduce duplicated computation. One way to avoid duplicating the computation of x_iE is to introduce a new binding to represent the expression, instead of duplicating it at each call-site of x_i. Then after the generalisation, we have:

$$\begin{aligned}
\textbf{letrec } x_1 &= E_1[x_i := x_i'] \\
&\ldots \\
x_i &= \lambda z.C[z][x_i := x_iz] \\
x_i' &= x_iE
\end{aligned}$$

$$
\begin{aligned}
&\cdots \\
&x_n = E_n[x_i := x_i'] \\
\textbf{in} \quad & E[x_i := x_i'], \text{ where z, } x_i' \text{ are fresh variables.}
\end{aligned}
$$

7.4 Verification of *Generalise a Definition*

In order to prove that this refactoring is behaviour-preserving, we decompose the transformation into a number of steps. If each step is behaviour-preserving, then we can conclude that the whole transformation is behaviour-preserving.

Proof. Given the original expression:

$$
\begin{aligned}
&\textbf{letrec } x_1 = E_1, \\
&\cdots, \\
&x_i = E_i, \\
&\cdots, \\
&x_n = E_n \\
&\textbf{in } E
\end{aligned}
$$

Generalising the definition $x_i = E_i$ on the sub-expression E can be decomposed into the following steps:

Step 1. add definition $x_i' = \lambda z.C[z]$, where x_i' and z are fresh variables, and $C[E] \equiv E_i$, we get

$$
\begin{aligned}
&\textbf{letrec } x_1 = E_1, \\
&\cdots, \\
&x_i = E_i, \\
&x_i' = \lambda z.C[z], \\
&\cdots,
\end{aligned}
$$

$$x_n = E_n$$
$$\textbf{in } E$$

This step does not change the semantics as x'_i is not used. Formally, the equivalence of semantics is guaranteed by the garbage collection axiom and the commutability of bindings within **letrec**.

Step 2. By the side-conditions and axioms, in the context of the definition of x'_i, we can prove

$$\begin{aligned}
x'_i E &\equiv (\lambda z.C[z])E \\
&\doteq \textbf{letrec } z = E \textbf{ in } C[z] \qquad \text{by } \beta\circ \\
&\doteq \textbf{letrec } z = E \textbf{ in } C[E] \qquad \text{by } \textit{substitution} \text{ axiom and side-conditions} \\
&\doteq C[E] \qquad \text{by } \textit{garbage collection} \text{ axioms} \\
&\equiv E_i
\end{aligned}$$

Therefore replace E_i with $x'_i E$ in the context of the definition does not change its semantics, and the original expression is equivalent to:

$$\begin{aligned}
&\textbf{letrec } x_1 = E_1, \\
&\quad \ldots, \\
&x_i = x'_i E, \\
&x'_i = \lambda z.C[z], \\
&\quad \ldots, \\
&x_n = E_n \\
&\textbf{in } E
\end{aligned}$$

Step 3. Using the second *substitution axiom*, it is trivial to prove that substituting $x'_i E'_i$ for the free occurrences of x_i in the right-hand-side of x'_i does not change the

semantics of x'_i. We get

$$\begin{aligned}
\textbf{letrec } & x_1 = E_1, \\
& \ldots, \\
& x_i = x'_i E, \\
& x'_i = (\lambda z.C[z])[x_i := x'_i E], \\
& \ldots, \\
& x_n = E_n \\
\textbf{in } & E
\end{aligned}$$

As $z \notin FV(x'_i E]$, we have:

$$\begin{aligned}
\textbf{letrec } & x_1 = E_1, \\
& \ldots, \\
& x_i = x'_i E, \\
& x'_i = \lambda z.C[z][x_i := x'_i E], \\
& \ldots, \\
& x_n = E_n \\
\textbf{in } & E
\end{aligned}$$

Step 4. In the definition of x'_i, replace E with z. we get:

$$\begin{aligned}
\textbf{letrec } & x_1 = E_1, \\
& \ldots, \\
& x_i = x'_i E, \\
& x'_i = \lambda z.C[z][x_i := x'_i z], \\
& \ldots,
\end{aligned}$$

$$x_n = E_n$$
$$\textbf{in} \quad E$$

It is evident that the right-hand side (RHS) of the definition of x'_i defined in this step is not semantically equal to the RHS defined in step 3. However we can prove the equivalent of $x'_i E$ from step 3 to step 4 in the context of the bindings for $x_1, ..., x_n$ (note that x'_i does not depend on the definition of x_i, so there is no mutual dependency between x_i and x'_i). Let's use $x'_{i(E)}$ and $x'_{i(z)}$ to represent the x'_is defined in step 3 and 4 respectively. Then

$$\begin{aligned}
x'_{i(z)}E &\doteq (\lambda z.(C[z][x_i := x'_{i(z)}z]))E \\
&\doteq \textbf{letrec} \;\; z = E \;\textbf{in}\; C[z][x_i := x'_{i(z)}z] \qquad \text{by } \beta\circ \\
&\doteq \textbf{letrec} \;\; z = E \;\textbf{in}\; C[z][x_i := x'_{i(z)}z][z := E] \qquad \text{by } \textit{substitution} \\
&\doteq C[z][x_i := x'_{i(z)}z][z := E] \qquad \text{by } \textit{garbage collection} \\
&\doteq C[z][z := E][x_i := x'_{i(z)}z[z := E]] \qquad \text{by the substitution lemma} \\
&\doteq C[E][x_i := x'_{i(z)}E] \qquad \text{by lemma 1} \\
&\doteq E_i[x_i := x'_{i(z)}E]
\end{aligned}$$

In a similar way, we can derive: $x'_{i(E)}E \doteq E_i[x_i := x'_{i(E)}E]$. The equivalent of $x'_{i(E)}E$ between $x'_{i(z)}E$ can be proved using scoped lambda-graphs proposed by Z. M. Ariola and S. Blom in *Lambda Calculi plus Letrec* [120], as they correspond to the same scoped lambda-graph. As $x_i \doteq x'_{i(E)}E$, we have $x_i \doteq x'_{i(z)}E$.

Step 5. Substituting $x'_i E$ for the free occurrences of x_i outside the definition of x_i and x'_i does not change the semantics of the let-expression, as $x_i \doteq x'_{i(z)}E$ from step 4.

$$\begin{aligned}
\textbf{letrec}\; x_1 &= E_1[x_i := x'_i E], \\
&\ldots, \\
x'_i &= \lambda z.C[z][x_i := x'_i z],
\end{aligned}$$

$$
\begin{aligned}
&\ldots,\\
&x_n = E_n[x_i := x_i'E]\\
\textbf{in}\quad & E[x_i := x_i'E]
\end{aligned}
$$

Step 6. Remove the definition of x_i, we get

$$
\begin{aligned}
\textbf{letrec}\ x_1 &= E_1[x_i := x_i'E],\\
&\ldots,\\
x_i' &= \lambda z.C[z][x_i := x_i'z],\\
&\ldots,\\
x_n &= E_n[x_i := x_i'E]\\
\textbf{in}\quad & E[x_i := x_i'E]
\end{aligned}
$$

This does not change the semantics because of the *garbage collection* axiom.

Step 7. Renaming x_i' to x_i, we have

$$
\begin{aligned}
\textbf{letrec}\ x_1 &= E_1[x_i := x_i'E][x_i' := x_i],\\
&\ldots,\\
x_i &= \lambda z.C[z][x_i := x_i'z][x_i' := x_i],\\
&\ldots,\\
x_n &= E_n[x_i := x_i'E][x_i' := x_i]\\
\textbf{in}\quad & E[x_i := x_i'E][x_i' := x_i]
\end{aligned}
$$

Capture-free renaming of bound variables, i.e. $\alpha\_renaming$, does not change the semantics. Finally, by the substitution lemma, we have

$$
\begin{aligned}
\textbf{letrec}\ x_1 &= E_1[x_i := x_iE],\\
&\ldots,\\
x_i &= \lambda z.C[z][x_i := x_iz],
\end{aligned}
$$

$$
\begin{aligned}
&\ldots, \\
&x_n = E_n[x_i := x_i E] \\
\textbf{in} \quad & E[x_i := x_i E]
\end{aligned}
$$

□

Some of the steps used in this proof, such as step 1 (*adding a new definition*), step 2 (*remove an unused definition*), step 3 (*unfold a definition*), and step 7 (*rename a function name*), are elementary refactorings on their own, while the others are not. This raises the question of whether *generalise a definition* should be treated as a composite refactoring, or more generally, whether there is a clear distinction between elementary refactorings and composite refactorings. In this thesis, we distinguish elementary and composite refactorings from the implementation point of view. If the implementation of a refactoring is based on the implementation of other refactorings, as is the case for *from concrete to abstract data type*, then we regard this refactoring as a composite refactoring, otherwise, we say that the refactoring is elementary. While elementary refactorings are used in the above proof, they are not used in the implementation for efficiency reason, therefore, *generalise a definition* is treated as an elementary refactoring in this thesis.

7.5 λ_{Letrec} Extended With a Module System

A module-aware refactoring normally affects not only the definitions in a module, but also the imports and exports of the module. More than that, it may potentially affect every module in the system. A typical module-aware refactoring is *move a definition from one module to another*. This refactoring moves an identified declaration from its current module to a specified target module, as shown in the example in Figure 22, where the definition of *foo* is moved from module $M1$ to $M2$. Together with the move of *foo*'s definition is the modification to the imports/exports of the affected modules, which compensates for the changes caused

by moving the definition.

```
module M1(foo, sq) where

sq x = x^ pow
  where pow = 2

foo x y = sq x + sq y

module M2 where
import M1(sq)

bar x y = sq(x + y)

module Main where
import M1
import M2(bar)

main
  = print $ foo 10 20 + bar 30 40
```

```
module M1(sq) where

sq x = x^ pow
  where pow = 2

module M2 where
import M1(sq)

foo x y = sq x + sq y

bar x y = sq(x + y)

module Main where
import M1
import M2(bar, foo)

main
  = print $ foo 10 20 + bar 30 40
```

Figure 22: Move the definition of *foo* to module *M2*

In order to describe module-aware refactorings, we extend λ_{Letrec} with a module system. The definition of the resulted new language, which is called $\lambda_{\mathbf{M}}$, is given next'.

The syntax of λ_M terms is defined as:

$$Program ::= \textbf{let } Mod \textbf{ in } (Exp; Imp;\ \textbf{letrec } D \textbf{ in } E)$$

$$Mod ::= \varepsilon \mid Modid = (Exp; Imp; D) \mid Mod; Mod$$

$$Exp ::= \varepsilon \mid (Ep_1, ..., Ep_n)\ \ (n \geq 0)$$

$$Ep = x \mid Modid.x \mid\ \textbf{module } Modid$$

$$Imp ::= (Ip_1, ..., Ip_n)\ \ (n \geq 0)$$

$$Ip =\ \textbf{import } Qual\ Modid\ Alias\ ImpSpec$$

$$Modid ::= M_i\ \ (i \geq 0)$$

$$Qual ::= \varepsilon \mid \textbf{qualified}$$

$$
\begin{aligned}
&ImpSpec ::= \varepsilon \mid (x_1, ..., x_n) \mid \ \textbf{hiding}\ (x_1, ..., x_n)\ \ (n \geq 0)\\
&Alias ::= \varepsilon \mid \ \textbf{as}\ Modid\\
&E ::= x\\
&\quad \mid \lambda x.E\\
&\quad \mid Modid.x\\
&\quad \mid E_1\ E_2\\
&\quad \mid \textbf{letrec}\ D\ \textbf{in}\ E\\
&D ::= \varepsilon \mid x = E \mid D, D
\end{aligned}
$$

In the above definition, $Program$ represents a program and Mod is a sequence of modules. Each module has a unique name in the program. A module consists of three parts: Exp, which exports some of the locally available identifiers for use by other modules; Imp, which imports identifiers defined in other modules; and D, which defines a number of value identifiers. The (Exp; Imp; **letrec** D **in** E) part of the definition of $Program$ represents the $Main$ module of the program, and the expression E represents the *main* expression. ε means a null export list in the definition of Exp, a null entity list in the definition of $ImpSpec$, and *empty* in other definitions. Qualified names are allowed, and we assume that the usage of qualified names follows the rules specified in the Haskell 98 Report [53].

The module system has been defined to model aspects of the Haskell 98 module system. Because only value variables can be defined in λ_M, λ_M's module system is actually a subset of the Haskell 98 module system. We assume that the semantics of this module system follows the semantics of the Haskell 98 module system.

A formal specification of the Haskell 98 module system has been described in the paper *A Formal Specification for the Haskell 98 Module System* [28], where the semantics of a Haskell program with regard to the module system is a mapping from the collection of modules to their corresponding *in-scope* and *export* relations. The *in-scope* relation of a module represents the set of names (with the represented entities) that are visible within this module, and this forms the

top-level environment of the module. The *export* relation of a module represents the set of names (also with the represented entities) that are made available by this module for other modules to use; in other words, it defines the interface of the module. Although the term "relation" is used in the specification, a name should only refer to one entity in a valid Haskell program.

In the following specification of module-aware refactorings, we assume that, using the module system analysis algorithm from the formal specification given in [28], we are able to get the *in-scope* and *export* relations of each module, and for each identifier in the *in-scope/export* relation, we can infer the name of the module in which the identifier is defined. In fact, the same module analysis system is used in the implementation of HaRe.

When only module-level information is relevant, i.e., the exact definitions of entities is not of concern, we can view a multi-module program in this way: a program P consists of a set of modules and each module consists of four parts: the module name, M, the set of identifiers defined by this module, D, the set of identifiers imported by this module, I, and the set of identifiers exported by this module, E. Each top-level identifier can be uniquely identified by the combination of the identifier's name and its defining module as *(modid, id)*, where *modid* is the name of the identifier's defining module and *id* is the name of the identifier. Two identifiers are the same if they have the same name and defining module. Accordingly, we can use $P = \{(M_i, D_i, I_i, E_i)_{i=1..n}$ to denote the program.

7.6 Fundamentals of λ_M

Some definitions related to λ_M (mainly the module system part) are introduced in the section. These definitions, together with the definitions given in Section 7.2, serve as the basis for the specification and verification of module-aware refactorings.

Definition 11. *A* client module *of module M is a module which imports M either directly or indirectly.*

Definition 12. *A* server module *of module M is a module which is imported by module M either directly or indirectly.*

Definition 13. *Given a module M=(Exp, Imp, D), we say module M is exported by itself if Exp is* ε *or* module M *occurs in Exp.*

Definition 14. *The* defining module *of an identifier is the name of the module in which the identifier is defined.*

Definition 15. *Suppose v is an identifier that is in scope in module M, we use* defineMod(v, M) *to represent the name of the module in which the identifier is defined.*

Definition 16. *TBV(D) is the set of top-level identifiers declared in D (a sequence of declarations) and can be defined as:*

$TBV(\varepsilon)=\{\ \}$

$TBV(x = E) = \{\ x\ \}$

$TBV\ (D_1, D_2)=TBV(D_1) \cup TBV(D_2)$

Definition 17. *FV(D) is the set of free variables in D (a sequence of declarations), and can be defined as:*

$FV(\varepsilon) = \{\}$

$FV(x = E) = FV(E) - \{x\}$

$FV(D_1, D_2) = FV(D_1) \cup FV(D_2) - TBV(D_1, D_2)$

Definition 18. Binding structure *refers to the association of uses of identifiers with their definitions in a program. Binding structure involves both top-level variables and local variables. When analysing module-level phenomena, it is only the top-level bindings that are relevant, in which case we define the binding structure, B, of a program* $P = \{(M_i, D_i, I_i, E_i)\}_{i=1..n}$ *as* $B \subset \cup(D_i \times (D_i \cup I_i))_{i=1..n}$*, so that* $\{((m_1, id_1), (m_2, id_2)) \in B \mid id_2$ *occurs free in the definition of* id_1*; the defining module of* id_1 *is* m_1*, and the defining module of* id_2 *is* $m_2\ \}$*.*

Definition 19. *We say that the identifier x defined in module N is used by module M=(Exp, Imp, D) (*$M \neq N$*) if DefineMod(x,M) =* N *and either* $x \in FV(D)$ *or x*

is exported by module M, otherwise we say that the x defined in module N is not used by module M.

The following definitions, which modify the export/import list of a module, define the most commonly used operations to the module interface when module-aware refactorings are implemented.

Definition 20. *Given a set of identifiers Y and an export list Exp, rmFromExp(Exp,Y) is the export list Exp with the occurrences of the identifiers from Y removed.*

$rmFromExp\ (\varepsilon, Y) = \varepsilon$

$rmFromExp\ ((), Y) = ()$

$rmFromExp\ ((e, Ep_2, ..., Ep_n), Y)\ \ (e \notin Y)$

$\quad = (e, rmFromExp(Ep_2, ..., Ep_n), Y)$

$rmFromExp\ ((e, Ep_2, ..., Ep_n), Y)\ \ (e \in Y)$

$\quad = rmFromExp\ ((Ep_2, ..., Ep_n), Y)$

Definition 21. *Given an identifier y which is defined in module M, and the export list, Exp, of module M, addToExp (Exp, y M) is the export list with y added if it is not already exported by Exp, and can be defined as:*

$addToExp\ (\varepsilon, y, M) = \varepsilon$

$addToExp\ (( ), y, M) = (y)$

$addToExp\ ((Ep_1, ..., Ep_n), y, M)$

$\quad = (Ep_1, ..., Ep_n)\ \ if\ \exists i,\ y \equiv Ep_i\ \ (1 \leq i \leq n);$

$\qquad otherwise\ (Ep_1, ..., Ep_n, y)$

In the following two definitions, qualifiers and alias used in import declarations are not affected, so we omit them from the definitions for simplicity reason.

Definition 22. *Given an identifier y which is exported by module M, and Imp which is a sequence of imports, rmFromImp (Imp, y, M) is the import sequence Imp with the occurrences of y removed from the import declarations that import M. The function can be used to cleanup the uses of y in import declarations that*

import module M when y is no longer exported by M. rmFromImp (Imp, y, M) is defined as:

$rmFromImp\ (( ), y, M) = ( )$

$rmFromImp\ ((\text{import } N, Ip_1, Ip_2, ..., Ip_n), y, M)$

$= (\text{import } N, rmFromImp\ ((Ip_2, ..., Ip_n), y, M))$

$rmFromImp\ ((\text{import } N(x_1, ..., x_i, ..., x_n), Ip_2, ..., Ip_n), y, M)$

$= if\ N \equiv M \wedge x_i \equiv y$

$then\ (\text{import } N(x_1, ..., x_{i-1}, x_{i+1}, ..., x_n),$

$rmFromImp\ ((Ip_2, ..., Ip_n), y, M))$

$else\ (\text{import } N(x_1, ..., x_i, ..., x_n),$

$rmFromImp\ ((Ip_2, ..., Ip_n), y, M))$

$rmFromImp\ ((\text{import } N \text{ hiding } (x_1, ..., x_i, ..., x_n)), Ip_2, ..., Ip_n), y, M)$

$= if\ N \equiv M \wedge x_i \equiv y$

$then\ (\text{import } N \text{ hiding } (x_1, ..., x_{i-1}, x_{i+1}, ..., x_n),$

$rmFromImp\ ((Ip_2, ..., Ip_n), y, M))$

$else\ (\text{import } N \text{ hiding } (x_1, ..., x_i, ..., x_n),$

$rmFromImp\ ((Ip_2, ..., Ip_n), y, M))$

Definition 23. *Given an identifier y which is exported by module M (M is not necessarily the module where y is defined) and Imp which is a sequence of imports, then hideInImp(Imp, y, M) is the import sequence Imp with y removed from the explicit entity list or added to the explicit hiding entity list in the import declarations which import module M, so that the resulting Imp does not bring this identifier into scope by importing it from module M. hideInImp(Imp, y, M) is defined as:*

$hideInImp\ (( ), y, M) = ( )$

$hideInImp\ ((\text{import } N, Ip_2, ..., Ip_n), y, M)$

$= if\ (N \equiv M)$

$then\ (\text{import } N \text{ hiding } (y), hideInImp\ ((Ip_2, ..., Ip_n), y, M))$

$else\ (\text{import } N, hideInImp\ ((Ip_2, ..., Ip_n), y, M))$

$hideInImp\ ((\text{ import } N(x_1, ..., x_i, ..., x_n), Ip_2, ..., Ip_n), y, M)$

$= if\ (N \equiv M, x_i \equiv y)$

$then$ (import $N(x_1, ..., x_{i-1}, x_{i+1}, ..., x_n)$,

$hideInImp\ ((Ip_2, ..., Ip_n), y, M)$)

$else$ (import $N(x_1, ..., x_n), hideInImp\ ((Ip_2, ..., Ip_n), y, M)$)

$hideInImp$ ((import N hiding $(x_0, ..., x_n), Ip_2, ..., Ip_n), y, M)$

$= if\ (N \equiv M \wedge y \notin \{x_1, ..., x_n\})$

$then$ (import N hiding $(x_0, ..., x_n, y), hideInImp\ ((Ip_2, ..., Ip_n), y, M)$)

$else$ (import N hiding $(x_0, ..., x_n), hideInImp\ ((Ip_2, ..., Ip_n), y, M)$)

Definition 24. *Suppose the same binding, say y, is exported by both module M_1 and M_2, and Imp is a sequence of import declarations, then chgImpPath(Imp, y, M_1, M_2) is the import sequence Imp with the importing of y from M_1 changed to M_2, and is defined as:*

$chgImpPath\ (( ), y, M_1, M_2) = ( )$

$chgImpPath$ ((import $Qual\ N$ as $Modid\ \varepsilon), Ip_2, ..., Ip_n), y, M_1, M_2)$

$= if\ (N \equiv M_1)$

$then$ (import $Qual\ N$ as $Modid$ hiding (y), import $Qual\ M_2$ as $Modid\ (y)$,

$chgImpPath\ ((Ip_2, ..., Ip_n), y, M_1, M_2))$

$else$ (import $Qual\ N$ as $Modid\ \varepsilon$,

$chgImpPath\ ((Ip_2, ..., Ip_n), y, M_1, M_2))$

$chgImpPath$ ((import $Qual\ N\ \varepsilon\ \varepsilon), Ip_2, ..., Ip_n), y, M_1, M_2)$

$= if\ (N \equiv M_1)$

$then$ (import $Qual\ N\ \varepsilon$ hiding (y), import $Qual\ M_2$ as $N\ (y)$,

$chgImpPath\ ((Ip_2, ..., Ip_n), y, M_1, M_2))$

$else$ (import $Qual\ N$ as $Modid\ \varepsilon$,

$chgImpPath\ ((Ip_2, ..., Ip_n), y, M_1, M_2))$

$chgImpPath$ ((import $Qual\ N$ as $Modid\ (x_1, ..., x_i, ..., x_n), Ip_2, ..., Ip_n), y, M_1, M_2)$

$= if\ (N \equiv M_1 \wedge x_i \equiv y)$

$then$ (import $Qual\ N$ as $Modid\ (x_1, ..., x_{i-1}, x_{i+1}, ..., x_n)$,

import $Qual\ M_2$ as $Modid\ (y), chgImpPath((Ip_2, ..., Ip_n), y, M_1, M_2))$

$else$ (import $Qual\ N$ as $Modid\ (x_1, ..., x_i, ..., x_n)$,

$chgImpPath((Ip_2, ..., Ip_n), y, M_1, M_2))$

$chgImpPath\ ((\text{import}\ \ Qual\ N\ \varepsilon\ (x_1, ..., x_i, ..., x_n), Ip_2, ..., Ip_n), y, M_1, M_2)$

$= if\ (N \equiv M_1 \wedge x_i \equiv y)$

$then\ (\text{import}\ \ Qual\ N\ \varepsilon\ (x_1, ..., x_{i-1}, x_{i+1}, ..., x_n),$

$\text{import}\ Qual\ M_2\ \ \text{as}\ N\ (y), chgImpPath((Ip_2, ..., Ip_n), y, M_1, M_2))$

$else\ (\text{import}\ \ Qual\ N\ \varepsilon\ (x_1, ..., x_i, ..., x_n),$

$chgImpPath((Ip_2, ..., Ip_n), y, M_1, M_2))$

$chgImpPath\ ((\text{import}\ \ Qual\ N\ \ \text{as}\ Modid\ \text{hiding}\ (x_1, ..., x_n), Ip_2, ..., Ip_n), y, M_1, M_2)$

$= if\ (N \equiv M_1 \wedge y \notin \{x_1, ..., x_n\})$

$then\ (\text{import}\ \ Qual\ N\ \ \text{as}\ Modid\ \text{hiding}\ (x_1, ..., x_n, y),$

$\text{import}\ Qual\ M_2\ \ \text{as}\ Modid\ (y), chgImpPath((Ip_2, ..., Ip_n), y, M_1, M_2))$

$else\ (\text{import}\ \ Qual\ N\ \ \text{as}\ Modid\ \text{hiding}\ (x_1, ..., x_n),$

$chgImpPath\ ((Ip_2, ..., Ip_n), y, M_1, M_2))$

$chgImpPath\ ((\text{import}\ \ Qual\ N\ \varepsilon\ \text{hiding}\ (x_1, ..., x_n), Ip_2, ..., Ip_n), y, M_1, M_2)$

$= if\ (N \equiv M_1 \wedge y \notin \{x_1, ..., x_n\})$

$then\ (\text{import}\ \ Qual\ N\ \varepsilon\ \text{hiding}\ (x_1, ..., x_n, y),$

$\text{import}\ Qual\ M_2\ \text{as}\ N\ (y), chgImpPath((Ip_2, ..., Ip_n), y, M_1, M_2))$

$else\ (\text{import}\ \ Qual\ N\ \varepsilon\ \text{hiding}\ (x_1, ..., x_n),$

$chgImpPath\ ((Ip_2, ..., Ip_n), y, M_1, M_2))$

7.7 Specification of *Move a Definition from One Module to Another*

Move a definition from one module to another is a non-trivial refactoring, and the realisation of this refactoring is also non-unique. Suppose we would like to move the definition of *foo* from module M to module N, the design decisions we made during the implementation of HaRe are:

- If a variable occurs free in the definition of *foo*, but is not in scope in module N, then the refactorer will ask the user to make this variable visible within module N first.

- If the identifier *foo* is already in scope in module N (either defined by module N or imported from other modules), but it refers to another *foo* other than the one defined in module M, the user will be prompted to do renaming first.

- Mutually recursive modules should not be introduced during the refactoring. Although mutually recursive modules are allowed in Haskell 98, transparent compilation of mutually recursive modules are not yet supported by the current working Haskell compilers/interpreters. Therefore, we try to avoid introducing mutually recursive modules during refactoring.

- If module M exports *foo* before the refactoring, then it still exports *foo* after the refactoring as long as doing this does not introduce recursive modules; If module M does not export *foo* before the refactoring, then it does not export *foo* after the refactoring either.

- Module N will export *foo* after the refactoring only if *foo* is either exported by module M or used by the other definitions in module M before the refactoring.

- The importing of *foo* will be via M if module M still exports *foo* after the refactoring; otherwise via N.

The following definition specifies *move a definition from one module to another*. A commentary on the definition follows, and it may be helpful to read this in conjunction with the specification.

Definition 25. *Given a valid program P:*

$$\begin{aligned} P = \textbf{let } & M_1 = (Exp_1;\ Imp_1;\ x_1 = E_1, ..., x_i = E_i, ..., x_n = E_n); \\ & M_2 = (Exp_2;\ Imp_2;\ D_2); \\ & \quad \ldots; \\ & M_m = (Exp_m;\ Imp_m;\ D_m) \\ \textbf{in } & (Exp_0;\ Imp_0;\ \textbf{letrec } D_0 \textbf{ in } E) \end{aligned}$$

The conditions for moving the definition $x_i = E_i$ from module M_1 to another module, M_2 say, are:

a) If x_i is in scope at the top level of M_2, then DefineMod(x_i, M_2) = M_1.

b) $\forall\ v \in FV(x_i = E_i)$, if DefineMod(v, M_1)=N,

then v is in scope in M_2 and DefineMod(v, M_2)=N.

c) If M_1 is a server module of M_2, then $\{x_i, M_1.x_i\} \cap FV(E_{j(j\neq i)}) = \emptyset$.

d) If module $M_{j(j\neq 1)}$ is a server module of M_2, and $x_i \in FV(D_j)$,

then $DefineMod(x_i, M_j) \neq M_1$ (x_i could be qualified or not).

To make the specification clear, the program after the refactoring, P', is given by two cases according to whether x_i is exported by M_1. In each case, different situations are considered.

Case 1. x_i is not exported by M_1.

Case 1.1. x_i is not used by other definitions in M_1,

that is, $\{x_i, M_1.x_i\} \cap FV(E_{j(j\neq i)}) = \emptyset$

$$
\begin{aligned}
P' = \textbf{let}\quad & M_1 = (Exp_1;\ Imp_1;\ x_1 = E_1, ..., x_{i-1} = E_{i-1}, x_{i+1} = E_{i+1}, ..., x_n = E_n);\\
& M_2 = (Exp_2;\ Imp_2;\ x_i = E_i[M_1.x_i := M_2.x_i], D_2);\\
& \quad \ldots;\\
& M_m = (Exp_m;\ Imp'_m;\ D_m)\\
\textbf{in}\quad & (Exp_0;\ Imp'_0;\ \textbf{letrec}\ D_0\ \textbf{in}\ E)
\end{aligned}
$$

where

$$
\begin{aligned}
Imp'_j = & \ hideInImp\ (Imp_j, x_i, M_2)\ if\ M_2\ is\ exported\ by\ itself;\\
& \ Imp_j\ \ otherwise.\ \ (3 \leq j \leq m\ \ or\ \ j = 0)
\end{aligned}
$$

Case 1.2. x_i is used by other definitions in M_1.

$$
\begin{aligned}
P' = \textbf{let}\quad & M_1 = (Exp_1;\ Imp'_1;\ x_1 = E_1, ..., x_{i-1} = E_{i-1}, x_{i+1} = E_{i+1}, ..., x_n = E_n);\\
& M_2 = (Exp'_2;\ Imp_2;\ x_i = E_i[M_1.x_i := M_2.x_i], D_2);\\
& \quad \ldots;
\end{aligned}
$$

$$
\begin{array}{ll}
 & M_m = (Exp_m;\ Imp'_m;\ D_m) \\
\textbf{in} & (Exp_0;\ Imp'_0;\ \textbf{letrec}\ D_0\ \textbf{in}\ E)
\end{array}
$$

where

$$
\begin{aligned}
Imp'_1 &= hideInImp\ (Imp_1, x_i, M_2); \textbf{import}\ M_2\ \textbf{as}\ M_1(x_i) \\
Exp'_2 &= addToExp\ (Exp_2, x_i, M_2) \\
Imp'_j &= hideInImp\ (Imp_j, x_i, M_2)\ \ (3 \leq j \leq m\ \ or\ \ j = 0)
\end{aligned}
$$

Case 2. x_i is exported by M_1.

Case 2.1. M_2 is not a client module of M_1.

$$
\begin{array}{ll}
P' = \textbf{let} & M_1 = (Exp_1;\ Imp'_1;\ x_1 = E_1, ..., x_{i-1} = E_{i-1}, x_{i+1} = E_{i+1}, ..., x_n = E_n); \\
 & M_2 = (Exp'_2;\ Imp_2;\ x_i = E_i[M_1.x_i := M_2.x_i], D_2); \\
 & \quad \ldots; \\
 & M_m = (Exp_m;\ Imp'_m;\ D_m) \\
\textbf{in} & (Exp_0;\ Imp'_0;\ \textbf{letrec}\ D_0\ \textbf{in}\ E)
\end{array}
$$

where

$$
\begin{aligned}
Imp'_1 &= Imp1;\ \textbf{import}\ M_2\ \textbf{as}\ M_1(x_i) \\
Exp'_2 &= addToExp\ (Exp_2, x_i, M_2) \\
Imp'_j &= hideInImp\ (Imp_j, x_i, M_2)\ \ (3 \leq j \leq m\ \ or\ \ j = 0)
\end{aligned}
$$

Case 2.2. M_2 is a client module of M_1.

$$
\begin{array}{ll}
P' = \textbf{let} & M_1 = (Exp'_1;\ Imp_1;\ x_1 = E_1, ..., x_{i-1} = E_{i-1}, x_{i+1} = E_{i+1}, ..., x_n = E_n); \\
 & M_2 = (Exp'_2;\ Imp'_2;\ x_i = E_i[M_1.x_i := M_2.x_i], D_2); \\
 & \quad \ldots; \\
 & M_m = (Exp_m;\ Imp'_m;\ D_m) \\
\textbf{in} & (Exp_0;\ Imp'_0;\ \textbf{letrec}\ D_0\ \textbf{in}\ E)
\end{array}
$$

where

$$
\begin{aligned}
Exp_1' &= rmFromExp\ (Exp_1, x_i, M_1) \\
Exp_2' &= addToExp\ (Exp_2, x_i, M_2) \\
Imp_2' &= rmFromImp\ (Imp_2, x_i, M_1) \\
Imp_j' &= if\ M_j\ is\ a\ server\ module\ of\ M_2 \\
&\qquad then\ \ rmFromImp\ (Imp_j, x_i, M_1) \\
&\qquad else\ \ rmFromImp\ (chgImportPath\ (Imp_j'', x_i, M_1, M_2), x_i, M_1) \\
&\qquad (3 \leq j \leq m\ \ or\ \ j = 0) \\
Imp_j'' &= if\ x_i\ is\ exported\ by\ M_2\ before\ refactoring,\ then\ Imp_j; \\
&\qquad hideInImp\ (Imp_j, x_i, M_2)\ otherwise. \quad (3 \leq j \leq m\ \ or\ \ j = 0)
\end{aligned}
$$

What follows provides some explanation of the above definition:

- As to the side-conditions, condition a) means that if x_i is in scope in the target module, M_2, then this x_i should be the same as the x_i whose definition is to be moved. This condition aims to avoid causing name conflict/ambiguity in M_2; condition b) requires that all the free variables used in the definition of x_i are in scope in M_2. An entity can be brought into scope by either refactoring the exports/imports of the involved modules, or using the *move a definition from one module to another* refactoring to move the definition into scope. While it is possible for this refactoring itself to bring those variables into scope, doing this will make its definition and implementation more complicated, therefore we chose to divide the functionality into more elementary ones; finally, conditions c) and d) together guarantee that mutual recursive modules won't be introduced during the refactoring process.

- The design of transformation rules was made complicated mainly by two reasons.

  First, it is not clear whether M_1 should still export x_i after the refactoring, if it does before the refactoring. After having examined a number of examples with the original module, M_1, and the target module, M_2, having different

relationships in the module graph, we concluded that the answer to whether M_1 should still export x_i depends on the concrete situation and the user's intention. Either answering yes or no will only be reasonable for some cases, but unreasonable for some others. In this specification, we choose to let M_1 still export x_i whenever possible.

The second reason is due to the Haskell 98 module system. The module system of Haskell 98 is simple and flexible, but not very powerful at controlling the export list. For example, in Haskell 98, an entity in the export list can be of the form "Module M", which represents the set of all entities that are in scope with both an unqualified name "e" and a qualified name "M.e". But unlike the case for import declarations, where entities can be excluded using *hiding* (ip_1,...,ip_n), there is no such mechanism with exports. Therefore, when "module M" is used in the export list, no entity which is in scope with both an unqualified name "e" and a qualified name "M.e" can be excluded from being exported. Another example is that if the export list of a module is omitted, then all values, types and classes defined in the module are exported. The only way to exclude some entities from being exported is to use an explicit list to specify those entities to export. This is inconvenient when the program developer wants to export most of, but not all of, the defined entities in the module.

From the refactoring point of view, a major inconvenience caused by this lack of control in the export list is that, when a new identifier is brought into scope in a module, the identifier could also be exported automatically by this module, and then further exported by other modules if this module is imported and exported by those modules. This is dangerous in some cases as the new entity could cause name conflict/ambiguity in modules that import it either directly or indirectly, as shown in the example in Figure 23. While it is possible to check each potentially affected module to detect these problems, it will certainly slow down the refactoring process. Two

```
module M1 where

sq x = x^ 2

module M2 ( module M1, bar) where
import M1

bar x y = x + y

module Main where
import M2

foo = x^ 3
main
   = print $ foo 10 + bar 20 30
```

Figure 23: Adding a definition named *foo* to module *M1* will cause ambiguity in the module *Main*

strategies are used in the transformation rules in order to overcome the inconvenience caused by this lack of control in the export list. The first strategy is to use *hiding* in an import declaration to exclude an identifier from being imported by a client module of a module, N say, when we would like to, but unable to, exclude it from being exported by module N, as in case 1.1. The second strategy is to use a proper *alias* in the import declaration that changes the import path of an identifier from one module to another, therefore avoiding the changes to the module interface. This is used in the definition of *chgImpPath(Imp, y,* M_1*,* M_2*)*, as well as in the specification of Imp_1' in case 1.2 and case 2.1, where **import** M_2 **as** $M_1(x_i)$ is used to ensure that the interface of module M_1 stays unchanged. This way, we are able to confine the affected modules to M_1, M_2, and those that directly import M_1 or M_2, and we are also able to keep the qualifiers associated with the identifier whose definition is being moved unchanged (except in module M_2).

Let us re-visit the example shown in Figure 22, where the definition of *foo*

is moved from module M_1 to M_2. The result shown in the right-hand side of Figure 22 is slightly different from what we would get, as shown in Figure 24, by applying the above specified transformation rules to the original program. However, a couple of subsequent refactorings, i.e., *removing an unused module alias* and *cleaning the import list*, could simplify the import declarations of the $Main$ module to a single import declaration: *import* $M2(bar, foo)$.

```
module M1(foo, sq) where

sq x = x^ pow
  where pow = 2

foo x y = sq x + sq y

module M2 where
import M1(sq)

bar x y = sq(x + y)

module Main where
import M1
import M2(bar)

main
  = print $ foo 10 20 + bar 30 40
```

```
module M1(sq) where

sq x = x^ pow
  where pow = 2

module M2 where
import M1(sq)

foo x y = sq x + sq y

bar x y = sq(x + y)

module Main where
import M1
import M2(bar)
import M2 as M1(foo)

main
  = print $ foo 10 20 + bar 30 40
```

Figure 24: Move the definition of *foo* from module *M1* to *M2*

7.8 Verification of *Move a Definition from One Module to Another*

We argue towards the correctness of this refactoring from two aspects: the program after the refactoring is syntactically correct, and does not violate any static semantic properties, i.e., the program after the refactoring compiles without errors;

the refactoring does not change the behaviour of individual functions throughout the program. More details follow.

- Syntactically, this refactoring affects modules M_1, M_2 and all those modules that directly import M_1 or M_2. Syntactic correctness is obvious from the transformation rules. As to static semantic properties, possible violations are avoided in a number of ways. In this proof, we take case 2.2 in the specification as an example. Proof for the other cases follows the similar pattern.

  In module M_1, where the definition is no longer in scope after the refactoring, side-condition c) guarantees that *undefined identifier* will not be caused in the body of the module, and $rmFromExp\ (Exp_1, x_i, M_1)$ ensures that *undefined identifier* will not occur in the export list of this module.

  In module M_2, side-condition a) guarantees that adding the definition does not cause *name conflict* in the top level of M_2; side-condition b) guarantees that *undefined identifier* will not occur within the definition after it has been moved to M_2; $rmFromImp\ (Imp_2, x_i, M_1)$ ensures that M_2 no longer imports the identifier from M_1, therefore *unexported identifier* in the import list of M_2 is avoided; $addToExp\ (Exp_2, x_i, M_2)$ ensures that the identifier is available to other modules.

  In the client modules of M_1, M_j say, uses of $rmFromImp$ ensure that M_j no longer imports the definition from M_1, therefore *unexported identifier* in the import list is avoided; The uses of *ChgImpPath* ensure that the identifier is still visible to the sites where it is called after its definition has been moved. The visibility of the definition in the client modules of M_2 is unchanged after the refactoring.

  Recursive modules are not introduced during this refactoring. On one hand, moving the definition does not add any import declarations to M_2, therefore, there is no chance for M_2 to import any of its client modules. On the other hand, an import declaration importing M_2 is added to other modules only

when it is necessary and M_2 is not a client module of them because of conditions c), d) and the condition checking in case 2.2.

- The refactoring does not change the behaviour of individual functions. Firstly, the refactoring does not change the structure of individual definitions. It is obvious from the transformation rules that, apart from the definition of $x_i = E_i$, none of the other function declarations is syntactically modified by this refactoring. Within the definition of $x_i = E_i$, the only change is that the uses of $M_1.x_i$ have been changed to $M_2.x_i$, this is necessary as x_i is now defined in module M_2. We keep the qualified names qualified in order to avoid name capture within E_i.

  Secondly, the refactoring does not change the binding structure of variables. It is clear that the binding structure of local variables is not affected during the refactoring. As to lop-level identifiers, this refactoring creates a binding structure which is isomorphic to the one before the refactoring. Suppose the binding structures before and after the refactoring are B and B' respectively, then B and B' satisfy the following relation:

  $$B' = \{(f\ x, f\ y) | (x, y) \in B\},$$
  where $f(M, x) = (M_2, x_i)$ if$(M, x) \equiv (M_1, x_i)$; (M, x) otherwise.

  The only change from B to B' is that the defining module of x_i has been changed from the original M_1 to M_2. This is ensured through the changes to the import/export declarations of those involved modules and side-condition b).

7.9 Summary

Clear specification of refactorings provides a way to express the meaning of refactorings accurately, and a verification of behaviour preservation ensures the correctness of the specification. This chapter explores the specification and verification

of behaviour preservation of refactorings in the context of refactoring Haskell programs. To this purpose, we first defined the simple lambda-calculus called λ_{Letrec}, and then augmented it with a module system. Two representative refactorings are examined in this chapter, and they are *generalise a definition* and *move a definition from one module to another*.

More structural refactorings or module-related refactorings, such as *renaming*, *specialise a definition*, *lifting a definition*, *add an item to the export list*, etc, can be specified and verified in this framework without difficulty.

The framework needs to be extended to accommodate more features from the Haskell 98 language, such as constants, case expressions, data types, etc, so that more complex refactorings, such as data-type related refactorings can be specified. Nevertheless, this work provides a foundation for the further study of specification and verification of Haskell refactorings. Finally, a formally defined semantics for Haskell could help the (potentially automated) verifications of Haskell refactorings.

Chapter 8

Related Work

This chapter reviews relevant work in the literature in both refactoring and those related areas. It is structured as follows. Section 8.1 explores current research on the support of refactorings, including bad smell detections, guarantee of behaviour preservation, existing refactoring tools and their implementations, and language-parameterised refactoring. Section 8.2 discusses program transformations

in the functional programming paradigm, including the well-known *fold-unfold* program derivation system[18] , the Munich CIP Project [11, 12] and some other program transformation systems. The program slicing technique and its relation to refactoring are discussed in Section 8.3.

8.1 Existing Refactoring Approaches

Various tools, techniques and formalisms have been developed over the last decade for supporting different activities involved in the refactoring process for a variety of programming languages. There are tools for detecting where and which refactorings should be applied to a software, and tools for automating the application of refactorings. There are also proposed techniques for guaranteeing behaviour preservation or maintaining the consistency between the refactored program source and other software artefacts such as design specifications, documentations and tests.

8.1.1 Bad Smell Detection

When refactoring a program, the first decision to make is where and which refactoring should be applied to the program. The most widely used approach is the identification of bad smells. According to Martin Fowler and Kent Beck [35], bad smells are 'structures in the code that suggest(sometimes scream for) the possibility of refactoring'. For example, a parameter not used by the function body indicates the *remove an unused parameter* refactoring. While human examination is still the most widely used approach to detecting bad smells, some tools have emerged to help detecting the opportunities for refactoring.

One of the bad smells is duplicated code. Manual source code copy and modification is often used by programmers as an easy means for functionality reuse. Such practice produces duplicated pieces of code where consistent maintenance might be difficult. A variety of approaches to detecting duplicated or near-duplicated code have been proposed in [13], [29], [9], [24]. While different strategies are used in these papers, duplicated code detection usually involves three steps: first transform the source code into an internal format such as AST, token stream or processed string; then a comparison algorithm is performed on the internal data, and after that the result is shown in an understandable format. $CloneDR^{TM}$ [99] is a commercial product sold by Semantic Designs, Inc. The tool can automatically locate exact and near-miss duplicated code in software systems written in C, C++, Java or COBOL. Detected duplicated code can be automatically or interactively removed depending on the language.

As a proof of concept, T. Tourwé *et al.* [106] proposed a semi-automated approach based on logic meta programming (LMP) to formally specify and detect bad smells, and to propose refactoring opportunities that remove those bad smells. This LMP technique is independent of the particular base language that is used. One prototype tool they have implemented was within the VisualWorks object-oriented programming environment, of which the Refactoring Browser[17] is an integral part. The tool offers a list of logic queries that can directly be invoked

by the user from within the Smalltalk browser. Upon selection of a class and a query, the user can invoke the logic query, and the logic query will be executed and the results will be shown.

In [101], F. Simon *et al.* use object-oriented metrics to identify bad smells and propose refactorings. They focus on use relations (features that heavily use each other should belong to the same class) to propose *move method/attribute* and *extract/inline class* refactorings. The key underlying concept is the distance-based cohesion metric, which measures the degree to which methods and variables of a class belong together. This approach is combined with automatic visualisation technique to make the results understandable and adjustable to individual goals.

In [55], Kataoka *et al.* proposed to use program invariants to automatically identify candidate refactorings based on the idea that a particular pattern of invariants identifies a candidate refactoring and where to apply it. For example, *remove parameter* is applicable when a parameter is a constant or a function of other variables in scope at the procedure entry (invariants at a procedure entry). In this approach, a dynamic invariant detection technique is used to discover possible invariants from program executions by instrumenting the target program to trace the variables of interest, running the instrumented program over a test suite, and inferring invariants over the instrumented values. This approach is complementary to other approaches based on static information.

The number of refactorings that can be proposed by tools is still limited as the detection of some bad smells can be very computation intensive. Another problem is that the proposed refactorings may not always the ones that were needed, human judgement is still required to decide whether a candidate refactoring should be applied.

8.1.2 Guarantee of Behaviour Preservation

Refactorings should preserve the behaviour of software. Ideally, the most fundamental approach is to formally prove that refactorings preserve the full program semantics. This requires a formal semantics for the target language to be defined.

However, for most complex languages such as C++, it is very difficult to define a formal semantics. Even for a functional programming language like Haskell 98, there is still a lack of an officially defined semantics. In this case, people usually adopt the idea of invariants, pre-conditions or post-conditions, to ensure the preservation of semantics. The refactorings in HaRe were written in a compact and transparent way using Strafunski and the HaRe API. Together with the formal specification of refactorings, this gives us a high degree of assurance about the behaviour-preservation of the implemented refactorings.

Opdyke [79] proposed a set of seven invariants to preserve behaviour for refactorings. These invariants, which were found to be easily violated if explicit checks were not made before a program was refactored, are: unique superclass, distinct class names, distinct member names, inherited member variables not redefined, compatible signatures in member function redefinition, type-safe assignments and semantically equivalent reference and operations. Opdyke's refactorings were accompanied by proofs which demonstrated that the enabling conditions he identified for each refactoring preserved the invariants. Opdyke did not prove that preserving these invariants preserves program behaviour. In [105], Tokuda *et al.* also made use of program invariants to preserve behaviour of refactorings.

The notion of precondition is also used in [15] for formal restructuring using the formal language WSL. The objective of this paper is to recover a formal requirement specification for a legacy system, given only the source written in a typical second or third generation language.

In [104], F. Tip *et al.* explored the use of type constraints to verify the preconditions and to determine the allowable source code modifications for a number of generalisation related refactorings (e.g., *extract interface* for re-routing the access to a class via a newly created interface, and *pull up members* for moving members into a superclass) in an object-oriented program language context. In this setting, the authors start with a well-typed program, and use type constraints to determine whether declarations can be updated, or whether members can be moved without affecting a program's well-typedness.

Using a different approach, Tom Mens *et al.* [74] explored the idea of using graph transformation to formalise the effect of refactorings and prove behaviour preservation. Their motivation is that there is a direct correspondence between refactoring and graph transformation: programs can be expressed as graphs, and refactorings correspond to graph production rules, the application of a refactoring corresponds to a graph transformation, refactoring pre and post conditions can be expressed as application pre and post conditions. This approach proposed the graph representation of those aspects (access relation, update relation and call relation) of the source code that should be preserved by a refactoring, and graph rewriting rules as a formal specification for the refactoring transformations themselves. Type graphs, forbidden subgraphs, embedding mechanisms, negative application conditions and controlled graph rewriting were used in the formalisation. The formalisation of two sample refactorings, *EncapsulateField* and *PullupMethod*, was discussed in [74]. This research is still in its early stage, further research is needed to find out whether this approach can handle more complex refactorings, such as multi-module refactorings, and other aspects of behaviour preservation.

8.1.3 Existing Refactoring Tools

A number of tools have been developed to automate the application of refactorings, especially for object-oriented programming languages. While most of these tools have been populated with some basic refactorings such as extracting a method, renaming and moving a piece of code around, the number of supported refactorings is still limited compared with the catalogue of refactorings proposed by Martin Fowler in his refactoring book [35]. Table 2 summarises most of the currently available refactoring tools and the systems and languages they support, many of which are commercial products. More detail information can be found at Fowler's refactoring website *http://www.refactoring.com/tools.html*. The following is an overview of a representative selection of those tools mainly from the academic area.

Language	*Refactoring Tool*	*Description*
Smalltalk	Refactoring Browser	A browser for VisualWorks, VisualWorks/ENVY, and IBM Smalltalk.
Java	IntelliJ IDEA	A Java IDE with refactoring support.
	CodeGuide	A Java IDE with refactoring support.
	Eclipse	A Java-based extensible development platform.
	JFactor	A plug-in for JBuilder and Visual Age.
	XRefactory	A C and Java Refactoring Browser for Emacs, XEmacs and jEdit.
	JRefactory	A plug-in for JBuilder, NetBeans, and Elixir IDEs.
	RefactorIt	A plug-in for NetBeans, Sun Java Studio, Eclipse, JDeveloper and JBuilder, or as a stand-alone tool.
	JavaRefactor	A plug-in for JEdit.
	Elbereth	A Java refactoring tool based on Star Diagram
	DPT	A tool for introducing design patterns
	ConTraCT	A refactoring editor for Java
.NET	ReSharper	An add-in for VisualStudio.NET.
	C# Refactory	An add-in for VisualStudio.NET.
	Refactor!Pro	A .NET refactoring tool that supports both C# and Visual Basic.
C/C++	SlickEdit	A program editor supporting C/C++ refactorings.
	Ref++	A visual studio add-in that supports C++ refactorings.
	XRefactory	A C and Java Refactoring Browser for Emacs, XEmacs and jEdit.
	CRefactory	A refactoring tool for C programs.
	Proteus	A tool focus on C/C++ source transformation.
Visual Basic	Refactor!Pro	A .NET refactoring tool that supports both C# and Visual Basic.
Python	Bicycle Repair Man	A refactoring browser for python.
Self	Guru	A tool for restructuring inheritance hierarchies written in Self.
Delphi	ModelMaker	A class explorer and refactoring browser for Delphi.
Haskell	HaRe	A Haskell 98 refactorer embedded in Emacs and Vim.

Table 2: A summary of the currently available refactoring tools

The Smalltalk Refactoring Browser

The Refactoring Browser is the first successfully implemented tool, and still one of the most full-featured refactoring tools. It supports Smalltalk refactorings [93, 94]. The success of this tool is mostly due to its integration with the Smalltalk environment and development tools, its support for undo/redo of refactorings, and its efficiency. The Refactoring Browser can be considered as an extension to the Smalltalk development browser, which offers both program transformation and code browsing facilities.

The Refactoring Browser implements the preconditions proposed by Opdyke [79], and it also uses postconditions, which were proposed by Roberts in his PhD thesis [94], to eliminate some of the analysis in proving preconditions inside composite refactorings.

The Refactoring Browser operates by first parsing the code to be refactored and creating an abstract syntax tree (AST). The available transformations are encoded as templates in the form of ASTs, which may contain template variables. The transformation is accomplished by a *parse tree rewriter* that matches the concrete AST with a template AST and performs the tree-to-tree transformation. Finally, the modified AST is passed to the *Formatter* to get the source back from the tree. Instead of using the standard Smalltalk parser, the Refactoring Browser uses its own Smalltalk parser in order to accept pattern variables and keep comments in the AST. The Refactoring Browser does not preserve program layout.

Refactorings are implemented using *RefactoryChange* objects in the Refactoring Browser. Each RefactoryChange object also implements an *undo* method which can undo the changes performed by the object. By ensuring that each of small changes can undo itself reliably, the Refactoring Browser can ensure that complex refactorings can be undone safely.

The refactorings implemented in the Refactoring Browser, as shown in table

Add Class	Add Instance Variable
Remove Class	Remove Instance Variable
Rename Class	Rename Instance Variable
Remove Method	Abstract Instance Variable
Rename Method	Create Accessors for Instance Variable
Add Parameter to Methods	Add Class Variable
Remove Parameter from Method	Remove Class Variable
Rename Temporary	Rename Class Variable
Inline Temporary	Abstract Class Variable
Convert Temporary to Instance Variable	Create Accessors for Class Variable
Extract Code as Temporary	Convert Superclass to Sibling
Extract Code as Method	Inline Call
Push Up/Down Method	Push Up/Down Instance Variable
Push Up/Down Class Variable	Move Method to Component
Convert Instance Variable to Value Holder	Protect Instance Variable
Move Temporary to Inner Scope	

Table 3: Refactorings implemented in the Refactoring Browser

3, are typical to most object-oriented programming languages, and can be categorised as: *class refactorings* which change the relationships between the classes in the systems, *method refactorings* which change the methods within the system, and *variable refactorings* which change the instance variables within classes.

CRefactory

A. Garrido at the UIUC is working on a refactoring tool, called CRefactory, for C Programs [36]. A major challenge with refactoring C programs is that the source code of C programs has preprocessor directives intermixed. Preprocessor directives are hard to handle because it is difficult to carry information of directives from the source code to abstract program representations and it is difficult to guarantee correctness in the transformation. As a matter of fact, preprocessor directives is also an issue for practical Haskell libraries. In [37], Garrido proposed an approach to allow incompatible conditional branches to be analysed and modified simultaneously, and this is achieved by maintaining multiple branches in the transformed program tree, each annotated with its respective conditions. In order to include conditional directives in the AST, a pre-transformation phase

is performed to ensure that condition directives appear at the same level as external declarations or statements in the C program. As to the implementation architecture, Garrio reused most of the design ideas of the Smalltalk Refactoring Browser [93], with the re-implementation of some components of the architecture in a C context.

Proteus

Proteus [115] is a current research project at Bell Labs focusing on the development of a C/C++ source transformation system. Two significant problems addressed in their research are: 1) retaining layout and comment details in the transformed code whenever possible; and 2) correct handling of C preprocessing and the presentation of a semantically correct view of the program during transformation. Differing from our approach to program appearance preservation, Proteus uses a specialised form of AST which retains literal (keywords and punctuation), layout and commenting information. This form of AST, also called Literal-Layout AST (LL-AST), is usually much larger than the equivalent basic AST. As to the handling of C/C++ preprocessor directives, instead of extending the grammar to cater for directives (as in [37]), Proteus treats all directives as layout by embedding them directly into layout strings, and uses recorded macro expansion coupled with slicing and merging of parallel conditional branches. In order to hide the complexity of the LL-AST, a transformation language call YATL(Yet Another Transformation Language) was developed over the LL-AST. Primitives provided by Stratego (a weakly-typed strategy library akin to Strafunski) [112], including generic traversals, term matching, construction and deletion, are used by YATL.

Star Diagram

The Star Diagram is a graphical visualisation tool developed by Bowdidge [16]. A star diagram is generated from the abstract syntax tree and the program dependency graph. It provides a hierarchical tree-structured visual representation

of the source code relating to a particular data structure, eliding code unrelated to the data structure's use (this technique is also known as program slicing, see Section 8.3). Similar code fragments are merged into *node stacks* to reveal potentially redundant computations. The tool can help a programmer plan out a change with respect to how the change is distributed and how the fragments are related to each other. The visualisation is mapped directly to the program text, therefore manipulation of the visualisation also restructures the program.

The visualisation provided by the Star Diagram system is targeted at supporting the specific task of data encapsulation. Other kinds of transformation would require the assistance of other views.

Tools based on the notion of star diagram have been developed for C, Ada, and Tcl/Tk. Korman applied the star diagram concept to Java programs and implemented a tool called *Elbereth*. In [59], he described how programmers can be supported in performing a variety of refactoring tasks, such as extracting a method or replacing an existing class with an enhanced version. While the tool can assist the programmer in *planning* the restructuring, the restructuring itself has to be performed by hand.

Guru

Guru is a prototype tool developed by I. Moore for restructuring inheritance hierarchies expressed in the Self programming language [76]. This tool can automatically restructure an inheritance hierarchy into an optimal one for the objects currently in the system, whilst preserving the behaviour of programs. Here, *optimal* means that there are no duplicated methods and there are the minimum number of objects and inheritance relationships required for such an inheritance hierarchy.

The optimisation is achieved by first creating a copy of the objects to be restructured, in which the inheritance hierarchy is thrown away, then building a replacement inheritance which ensures no duplication.

Moore found that the inheritance hierarchies produced by Guru are easy to

understand when restructuring well-written code. For poorly written code, the inheritance hierarchies created by Guru may bear so little resemblance either to the original system or to any concepts that are understood by the programmer or designer that the restructured system may be very difficult to understand, although it may assist the programmer in identifying the faults of the original design.

The Design Pattern Tool (DPT)

In his PhD thesis [21], Mel Ó Cinnéide extended the existing work on refactoring and behaviour preservation (primarily that of Opdyke and Roberts) by merging this with the notion of design patterns as targets for automated program transformations. His methodology deals with the issues of reusing existing transformations for building more complex transformations, preservation of program behaviour, and the application of the transformations to existing program code. He also extended the existing refactoring composition method by allowing the transformations (composite refactorings) to contain not only simple sequences, but also iteration and conditional statements.

Mel Ó Cinnéide developed a prototype tool called DPT for the automatic application of design patterns to an existing Java program in a behaviour-preserving way. To apply a design pattern with this tool, the user first selects the program entities, then requests the tool to perform the transformation. DTP makes sweeping changes to a program when it applies a pattern, and this may prevent programmers from using the tool when they do not have a clear mental model of what the tool does.

ConTraCT

ConTraCT, a *Conditional Transformation Composition Tool*, is a refactoring editor for Java developed by G. Kniesel and H. Koch [57]. In this experimental system, the authors further explored the idea of composite refactorings from D. Roberts [94] and Mel Ó Cinnéide [22], and examined the idea of static composition

of a sequence of Java Refactorings. The *refactoring editor* provides the ability to compose larger refactorings from existing ones, that is, it provides a set of atomic conditions and transformations along with the ability to edit, compose, store, load and execute *conditional transformation* of Java programs. In this research, *conditional transformation*(CT) is used to represent a pair consisting of a precondition and a transformation, where the transformation is performed on a given program only if its precondition evaluates to *true*, therefore, a refactoring is just a special form of conditional transformation, i.e., a behaviour-preserving one. As revealed in [57], one major issue with static composition of refactorings is the automatic derivation of the preconditions for the composed refactoring; and another is the design of a complete, but minimal, collection of condition and transformation API.

8.1.4 Language-parameterised Refactoring

Ralf Lämmel has proposed the idea of representing program transformations for refactoring in a language-parametric manner using *Strafunski* [62, 65].

The basic idea of language-parameterised refactoring (or generic refactoring) framework is like this: first, *generic algorithms* are offered to perform simple analysis and transformations in the course of refactoring; second, an *abstraction interface* is provided to deal with the relevant abstractions of a language; and then the actual *generic refactorings* are defined in terms of generic algorithms and against the abstraction interface. The framework is designed in a way that it can be instantiated for different languages, such as Java, Prolog and Haskell. Lämmel used the *abstraction extraction* refactoring as a running example, and illustrated how this framework can be instantiated for (a subset of) Java. This is a challenging task because of the multiple languages that are subject to analysis and transformation, the program-entity based nature of refactoring tools, the complexity of language semantics, and the different semantics between different programming languages. In other words, it is impossible to be completely language independent.

8.2 Program Transformation for Functional Programs

Program transformation for functional programs has a long history, with early work in this field being described in the survey papers [83, 44]. Apart from refactorings, which transform the structure of a program and many are 'bidirectional', most other functional program transformations have a 'direction' , for instance from less to more efficient. In this context, a program transformation system takes a functional specification of an intended computation or a source-level program, rewrites the program using transformation rules into an efficient program. Program transformations are also used automatically in optimising compilers, acting either on source level programs or their intermediate language representation. For example, in [52], Simon Peyton Jones describes the experience of the GHC team in applying transformational techniques in a particularly thorough-going way in the Glasgow Haskell Compiler (GHC) [38].

Representative works of program transformation are Burstall and Darlington's well known *fold/unfold* program derivation system [18], the Munich CIP Project [11, 12] and the Bird-Meertens formalism (BMF) [73, 8, 90], which we discuss in more detail now.

8.2.1 Fold/Unfold

The fold/unfold system [18] was intended to transform recursively defined functions. This system is based on six transformation rules:

- *Definition*: introducing a new recursion equation whose left-hand expression is not an instance of the left-hand expression of any previous equation.
- *Unfolding*: replacing a function call with the body of the function where actual parameters are substituted for formal parameters.
- *Folding*: replacing an expression with a function call if the function's body can be instantiated to the given expression with suitable actual parameters.

- *Instantiation*: introducing a substitution instance of an existing equation.
- *Abstraction*: introducing a where clause by deriving from a previous equation a new equation.
- *Laws*: the use of laws about the primitives (associativity, commutativity, etc) of the language to obtain a new equation.

The advantage of this methodology lays on its simplicity and effectiveness at a wide range of program transformations. One disadvantage is that the use of the *fold* rule may result in non-terminating definitions, and apart from that, this rule requires that a history of the program must be kept as it is being transformed, which is not supported by our current refactoring framework.

8.2.2 The Munich CIP Project

Another representative work of program transformation is the Munich project CIP (Computer-aided Intuition-guided Programming) [11, 12], consists of two main parts: the design of a programming language and the development of a program transformation system. The key idea of the program transformation system was to develop programs by a series of small, well understood transformations. It used a rule-based language to describe transformations, and all transformation rules are specified by laws about program schemes (of which concrete programs are a special case) [84]. For instance, a rule to eliminate a conditional statement can be expressed as:

$$\frac{\text{if B then E else E}}{\text{E}} \; \text{DEF(B)}$$

This rule uses scheme variables B and E to identify parts of the expression to be transformed. The side condition of this rule states that the transformation is only valid if the Boolean expression B is defined.

This approach preserves total correctness as long as all rules preserve total correctness, and it allows the user to derive their own transformation rules. Somehow

similar to refactoring tools, the expressiveness of this approach depends on the available transformation rules, so systems using this approach may have dozens of transformation rules, and the user needs to search for applicable rules to solve the problem at hand.

8.2.3 The Bird-Meertens Formalism (BMF)

The Bird-Meertens Formalism (BMF) [73, 8, 90] also called Squiggol, is a calculus for deriving programs from their specification by a process of equational reasoning. It consists of a set of higher-order functions that operate mainly on lists including map, fold, scan, filter, inits, tails, cross product and function composition. BMF is based on a computational model of categorical data types and their accompany operations. Program developments in BMF are directed by considerations of *data* structure, as opposed to *program* structure.

8.2.4 Other Program Transformation Systems

What follows is a list of more recently developed transformation tools, most of which are aimed on program derivation and optimisation.

- **Stratego/XT** [112, 113] is a framework for the development of fully automatic program transformation systems. The framework consists of the transformation language Stratego and the XT collection of transformation tools. Stratego is a modular language for the specification of fully automatic program transformation systems based on the paradigm of correctness preserving rewriting under the control of programmable rewriting strategies in different ways. In Stratego, basic transformation steps are specified by means of *conditional rewrite rules*, and different transformation rules can be composed into *rewriting strategies*. Similar to Strafunski, Stratego also supports *Generic traversal* by means of a set of traversal combinators. Another feature of Stratego is the use of *concrete syntax* patterns for specifying transformation rules. The XT tools provide facilities for the infrastructure

of transformation systems including parsing and pretty-printing. Stratego has been used in the implementation of the refactoring tool Proteus as mentioned in the previous section.

- **HsOpt** [111] is an optimiser for the Helium (a subset of Haskell) compiler implemented in the transformation language Stratego.

- **Ultra** [40] is an interactive program transformation system intends to assist programmers in the formal derivation of correct and efficient programs from high-level descriptive or operational specifications. The transformation calculus supported by Ultra has its roots in the transformation semantics of the CIP system [11, 12]. The formulation of target programs in Ultra is based on the functional language Haskell.

- **PATH** [107] is another interactive program transformation system. The system was built with an aim to have the advantages of both the unfold/unfold approach and the approach taken by the CIP project, and the disadvantages of neither. PATH preserves the termination of definitions.

- **MAG** [102] is a program transformation system for a small functional language similar to Haskell. One feature of MAG is that it allows the user to write source code that actively takes part in the compilation process by providing instructions to the compiler on how to optimise it. The other feature is that a novel higher order matching algorithm [27] for lambda expressions is used in the implementation.

- **HULA** [30, 31] is a rule-based language for expressing changes to Haskell programs in a systematic and reliable way. The update language essentially offers update commands for all constructs of the object language (a subset of Haskell). The update language can be translated into a core calculus consisting of a small set of basic updates and update combinators. The idea underneath the update language is to view programs as abstract data types (ADT) [32] and performing program changes by applying well-defined

ADT operations on the program, and basic updates are be combined into update programs that can be stored. A type change inference system that can automatically infer type changes for updates has been developed, and the type of an update program is given by the possible type changes it can cause for an object programs.

8.3 Program Slicing

Program slicing is a technique for aiding program debugging, testing and understanding by isolating portions of a software system to reduce its complexity. Two major forms of slicing are static slicing and dynamic slicing. A *static program slice* with respect to a set of variables V at some point of interest p is the parts of the program that may affect the values of some variables in V at the point p. (p,V) is also called a *slicing criterion*. Therefore, a static program slice can be derived by deleting the statements that have no effect on the slicing criterion. The concept of static slicing was originally proposed by Mark Weiser in [117, 118]. A dynamic program slice is the part of a program that "affects" the computation of a variable of interest during program execution of a specific program input. Dynamic program slicing was originally proposed by Korel *et al.* [58] for program debugging purpose, but its application has been extended beyond that. Normally, static slices are typically larger, but cater for every possible execution of the original program; dynamic slices are much smaller, but only cater for a single input. Different slicing techniques and algorithms have been proposed in the last decade to improve the precision and performance of program slicing tools [103].

Program slicing can be used for refactoring to reduce the unnecessary coupling of parameters, variables or design concerns. An example is to refactor a function that returns a tuple, say (f, g), into two separate functions returning f and g respectively.

The idea of *sliced-based method extraction* was firstly explored in [71], and more recently by **Nate** [26], a project currently carried out at Oxford University

with a focus on slicing-based refactorings. Compared with the *method extraction* refactoring provided by traditional refactoring tools, *sliced-based method extraction* allows extracting a non-consecutive computation for the program.

8.4 Summary

The main challenges faced by automating the refactoring process are: a) identifying refactoring opportunities, b) global behaviour-preserving program transformation, and c) program appearance preservation. Automating the refactoring process benefits from the previous work on program lexing, parsing, analysis and transformation, but also exposes new research areas, such as producing ASTs or token streams with richer information, verifying global behaviour preservation, a general framework for building refactoring tools, etc. In the last decade, efforts have been made to support various aspects of the refactoring process, especially for object-oriented programming languages, and our work complement the existing research by exploring the properties of refactoring from the functional program language paradigm and building a refactoring tool for a full and layout-sensitive language. As functional programming languages and object-oriented programming languages expose different program structures, it is no surprise that each of them have a collection refactorings particularly favoured by their own program structures. However, there are still some refactorings, such as *renaming*, *add/remove parameters*, *duplicated code elimination*, etc, common to both programming paradigms, and the same challenges mentioned above need to be addressed when implementing refactoring tool for either programming paradigms. Proof of behaviour-preservation of refactorings for purely functional languages is relatively straightforward due to the clean semantics and the rich theoretical foundation for reasoning about functional programs.

Chapter 9

Conclusions and Future Work

9.1 Summary of Contributions

This study has explored several aspects of refactoring in the functional programming paradigm, as a complement to the existing works in refactoring programs written in object-oriented or imperative programming languages. The principal artefacts of this work are the Haskell refactorer, HaRe, and the API as a platform for implementing new refactorings and general program transformations. In particular, the following contributions have been made:

- The study of a set of Haskell refactorings.
- The design and implementation of the Haskell refactorer, which can be used by real-world Haskell 98 programmers.
- An approach to program appearance preservation.
- An API for implementing refactorings or general program transformations in HaRe.
- A simple language, λ_M (λ_{Letrec} extend with a module system), for the specification of refactorings.
- The specification and proof of a couple of representative refactorings using λ_M or λ_{Letrec}.

9.2 Future Work

Our current work can be further developed in a number of directions.

- ***Furture Development of HaRe***.
  - ***Adding more refactorings to HaRe***. The number of supported refactorings in HaRe is still limited comparing with the catalogue of refactorings listed on our project website [91]. C. Brown has been working on *duplicated code elimination*, which will be available from HaRe 0.4. Apart from that, more refactorings, such as the other refactorings mentioned in Section 2.7, can be implemented using the established framework and API.
  - ***Making use of type information***. The existing refactorings in HaRe do not utilize type information so far. This is partially because the type checking system from Programatica [81] is not efficient enough to be used in an interactive environment. However, type information is necessary for some refactorings. Apparently, any refactorings to do with classes and especially instances need type information. Apart from that, there are some other examples which need type information as well. For instance, when *generalising* a function definition which has a type signature declared, the type of the identified expression needs to be inferred, and added to the type signature as the type of the function's first argument. Another example is the *lifting a definition* refactoring. Lifting a simple pattern binding (i.e. a pattern binding in which the pattern consists of only a single variable) to the top level may make an originally polymorphic definition monomorphic, and fail the program at compilation. This problem could be avoided by adding proper type signature to the lifted pattern binding, but again, type information is needed in order to infer the type signature.

- ***Coping with modules without source code***. The current Programatica-based HaRe requires the source file for each module in the project for analysis purpose. A project normally contains the user's own source files and some library modules. The problem arises when the user is using a binary distribution of the libraries. One shortcut solution is to define a dummy module for each no-source module so that the project can be created and compiled. However, extra care still needs to be taken when a refactoring could involve the source code of the dummy functions defined in dummy modules. For example, apply the *unfolding* refactoring to the call-site of a library function does not make sense if the function is defined in a dummy module. C. Ryder from our research group is currently investigating the possibility of moving HaRe from Programatica to the GHC API [54] which is expected to be released via a package of GHC (6.6). If this switch succeeds, this problem will be solved automatically. Porting to GHC API would also give us faster type checking and support for commonly used language extensions. Moreover, the successful switching from Programatica to GHC API would also make it possible to incorporate HaRe with Visual Haskell [54], which is a full-featured Haskell development environment currently under development.

- ***More interaction between HaRe and the user***. Interaction between HaRe and the user during the refactoring process will allow the users to provide the refactorer with more information so as to guide the refactorer to proceed, therefore provides more flexibility to the user. For instance, when *introducing a definition* to represent a user-identified expression, the user may want to replace *only* the identified occurrence of the expression, *some* or *all* the occurrences of the expression. The current version of HaRe supports the two extreme cases, but does not support replacing *some* occurrences of the identified expression. Being able to interact with the user, the refactoring could

highlight the next found occurrence of the expression, and then ask the user whether he/she wants to replace this occurrence and proceed according to the user's answer. As to the implementation, this functionality mainly involves the interaction between the refactorer and the editor which hosts it.

- ***Optimization of HaRe***.
  - ***Reuse of the AST and module information between refactorings***. The efficiency of HaRe can be improved by reusing the available information as much as possible from one refactoring to another. The most obvious reusable information is the AST and the module system information. Being able to reuse this information, we could avoid the program source being re-parsed and modules being re-analysed when the next refactoring is invoked. However, this reusability is not straightforward. As the AST changes during the refactoring process, the refactorer needs to guarantee that no *dirty* information is introduced during the transformation phase. That is, apart from keeping the AST syntactically correct, the refactorer should also ensure the correctness of semantic information and location information in the ASTs, and that the module information reflects the new status of the program. For example, when a function definition, say *foo*, is lifted to the top level, the abstract syntax representation of *foo* should be changed to reflect the fact that *foo* is now a top-level identifier, and all the references to this identifier should be changed to refer to its new abstract representation. If *foo* will be exported by its defining module after the refactoring, then the defining module's export relation should be modified to accommodate this new top-level identifier. After the refactoring, the source locations of the identifiers which occur in the lifted definition will be changed as well. Tracing the changes of locations is very delicate and

complex in practice, and one possible solution is to lex the new program source after the refactoring, and then inject the new locations back into the ASTs. The new program source can be extracted from the token stream after the refactoring. The current implementation of HaRe allows *dirty* information to exist in the AST as long as this does not affect the refactoring process, and the module information is not updated during the refactoring process.

- ***Support for scripting refactorings***. A set of elementary refactorings can be applied sequentially or iteratively to a program in order to achieve the effect of a complex refactoring process, like the case of *introducing abstract data type* discussed in Section 5.3. In some other scenarios, it may well be that certain patterns of refactorings can be seen to occur. For example, *moving a definition from a module to another* can be followed by *cleaning up the import list*, and *lifting a definition* can be preceded by *renaming the identifier*. The more refactorings HaRe provides, the more likely that there will be some commonly used refactoring patterns. Therefore, it would be helpful to have a simple script language which can be used by HaRe or the users to easily build composite refactorings from the existing ones. This script language should provide the basic tactics for building composite refactorings, such as sequential, iterating and conditional application of refactorings. Obviously, there should also be a mechanism for interpreting and storing the script, so that the script can be invoked once it has been defined.

  One of the foreseen challenges with supporting scripting refactorings within the current HaRe framework lies in how to specify the parameters for the elementary refactorings involved in a composite refactoring. A refactoring normally takes some syntax phrase(s) as input. In the current implementation of HaRe, a syntax phrase is identified by its start and end location (or just the start location if the syntax phrase only contains an identifier)

in the source file, and this is very convenient for the user as he/she only needs to highlight the syntax phrase (or mouse click on the identifier) in the source, and HaRe will then infer the locations from the editor. However, for a scripted composite refactoring, users normally do not have access to the intermediate refactoring results, hence are not able to highlight any syntax phrase for the intermediate refactorings. Moreover, the location of identifiers can change from refactoring to refactoring, and the user will not be able to infer the changes and inform the refactorer the location of the syntax phrase. Therefore, except for some special cases where the parameters can be specified from the very beginning of the composite refactoring process using highlighting, some other method needs to be invented to identify syntax phrases in the program source. An obvious idea is to use addresses within the AST in some way or to use some scope information.

A naïve way of executing a composite refactoring is to execute the involved elementary refactorings in the specified order, and the composite refactoring fails if one of the elementary refactoring fails, and succeeds if all the elementary refactorings succeed. In this style, the execution of a composite refactoring follows a *side-condition checking* $\rightarrow$ *program transformation* $\rightarrow$ *side-condition checking* $\rightarrow$ *program transformation* $\rightarrow \ldots$ pattern. However, this can be improved by calculating the side-conditions for the whole composite refactorings beforehand. In this way, the execution of the composite refactoring will follow a *side-condition checking* $\rightarrow$ *program transformation* $\rightarrow$ *program transformation* $\rightarrow \ldots$ pattern, and if the composite refactoring will not succeed, it will fail in an early stage. In another word, we would like the scripts to put together the side-conditions and transformations, rather than the simple (whole) refactorings. The side-condition of a composite refactoring may *not* be the same as the union of the side-conditions of the elementary refactorings, as some conditions which are not satisfied by the current program might become satisfied after certain refactorings, and vice versa. The idea of *post-condition* has been used to infer the side-conditions

for composite refactorings [94], and how side-conditions for composite refactorings can be computed manually has been discussed in [22]. The automatic derivation of side-conditions is examined in [57].

- ***Towards metric-based refactoring***. The MEDINA [96] library is a collection of functions and data structures written in Haskell to aid in the implementation of software metrics for Haskell programs. Along with this library and a collection of simple metrics, visualisation functions are also provided to support the display of metric values and the program being measured. This work was carried out by C. Ryder for his PhD research at the University of Kent. By putting HaRe and MEDINA in the same program development environment, the user would be able to make use of the functionalities from both tools. Further more, the values of metrics can help the users to detect potential refactoring opportunities or *bad smells*, and after a refactoring, the program can be measured again to check whether the program's structure has been improved regarding to the measured parameters.

- ***Maintaining the consistency between the refactored program source and other accompanying software artefacts***. A real-world program source hardly exists alone. Together with the software, there might be documentation, testing suites, design documents, etc. It would be valuable to provide a systematic way to keep the different software artefacts consistent during the refactoring process. Even within the program source itself, the comments might become out-of-date when the commented source has been refactored. Automatic or semi-automatic refactorisation of the comments would also be helpful.

- ***Further development of the specification and verification of refactorings***. The current simple language λ_M can be extended to accommodate more features from the Haskell 98 language, such as constants, case-expressions, data types, etc, so that more refactorings can be specified and

verified in this framework. With the supporting of scripting refactorings, corresponding theory needs to be developed to specify and verify composite refactorings.

- ***Applying the research results to other functional programming languages***. The established framework gives us a starting point to look into refactorings in other functional programming languages, such as Clean [87], Standard ML [6], Erlang [89], OCaml [92], etc. Among these programming languages, Clean shares more features with Haskell than other languages, therefore applying the research results to it should be easier than to the other languages. We are going to build refactoring support for Erlang as part of the EPSRC supported project *Formally-based tool support for Erlang development*. Although Erlang differs from Haskell in many aspects, the experience gained from this research will be invaluable for building the Erlang refactoring tool.

Appendix A

The Definition of PNT

```
-- The definition of PNT and its comprising data types.

data PNT = PNT PName (IdTy PId) OptSrcLoc

data PN i = PN i Orig

type PName = PN HsName

type PId = PN Id

data HsName = Qual ModuleName Id
            | UnQual Id

type Id = String

data Orig = L Int
          | G ModuleName Id OptSrcLoc
          | D Int OptSrcLoc
          | S SrcLoc
          | Sn Id SrcLoc
          | P

data ModuleName = PlainModule String
                | MainModule FilePath

newtype OptSrcLoc = N (Maybe SrcLoc)
```

```
data SrcLoc = SrcLoc {srcPath :: FilePath
                      srcChar, srcLine, srcColumn :: !Int}

data IdTy i = Value
            | FieldOf i (TypeInfo i)
            | MethodOf i [i]
            | ConstrOf i (TypeInfo i)
            | Class [i]
            | Type (TypeInfo i)
            | Assertion
            | Property

data TypeInfo i = TypeInfo { defType :: (Maybe DefTy)
                             constructors :: [ConInfo i]
                             fields :: [i]
                           }
data DefTy = Newtype
           | Data
           | Synonym
           | Primitive

data ConInfo i = ConInfo { conName :: i
                           conArity :: Int
                           conFields :: (Maybe [i])
                         }
```

Appendix B

Some Combinators From StrategyLib

B.1 The Basic Combinators

```
-- Strategy application:
applyTP ::(Monad m, Term t) => TP m -> t -> m t
applyTU ::(Monad m, Term t) => TU a m -> t -> m a

-- Strategy update:
adhocTP ::(Monad m, Term t) => TP m -> (t -> m t) -> TP m
adhocTU ::(Monad m, Term t) => TU a m -> (t -> m a) -> TU a m

-- Deterministic combinators:
seqTP    :: Monad m => TP m -> TP m -> TP m
seqTU    :: Monad m => TP m -> TU a m -> TU a m

passTP   :: Monad m => TU a m -> (a -> TP m) -> TP m
passTU   :: Monad m => TU a m -> (a -> TU b m) -> TU b m

-- Combinators for partiality and non-determinism:
choiceTP :: MonadPlus m => TP m -> TP m -> TP m
choiceTU :: MonadPlus m => TU a m -> TU a m -> TU a m

-- * Traversal combinators:
-- Succeed for all children
allTP :: Monad m => TP m -> TP m
allTU :: Monad m => (a -> a -> a) -> a -> TU a m -> TU a m

-- Succeed for one child; don't care about the other children
```

```
oneTP :: MonadPlus m => TP m -> TP m
oneTU :: MonadPlus m => TU a m -> TU a m

-- Succeed for as many children as possible
anyTP :: MonadPlus m => TP m -> TP m
anyTU :: MonadPlus m => (a -> a -> a) -> a -> TU a m -> TU a m

-- Succeed for as many children as possible but at least for one
someTP :: MonadPlus m => TP m -> TP m
someTU :: MonadPlus m => (a -> a -> a) -> a -> TU a m -> TU a m

-- * Useful defaults for strategy update.
-- Returns the incoming term without change.
idTP        :: Monad m => TP m

-- Always fails, independent of the incoming term.
failTP          :: MonadPlus m => TP m
failTU          :: MonadPlus m => TU a m

-- Always returns the argument value 'a',
-- independent of the incoming term.
constTU         :: Monad m => a -> TU a m

-- Replace one monad by another:
msubstTP ::(Monad m, Monad m')
         => (forall t . m t -> m' t) -> TP m -> TP m'
msubstTU ::(Monad m, Monad m')
         => (m a -> m' a) -> TU a m -> TU a m'
```

B.2 The Recursive Traversal Combinators

```
-- * Full traversals
-- Full traversal in top-down order.
full_tdTP :: Monad m => TP m -> TP m
full_tdTP s = s `seqTP` (allTP (full_tdTP s))

full_tdTU :: (Monad m, Monoid a) => TU a m -> TU a m

-- | Full traversal in bottom-up order.
full_buTP :: Monad m => TP m -> TP m
full_buTP s = (allTP (full_buTP s)) `seqTP` s

-- * Traversals with stop conditions.
-- Top-down traversal that is cut of below nodes
-- where the argument strategy succeeds.
stop_tdTP :: MonadPlus m => TP m -> TP m
```

```
stop_tdTU :: (MonadPlus m, Monoid a) => TU a m -> TU a m

-- * Single hit traversal
-- Top-down traversal that performs its argument
-- strategy at most once.
once_tdTP :: MonadPlus m => TP m -> TP m
once_tdTU :: MonadPlus m => TU a m -> TU a m

-- Bottom-up traversal that performs its
-- argument strategy at most once.
once_buTP :: MonadPlus m => TP m -> TP m
once_buTU :: MonadPlus m => TU a m -> TU a m

-- * Traversal with environment propagation
-- Top-down type-unifying traversal with
-- propagation of an environment.
once_peTU :: MonadPlus m
  = e              -- initial environment
  -> (e -> TU e m) -- environment modification at downward step
  -> (e -> TU a m) -- extraction of value, dependent on environment.
  -> TU a m
```

Appendix C

The Layout Adjustment Algorithm

The function **adjustLayout** adjusts the token stream to compensate the change to layout caused by a token stream manipulation. It takes four parameters. The first parameter is the sequence of tokens starting from the token, say `t`, which is right after the added/deleted/updated tokens to the end of the token stream; the second parameter is `t` 's original off-side, and the third parameter is `t`'s new off-side; the fourth parameter is an interger used to create fresh locations.

```
-- Some auxiliary functions used by ‘adjustLayout’.
tokenRow (_, (Pos _ r _, _)) = r

tokenCon (_,(_,s)) = s

hasNewLn (_,(_,s))=isJust (find (=='\n') s)

isWhiteSpace (t,(_,s)) = t==Whitespace && s==" "

isWhite (t,_) = t==Whitespace || t==Commentstart
                || t==Comment || t==NestedComment

notWhite = not.isWhite

isKeyword t = elem (tokenCon t)["where","let","do","of"]

lenOfToks ts = length (concatMap tokenCon ts)

lastLineLenOfTok (_,(_,s))=
  = (length.(takeWhile (\c->c/='\n')).reverse) s

whiteSpaceTokens (row, col) n
   = if n<=0 then []
             else (Whitespace, (Pos 0 row,col, " ")
                   : whiteSpaceTokens (row,col+1) (n-1)
```

```
adjustLayout::[PosToken]->Int->Int->Int->[PosToken]
adjustLayout [] _ _ _ = []
adjustLayout _ oldOffset newOffset _ |oldOffset == newOffset = toks
adjustLayout toks oldOffset newOffset freshVal
 = case layoutRuleApplies of
    --offset: indentation of the not-white lexeme after the keyword.
    (True,offset)->let (ts:ts') = groupTokensByLine toks
                       chgInOffset = newOffset - oldOffset
                   in ts++addRmSpaces (last ts) chgInOffset offset ts'
    _   ->toks
  where
  layoutRuleApplies
    = let ts = takeWhile (not.hasNewLn) toks
          (ts1, ts2) = break isKeyword ts
      in case (filter notWhite ts2) of
         (_:t:_)->let ts2'= takeWhile (\t->isKeyword t || isWhite t) ts2
                  in (tokenCon t /="{", oldOffset+lenOfToks(ts1++ts2'))
         _      ->(False,0)

  groupTokensByLine [] = []
  groupTokensByLine xs
    = let (xs', xs'') = break hasNewLn xs
      in if (length xs''==0) then [xs']
          else (xs'++[head xs'']):groupTokensByLine (tail xs'')

  addRmSpaces t n offset [] = []
  addRmSpaces t n offset toks@(ts:ts')
    = case find notWhite ts of
      Just t  -> if lastLineLenOfTok t + lenOfToks ts1>= offset
                 then addRmSpaces' n ts
                      ++ addRmSpaces (last ts) n offset ts'
                 else concat toks
      _       -> ts ++ addRmSpaces (last ts) n col ts'
    where (ts1, _) = break notWhite ts

  addRmSpaces' 0 ts = ts
  addRmSpaces' _ [] = []
  addRmSpaces' n ts@(t:ts')
    = case n >0 of
       True -> whiteSpaceTokens (freshVal, 0) n ++ ts
       _    -> if isWhiteSpace t
               then addRmSpaces' (n+1) ts'
               else error $ "Layout adjusting failed at line:"
                            ++ show (tokenRow t)++ "."
```

Appendix D

The Implementation of *rename a Variable*

This appendix gives the implementation of *rename a value variable name*. The value variable to be renamed can be either a top-level variable or a local variable. In this implementation, qualified names are used to avoid *ambiguous occurrence* throughout the refactored program. This implementation works with multi-module programs.

This implementation can be further refactored by merging the two similar functions `doRename` and `doRenameInClientMod` into a single function, so as to eliminate the duplicated code. To make the representation clearer, we keep them as separate functions in this appendix.

The definitions of those API functions used by this implementation are not given in this appendix, nevertheless their names should reflect the meaning.

```
module RefacRenaming(rename) where

import Maybe
import List
import RefacUtils

-- The top-level function of renaming.
rename fileName newName row col
 = do modName <- fileNameToModName fileName
      -- inscps: in-scope entities; exps: exported entities;
      -- mod: the AST;  toks: the token stream.
      modInfo@(inscps, exps, mod, _) <- parseSourceFile fileName
      -- turns textual selction to the PNT represenation
      let pnt@(PNT pn _ _)= locToPNT fileName (row, col) mod
      -- * Condition checking
      -- Condition checking in the current module.
      condChecking pn newName modName (inscps, exps, mod)
      clientFiles
          <-if isExported pnt exps
              then do clientModsAndFile <- clientModsAndFiles modName
                      return (map snd clientModsAndFile)
              else return []
       clientModsInfo <- mapM parseSourceFile clientFiles
       -- Condition checking in client modules
       let _ = clientModsCondChecking pnt newName clientModsInfo
       -- * Transformation
       -- Renaming in the current module.
       r <- applyRefac (doRename pn newName modName)
                       (Just modInfo) fileName
       -- Renaming in client modules.
       rs <- applyRefacToMods (doRenameInClientMod pnt newName)
                              (Just clientModsInfo) clientFiles
       -- * output result
       writeRefactoredFiles False (r:rs)
```

```
-- Condition checking in the current module.
condChecking pn newName modName (inscps, exps, mod)
  = do condChecking1 pn newName modName
       condChecking2 pn newName modName (inscps, exps, mod)
 where
  defMod = if isTopLevelPN oldPN
             then fromJust (hasModName oldPN)
             else modName

  -- Some trivial condition checking.
  condChecking1 oldPN newName modName
   = do let old = pNtoName oldPN
        unless (oldPN /= defaultPN && isVarId old)
          $ error "Invalid cursor position!"
        unless (isVarId newName)
          $ error "The new name is invalid!"
        unless (oldName /= newName)
          $ error "The new name is the same as the old name!"
        unless (defMod == modName)
          $ error "The identifier is not defined in this module!"
        when (isTopLevelPN oldPN && old=="main" && isMainModule modName)
          $ error "This 'main' function should not be renamed!"
        when (isTopLevelPN oldPN
             && causeConflictInExports oldPN newName exps)
          $ error "Renaming will cause conflicting exports!"
        return ()

  -- Some non-trivial condition checking.
  condChecking2 oldPN newName modName (inscps, exps, mod)
    = applyTP (once_tdTP (failTP `adhocTP` inMod
                                 `adhocTP` inMatch
                                 `adhocTP` inPattern
                                 `adhocTP` inExp
                                 `adhocTP` inAlt
                                 `adhocTP` inStmts)) mod
     where
       -- return True if oldPN is declared by t.
       isDeclaredBy t = isDeclaredBy' t == Just True
        where
         isDeclaredBy' t
           = do (_ , d) <- hsFreeAndDeclaredPNs t
                Just (elem oldPN d )

       -- The name is a top-level identifier
       inMod (mod::HsModuleP)
         | isDeclaredBy (hsModDecls mod)
         = condChecking' mod
       inMod _ = mzero
```

```
-- The name is declared in a function definition.
inMatch (match@(HsMatch loc1 fun pats rhs ds)::HsMatchP)
  |isDeclaredBy pats
   = condChecking' (HsMatch loc1 defaultPNT pats rhs ds)
  |isDeclaredBy ds
   =condChecking' (HsMatch loc1 defaultPNT [] rhs ds)
  |otherwise = mzero

-- The name is declared in a pattern binding.
inPattern (pat@(Dec (HsPatBind loc p rhs ds)):: HsDeclP)
  |isDeclaredBy p
   = condChecking' pat
  |isDeclaredBy ds
   = condChecking' (Dec (HsPatBind loc defaultPat rhs ds))
inPattern _ = mzero

-- The name is declared in a expression.
inExp (exp@(Exp (HsLambda pats body))::HsExpP)
  |isDeclaredBy pats
  = condChecking' exp
inExp (exp@(Exp (HsLet ds e)):: HsExpP)
  |isDeclaredBy ds
  = condChecking' exp
inExp _ = mzero

-- The name is declared in a case alternative.
inAlt (alt@(HsAlt loc p rhs ds)::HsAltP)
  |isDeclaredBy p
   = condChecking' alt
  |isDeclaredBy ds
   = condChecking' (HsAlt loc defaultPat rhs ds)
  |otherwise = mzero

-- The name is declared in a do statement.
inStmts (stmts@(HsLetStmt ds _)::HsStmtP)
  |isDeclaredBy ds
  = condChecking' stmts
inStmts (stmts@(HsGenerator _ pat exp _)::HsStmtP)
  |isDeclaredBy  pat
  = condChecking' stmts
inStmts _ = mzero

condChecking' t
 = do when (elem newName (map pNtoName
             (declaredVarsInSameGroup oldPN t)))
        $ error "The new name exists in the same binding group!"
      (f, d) <- hsFreeAndDeclaredNames t
      when (elem newName f)
```

```
              $ error "Existing uses of the new name will be captured!"
            -- fetch all the declared variables in t that
            -- are visible to the places where oldPN occurs.
            ds<-hsVisibleNames oldPN t
            when (elem newName ds)
              $ error "The new name will cause name capture!"
            return t

-- Renaming in the current module.
doRename oldPN newName modName (inscps, _, mod)
  = do imps'<- renamePN' (hsModImports mod)
       exps'<- renamePN  (hsModExports mod)
       ds'  <- renamePN  (hsModDecls mod)
       return $ mod {hsModImports = imps',
                     hsModExports = exps', hsModDecls = ds'}
  where
   renamePN' = applyTP (stop_tdTP (adhocTP failTP inPNT))
    where
     inPNT pnt@(PNT pn _ _)
      |pn ==oldPN
      = update pnt (renameInPNT pnt Nothing newName) pnt

   renamePN = applyTP (stop_tdTP (adhocTP failTP inPNT))
    where
     inPNT pnt@(PNT pn _ _)
      |pn ==oldPN && defineLoc pnt == sourceLoc pnt
      = update pnt (renameInPNT pnt Nothing newName) pnt

     inPNT pnt@(PNT pn@(PN (UnQual _) _) _ _)
      |pn == oldPN
      = if isInScopeAndUnqualified newName inscps && isTopLevelPN oldPN
         then update pnt (renameInPNT pnt (Just modName) newName) pnt
         else update pnt (renameInPNT pnt Nothing newName) pnt

     inPNT pnt@(PNT pn@(PN (Qual _  _) _) _ _)
      |pn == oldPN
      = update pnt (renameInPNT pnt Nothing newName) pnt

     inPNT pnt@(PNT pn@(PN (UnQual _) _) _ _)
      |pNtoName pn == newName && isTopLevelPNT pnt && isTopLevelPN oldPN
      = do let qual'=Just $ ghead "renamePN" $ hsQualifier pnt inscps
           update pnt (renameInPNT pnt qual' newName) pnt

     inPNT _ = mzero
```

```
-- Condition checking in client modules.
clientModsCondChecking oldPN newName clientsInfo
  = any (==False) $ map (condChecking' oldPN newName) clientsInfo
  where
   condChecking' oldPN newName (_, exps, mod, _)
    = if causeConflictInExports oldPN newName exps
       then error
             $ "The new name will cause conflicting exports in "
               ++ show (hsModName mod) ++ "!"
       else False

-- Renaming in client moudles.
doRenameInClientMod pnt@(PNT oldPN _ _) newName (inscps,exps,mod)
  = do imps'<- renamePN' (hsModImports mod)
       exps'<- renamePN  (hsModExports mod)
       ds'  <- renamePN  (hsModDecls mod)
       return $ mod {hsModImports = imps',
                     hsModExports = exps', hsModDecls = ds'}
   where
    qual = ghead "doRenameInClientMod" $ hsQualifier pnt inscps

    renamePN' = applyTP (stop_tdTP (adhocTP failTP inPNT))
     where
      inPNT pnt@(PNT pn _ _)
        |pn ==oldPN
        = update pnt (renameInPNT pnt Nothing newName) pnt

    renamePN t = applyTP (stop_tdTP (adhocTP failTP inPNT)) t
     where
      inPNT pnt@(PNT pn@(PN (UnQual _) _) _ _)
        | pn == oldPN
        = if isInScopeAndUnqualified newName inscps
           then update pnt (renameInPNT pnt (Just qual) newName) pnt
           else do let qual' = do vs <-hsVisibleNames pnt t
                                  if elem newName vs
                                   then Just qual else Nothing
                   update pnt (renameInPNT pnt qual' newName) pnt

      inPNT pnt@(PNT pn@(PN (Qual qual _) _) _ _)
        | pn == oldPN
        = update pnt (renameInPNT pnt Nothing newName) pnt

      inPNT pnt@(PNT pn@(PN (UnQual _) _) _ _)
        | pNtoName pn == newName && isTopLevelPNT pnt
          && isInScopeAndUnqualified (pNtoName oldPN) inscps
        = do let qual' = Just $ head $ hsQualifier pnt inscps
             update pnt (renameInPNT pnt qual' newName) pnt

      inPNT _ = mzero
```

Appendix E

The Implementation of *From Concrete to ADT*

This appendix gives the implementation of the composite refactoring, *from concrete to abstract data type*, and its supporting refactorings.

```
module RefacADT(addFieldLabels,addDiscriminators,addConstructors,
                elimNestedPatterns,elimPatterns,createADTMod,
                fromAlgebraicToADT) where
import Maybe
import List
import Char
import Prelude hiding (putStrLn)
import AbstractIO (putStrLn)
import RefacUtils
-------------------------------------------------------------
-- Refactoring: from concrete data type to abstract data type.
-------------------------------------------------------------
fromAlgebraicToADT fileName row col
 = do info@(_, _, mod, _)<-parseSourceFile fileName
      case locToTypeDecl fileName row col mod of
       Left errMsg ->do putStrLn errMsg
       Right decl  ->
         do let typeCon = pNtoName $ fromJust (getTypeCon decl)
            seqRefac [doAddFieldLabels     typeCon (Just info) fileName,
                      doAddDiscriminators  typeCon  Nothing    fileName,
                      doAddConstructors    typeCon  Nothing    fileName,
                      doElimNestedPatterns typeCon  Nothing    fileName,
                      doElimPatterns typeCon Nothing fileName,
                      doCreateADT typeCon    Nothing fileName
                      ]
-- perform a list of refactorings.
seqRefac = seqRefac'.addFlagParam
 where
  addFlagParam [] = []
  addFlagParam (r:rs) = (r False): (map (\r'->r' True) rs)
  seqRefac' []= return ()
```

```
  seqRefac' (r:rs) = do r; seqRefac' rs

doAddFieldLabels typeCon info fileName isSubRefactor
 = applyRefac' addFieldLabels' typeCon info fileName isSubRefactor

doAddDiscriminators typeCon info fileName isSubRefactor
 = applyRefac' addDiscriminators' typeCon info fileName isSubRefactor

doAddConstructors typeCon  info fileName isSubRefactor
 = applyRefac'  addCons' typeCon info fileName isSubRefactor

doElimNestedPatterns typeCon info fileName isSubRefactor
 = applyRefac' elimNestedPatterns' typeCon info fileName isSubRefactor

doElimPatterns typeCon info fileName isSubRefactor
 = applyRefac' elimPatterns' typeCon info fileName isSubRefactor

doCreateADT typeCon info fileName isSubRefactor
 = applyRefac' createADT' typeCon info fileName isSubRefactor

applyRefac' fun typeCon info fileName isSubRefactor
  = do info'@(_,_, mod,_) <- if isJust info then return (fromJust info)
                             else parseSourceFile fileName
       case findDataTypeDecl typeCon mod of
         Left errMsg -> do putStrLn errMsg
         Right decl  -> fun decl info' fileName isSubRefactor

-------------------------------------------------------------
-- Refactoring:
-- adding field labels to the identified data type declaration.
-------------------------------------------------------------
addFieldLabels fileName row col
 =do info@(_, _, mod, _) <- parseSourceFile fileName
     case locToTypeDecl fileName row col mod of
       Left errMsg -> putStrLn errMsg
       Right decl  -> addFieldLabels' decl info fileName False

addFieldLabels' decl modInfo@(inscps, _, _, _) fileName isSubRefactor
 = do clients <-clientModsAndFiles =<<fileNameToModName fileName
      clientInfo <- mapM parseSourceFile (map snd clients)
      let inscps' = concatMap inScopeInfo
                      (inscps:(map (\(a,_,_,_)->a) clientInfo))
          inscpNames = map (\(x,_,_,_)->x) inscps'
      r <- applyRefac (addFieldLabels'' inscpNames decl)
                           (Just modInfo) fileName
      writeRefactoredFiles isSubRefactor  [r]

addFieldLabels'' inscpNames decl (_, _, mod)
  = do let (decls1, decls2) = break (==decl) (hsModDecls mod)
```

```
       newDecl <- conDeclToRecDecl inscpNames decl
       decl'<-update decl newDecl decl
       return $ mod {hsModDecls=decls1++(decl':(tail decls2))}

-- Add field labels to each data constructor declaration.
conDeclToRecDecl inscpeNames = applyTP strategy
  where
  strategy = (full_buTP (idTP `adhocTP` inConDecl))

  inConDecl (decl@(HsConDecl loc is c i ts):: HsConDeclP)
    = do ts'<-createFieldLabels 1 i ts
         return (HsRecDecl loc is c i ts')
  inConDecl x = return x

  createFieldLabels val dataCon [] = return []
  createFieldLabels val dataCon (t:ts)
   = do let prefix = map toLower (pNTtoName dataCon)
        name <- mkNewName prefix inscpeNames  (Just val)
        let nextVal = ord (glast "createFieldLabels" name)- ord '0'+ 1
        ds' <- createFieldLabels nextVal dataCon ts
        return $ ([nameToPNT name], t):ds'
-------------------------------------------------------------------------
-- Refactoring: adding a discriminator function for each data
-- constructor declared in the identified data type declaration.
-------------------------------------------------------------------------
addDiscriminators fileName row col
 =do info@(inscps, exps, mod, toks) <- parseSourceFile fileName
     case locToTypeDecl fileName row col mod of
       Left errMsg -> putStrLn errMsg
       Right decl  -> addDiscriminators' decl info fileName False

addDiscriminators' decl info@(inscps, _, _, _) fileName isSubRefactor
 = do clients <-clientModsAndFiles =<<fileNameToModName fileName
      clientInfo <- mapM parseSourceFile (map snd clients)
      let inscps' = concatMap inScopeInfo
                    (inscps:(map (\(a,_,_,_)->a) clientInfo))
          existingNames = map (\(x,_,_,_)->x) inscps'
      r <- applyRefac (addDiscriminators'' inscpNames decl)
                      (Just info) fileName
      writeRefactoredFiles isSubRefactor  [r]

addDiscriminators'' existingNames
                    decl@(Dec (HsDataDecl _ c tp conDecls _)) (_,_,mod)
 =do let consWithDiscrs = existingDiscriminators mod decl
     if (length conDecls == length consWithDiscrs)
      then return mod
      else do let conDecls' = filter (\x->isNothing (find
                      conName' x==) (map fst consWithDiscrs))) conDecls
              funs <- mapM (mkDiscriminator  tp mod) conDecls'
```

```
            addDecl mod Nothing (concat funs,Nothing) True
   where
   mkDiscriminator tp mod (decl@(HsRecDecl _ _ _ i ts):: HsConDeclP)
    = mkDiscriminator' tp mod i ts
   mkDiscriminator tp mod
                  (decl@(HsConDecl _ _ _ i ts):: HsConDeclP)
    = mkDiscriminator' tp mod i ts

   -- compose a discriminator function.
   mkDiscriminator' tp mod i ts
    =do newName <- mkNewName ("is"++(pNTtoName i))
                             existingNames (Just 0)
        let funNamePNT =nameToPNT newName
            typeSig = (Dec (HsTypeSig loc0 [funNamePNT] [](Typ
                (HsTyFun tp (Typ (HsTyCon (nameToPNT "Bool")))))))
            match1 =let pats= if length ts ==0
                              then [Pat (HsPId (HsCon i))]
                              else [Pat (HsPParen (Pat (HsPApp i
                                        (mkWildCards (length ts)))))]
                    in (HsMatch loc0  funNamePNT  pats
                                (HsBody (nameToExp "True")) [])
                match2 = HsMatch loc0 funNamePNT [Pat HsPWildCard]
                                 (HsBody (nameToExp "False")) []
                fun  = Dec (HsFunBind loc0 [match1, match2])
        return $ ([typeSig, fun])

   mkWildCards 0 = []
   mkWildCards n = (Pat HsPWildCard) : mkWildCards (n-1)

-- Collect the exisiting discriminator functions associated with
-- the specified data type.
existingDiscriminators mod (Dec (HsDataDecl _ c tp conDecls _))
  = filter (\x -> isJust (snd x))
       $ map (findDiscriminator (hsModDecls mod)) conDecls
  where
  findDiscriminator decls (conDecl:: HsConDeclP)
   = let (dataCon,numOfFields)
           = case conDecl of
               (HsRecDecl _ _ _ i ts) -> (pNTtoPN i, length ts)
               (HsConDecl _ _ _ i ts) -> (pNTtoPN i, length ts)
         decls' = filter (\x -> isDiscriminator x
                                (dataCon, numOfFields)) decls
     in if decls'==[]
         then (dataCon, Nothing)
         else (dataCon, Just (ghead "findDiscriminator"
                              (definedPNs (head decls'))))

  -- Return True if a function is a discriminator function
  -- for the specified data constructor.
```

```
  isDiscriminator (Dec (HsFunBind _ [m1@(HsMatch _ _ [p] _ _),m2]))
                       (dataCon,numOfFields)
   = case getConAndArity p of
       Just (con, arity) -> (con, arity) == (dataCon, numOfFields)
                            && (prettyprint (rhsExp m1))== "True"
                            && prettyprint (rhsExp m2)  == "False"
       _                 ->  False
   where
   getConAndArity (Pat (HsPApp i ps)) = Just ((pNTtoPN i), length ps)
   getConAndArity (Pat (HsPId (HsCon i))) = Just ((pNTtoPN i),0)
   getConAndArity (Pat (HsPParen p)) = getConAndArity p
   getConAndArity (Pat (HsPAsPat i p)) = getConAndArity p
   getConAndArity  _  = Nothing

   rhsExp (HsMatch _ _ _ (HsBody e) _) = Just e
   rhsExp _ = Nothing

   isDiscriminator _  _ = False

-------------------------------------------------------------------------
-- Refactoring: adding a constructor function for each data constructor
-- declared in the identified data type declaration.
-------------------------------------------------------------------------
addConstructors fileName row col
 =do info@(_,_,mod,_) <- parseSourceFile fileName
     case locToTypeDecl fileName row col mod of
       Left errMsg -> putStrLn errMsg
       Right decl  -> addCons' decl info fileName False

addCons' decl info@(inscps, _, _, _) fileName isSubRefactor
 = do clients <-clientModsAndFiles =<<fileNameToModName fileName
      clientInfo <- mapM parseSourceFile (map snd clients)
      let inscps' = concatMap inScopeInfo
                      (inscps:(map (\(a,_,_,_)->a) clientInfo))
          inscpNames = map (\(x,_,_,_)->x) inscps'
      r <- applyRefac (addCons'' inscpNames decl) (Just info) fileName
      writeRefactoredFiles isSubRefactor  [r]

addCons'' existingNames
          decl@(Dec (HsDataDecl _ c tp conDecls _))(_, _, mod)
 = do let consWithConstrs = existingCons mod decl
      if (length conDecls == length consWithConstrs)
       then return mod
       else do let conDecls' = filter (\x->isNothing (find
                      (conName' x==)(map fst consWithConstrs))) conDecls
               funs <- mapM mkCon conDecls'
               addDecl mod Nothing (concat funs,Nothing) True
  where
  mkCon (decl@(HsRecDecl _ _ _ i ts):: HsConDeclP)
```

```
    = mkCon' i (map (typeFromBangType.snd) ts)
  mkCon (decl@(HsConDecl _ _ _ i ts):: HsConDeclP)
    = mkCon' i (map typeFromBangType ts)

  mkCon' i ts
   = do newName <- mkNewName ("mk" ++ (pNTtoName i))
                               existingNames (Just 0)
        let funNamePNT =nameToPNT newName
        numOfParams = length ts
        typeSig = (Dec (HsTypeSig loc0 [funNamePNT] []
                        (mkTypeFun tp ts)))
        fun  = (Dec (HsFunBind loc0 [HsMatch loc0 funNamePNT []
                            (HsBody (pNtoExp (pNTtoPN i))) []]))
        return $ ([typeSig, fun])

  mkTypeFun t ts = foldr (\ t1 t2 ->(Typ (HsTyFun t1 t2))) t ts

  typeFromBangType (HsBangedType t)   = t
  typeFromBangType (HsUnBangedType t) = t

-- Fetch the existing constructor functions associated with
-- the specified data type declaration.
existingCons mod (Dec (HsDataDecl _ _ _ conDecls _))
  = filter (\x -> isJust (snd x))
           $ map (findCons (hsModDecls mod)) conDecls
  where
  findCons decls (conDecl:: HsConDeclP)
    = let (dataCon,numOfFields)
            = case conDecl of
              (HsRecDecl _ _ _ i ts) -> (pNTtoPN i, length ts)
              (HsConDecl _ _ _ i ts) -> (pNTtoPN i, length ts)
          decls' = filter (\x -> isCon x (dataCon, numOfFields)) decls
      in if decls'==[]
            then (dataCon, Nothing)
            else (dataCon,  Just (ghead "findCons"
                                  (definedPNs (head decls'))))

  isCon (Dec (HsFunBind _ [m1@(HsMatch _ _ ps (HsBody e) _ )]))
                    (dataCon,numOfFields)
   |length ps== numOfFields && all isVarPat ps
   = hsPNs e == dataCon : (hsPNs ps)
  isCon (Dec (HsPatBind _  _ (HsBody e) _)) (dataCon, _)
    = prettyprint e == pNtoName dataCon
  isCon _ _ = False

-----------------------------------------------------------------------
-- Refactoring: eliminating nested patterns, i.e. removing uses of the
-- ``other'' data constructors inside data constructors declared by
-- the specified data type.
```

```
-----------------------------------------------------------------------

elimNestedPatterns fileName row col
 =do info@(_, _, mod, _) <- parseSourceFile fileName
     case locToTypeDecl fileName row col mod of
       Left errMsg -> putStrLn errMsg
       Right decl  -> elimNestedPatterns' decl info fileName False

elimNestedPatterns' decl info fileName isSubRefactor
  = do r <- applyRefac elimNestedPats'' (Just info) fileName
       modName <- fileNameToModName fileName
       clients <- clientModsAndFiles modName
       rs <- applyRefacToClientMods elimNestedPats''
                                    Nothing (map snd clients)
       writeRefactoredFiles isSubRefactor $ (r:rs)
  where
  conNames = map pNTtoPN (conPNTs decl)

  elimNestedPats'' (_, _, mod) = applyTP strategy  mod
    where
    strategy = full_buTP (idTP `adhocTP` inMatch
                               `adhocTP` inExp)

    -- Eliminating nested patterns in the formal parameters
    -- of a function declaration.
    inMatch (m@(HsMatch loc i ps rhs ds)::HsMatchP)
      = do m' <-mkNewAST mkNewMatch conNames match ps
           if (m/=newMatch) then update m newMatch m
                            else return newMatch

    -- Eliminating nested patterns in expressions.
    inExp (exp@(Exp (HsLambda ps e))::HsExpP)
      = do newExp <-mkNewAST mkNewLambdaExp conNames exp ps
           if (exp/=newExp) then update exp newExp exp
                            else return newExp

    inExp exp@(Exp (HsListComp stmts))
     = do newExp <- applyTP (full_buTP (idTP `adhocTP` inStmt)) exp
          if (exp/=exp') then update exp newExp exp
                         else return newExp'
      where
       inStmt (stmt@(HsGenerator loc p e stmts)::HsStmtP)
         = mkNewAST mkNewListStmt conNames stmt p
       inStmt m = return m

    inExp exp@(Exp (HsDo stmts))
      = do exp' <-applyTP (full_buTP (idTP `adhocTP` inStmt)) exp
           if (exp/=exp') then update exp exp' exp
                          else return exp'
```

```
  where
   inStmt (stmt@(HsGenerator loc p e stmts)::HsStmtP)
     = mkNewAST mkNewDoStmt conNames stmt p
   inStmt m = return m

inExp exp@(Exp (HsCase e alts))
  = do exp'<-applyTP (full_buTP (idTP `adhocTP` inAlt)) exp
       if (exp/=exp') then update exp exp' exp
                      else return exp'
  where
   inAlt (alt@(HsAlt loc p rhs ds)::HsAltP)
     = mkNewAST mkNewAlt conNames alt p
inExp m = return m

mkNewAST fun conNames ast p
 = do fds <-existingVbls ast
      p' <-rmNestedPatternInParams fds conNames p
      resetVal
      varsAndPats <- getExpPatPairs fds conNames p
      resetVal
      return (fun ast p' varsAndPats)

mkNewDoStmt stmt@(HsGenerator loc p e stmts) p' varsAndPats
  = case varsAndPats of
      Nothing -> stmt
      Just (vars, pats) ->
       (HsGenerator loc p' e (HsLast (Exp (HsCase vars
       [HsAlt loc0 pats (HsBody (Exp (HsDo stmts))) []]))))

mkNewListStmt stmt@(HsGenerator loc p e stmts) p' varsAndPats
  = case varsAndPats of
      Nothing -> stmt
      Just (vars, pats)->
       (HsGenerator loc p' e
       (HsGenerator loc pats (Exp (HsList [vars])) stmts))

mkNewLambdaExp exp@(Exp (HsLambda ps e)) ps' varsAndPats
  = case varsAndPats of
      Nothing -> exp
      Just (vars,pats) ->
       (Exp (HsLambda ps' (Exp (HsCase vars
            [HsAlt loc0 pats (HsBody e) []]))))

mkNewAlt alt@(HsAlt loc p rhs@(HsBody e) ds) p' expsAndPats
  = case expsAndPats of
      Nothing -> alt
      Just (exps, pats)->
       (HsAlt loc p' (HsGuard [(loc0, (Exp (HsCase exps
        [HsAlt loc0  pats (HsBody fakeTrueExp) [],
```

```
         HsAlt loc0 (Pat HsPWildCard) (HsBody fakeFalseExp) []])),
        (Exp (HsApp (Exp (HsParen
                      (Exp (HsLambda [pats] e)))) exps)))]) ds)

mkNewAlt alt@(HsAlt loc p rhs@(HsGuard es) ds) p' expsAndPats
  = case expsAndPats of
      Nothing -> alt
      Just (exps, pats) ->
        let rhs'= HsGuard $ map (addToGuards (exps, pats)) es
        in (HsAlt loc p' rhs' ds)

mkNewMatch ((HsMatch loc i ps rhs@(HsBody e) ds)::HsMatchP)
           ps' expsAndPats
  = case expsAndPats of
      Nothing -> HsMatch loc i ps' rhs ds
      Just (exps, pats) ->
        (HsMatch loc i ps' (HsGuard [(loc0, (Exp (HsCase exps
         [HsAlt loc0  pats (HsBody (nameToExp "True")) [],
          HsAlt loc0 (Pat HsPWildCard)
                     HsBody (nameToExp "False")) []])),
         (Exp (HsApp (Exp (HsParen
                     (Exp (HsLambda [pats] e)))) exps)))]) ds)

mkNewMatch (HsMatch loc i ps rhs@(HsGuard es) ds) ps' expsAndPats
  = case expsAndPats of
     Nothing-> HsMatch loc i ps' rhs ds
     Just (exps, pats)->
       let rhs'= HsGuard $ map (addToGuards (exps,pats)) es
       in (HsMatch loc i ps' rhs' ds)

addToGuards (exp, pat) (loc, e1, e2)
  = let g1 = Exp (HsCase exp
             [HsAlt loc0  pat (HsBody (nameToExp "True")) [],
              HsAlt loc0 (Pat HsPWildCard)
                         (HsBody (nameToExp "False")) []])
        e1'= Exp (HsInfixApp g1 (HsVar (nameToPNT "&&")) e1)
        e2'= Exp (HsApp (Exp (HsParen
                         (Exp (HsLambda [pat] e2)))) exp)
    in (loc, e1', e2')


getExpPatPairs d conNames ps
  = do r <- applyTU (stop_tdTU (failTU `adhocTU` inPat)) ps
       let r' = filter (\(x,y) -> isJust x) r
           r''= map (\(x,y)->(patVarToExpVar (fromJust x),y)) r'
           (exps, pats) = (map fst r'', map snd r'')
           result = if lenght exps ==0 the Nothing
                    else if length exps == 1
                           then Just (head exps, head pats)
```

```
                              else Just (Exp (HsTuple exps),
                                         Pat (HsPTuple pats)))
        return r
  where
  inPat (pat@(Pat (HsPApp i is))::HsPatP)
    | isJust (find (==pNTtoPN i) conNames)
    = do  is'<- collectVarsAndPats d conNames is
          let caseExps = filter (\(x,y) ->isJust x) is'
          return caseExps
  inPat _ = mzero

  patVarToExpVar (Pat (HsPId (HsVar id)))
    = (Exp (HsId (HsVar id)))

collectVarsAndPats d conNames
  = applyTU (stop_tdTU (failTU `adhocTU` inAppPat))
  where
  inAppPat pat
    | isVarPat pat = return [(Nothing, pat)]
  inAppPat pat@(Pat (HsPApp i ps))
    |isNothing (find (==pNTtoPN i) conNames)
    = replacePatByVar pat Nothing
  inAppPat pat@(Pat (HsPAsPat i1 (Pat (HsPApp i2 ps))))
    |isNothing (find (==pNTtoPN i2) conNames)
    = replacePatByVar pat (Just (pNTtoName i1))
  inAppPat pat@(Pat (HsPInfixApp _ (HsCon i) _))
    |isNothing (find (==pNTtoPN i) conNames)
    = replacePatByVar pat Nothing
  inAppPat pat@(Pat (HsPAsPat i1
                     (Pat (HsPInfixApp _ (HsCon i2) _))))
    |isNothing (find (==pNTtoPN i2) conNames)
    = replacePatByVar pat (Just (pNTtoName i1))
  inAppPat pat@(Pat (HsPRec i _))
    |isNothing (find (==pNTtoPN i) conNames)
    = replacePatByVar pat Nothing
  inAppPat pat@(Pat (HsPAsPat i1 (Pat (HsPRec i2 _))))
    |isNothing (find (==pNTtoPN i2) conNames)
    = replacePatByVar pat (Just (pNTtoName i1))
  inAppPat pat
    = replacePatByVar pat Nothing

  replacePatByVar pat (Just varName)
    = return [(Just (nameToPat varName), pat)]
  replacePatByVar pat Nothing
    = do (_,d') <- hsFreeAndDeclaredPNs pat
         var <- mkNewName "p" (map pNtoName (d\\d')) Nothing
         return [(Just (nameToPat var), pat)]

rmNestedPatternInParams d conNames
```

```
      = applyTP (stop_tdTP (failTP `adhocTP` inPat))
      where
      inPat (pat@(Pat (HsPApp i is))::HsPatP)
        | isJust (find (==pNTtoPN i) conNames)
        = do is'<-rmNestedPatternInParams' is
             return (Pat (HsPApp i is'))
      inPat _ = mzero

      -- replace the nested patterns by variables.
      rmNestedPatternInParams'
        = applyTP (stop_tdTP (failTP `adhocTP` inAppPat))
        where
        inAppPat pat
          | isVarPat pat = mzero
        inAppPat pat@(Pat (HsPApp i ps))
          |isNothing (find (==pNTtoPN i) conNames)
          = replacePatByVar pat Nothing
        inAppPat pat@(Pat (HsPAsPat i1 (Pat (HsPApp i2 ps))))
          |isNothing (find (==pNTtoPN i2) conNames)
          = replacePatByVar pat (Just (pNTtoName i1))
        inAppPat pat@(Pat (HsPInfixApp _ (HsCon i) _))
          |isNothing (find (==pNTtoPN i) conNames)
          = replacePatByVar pat Nothing
        inAppPat pat@(Pat (HsPAsPat i1 (Pat
                          (HsPInfixApp _ (HsCon i2) _))))
          |isNothing (find (==pNTtoPN i2) conNames)
          = replacePatByVar pat (Just (pNTtoName i1))
        inAppPat pat@(Pat (HsPRec i _))
          |isNothing (find (==pNTtoPN i) conNames)
          = replacePatByVar pat Nothing
        inAppPat pat@(Pat (HsPAsPat i1 (Pat (HsPRec i2 _))))
          |isNothing (find (==pNTtoPN i2) conNames)
          = replacePatByVar pat (Just (pNTtoName i1))
        inAppPat pat
          = replacePatByVar pat Nothing

        replacePatByVar pat (Just varName)
          = update pat (nameToPat varName) pat
        replacePatByVar pat Nothing
          = do (_,d') <- hsFreeAndDeclaredPNs pat
               newName <- mkNewName "p" (map pNtoName (d\\d')) Nothing
               update pat (nameToPat newVarName) pat
-------------------------------------------------------------------------
-- Refactoring: eliminating patterns, i.e. eliminating the explicit uses
-- of data constructors defined by the specified data type declaration.
-------------------------------------------------------------------------
elimPatterns fileName row col
  = do info@(_,_,mod,_) <- parseSourceFile fileName
       case locToTypeDecl fileName row col mod of
```

```
        Left errMsg -> putStrLn errMsg
        Right decl  -> do elimPatsCondChecking decl mod
                          elimPatterns' decl info fileName False
  where
  -- Condition checking.
  elimPatsCondChecking decl@(Dec (HsDataDecl _ _ _ con _)) mod
    = do findFun conNames sels
               "Selector does not exist for data constructor: "
         findFun conNames discrs
               "Discriminator not exis for data constructor: "
         findFun conNames cons
               "Constructor does not exist for data constructor: "
    where
    conNames = map pNTtoPN $ conPNTs decl

    findFun conNames funs errMsg
      = mapM (flip findFun' funs) conNames
      where
      findFun' conName funs
        = let r = find (\(x,y)->x==conName) funs
          in if isNothing r  then error $ errMsg++ pNtoName conName
                             else return $ (snd.fromJust) r

    sels = map selectors con
      where
      selectors ((HsRecDecl _ _ _ i ts):: HsConDeclP)
        = (pNTtoPN i, map pNTtoName (concatMap fst ts))
      selectors ((HsConDecl _ _ _ i _)::HsConDeclP)
        = (pNTtoPN i, [])

    discrs = map (\(x,y)->(x, pNtoName (fromJust y)))
             $  existingDiscriminators mod decl

    cons  = map (\(x,y) -> (x, pNtoName (fromJust y)))
            $ existingCons mod decl

elimPatterns' decl@(Dec (HsDataDecl _ _ _ con _))
              info@(_, exps, mod, _) fileName isSubRefactor
  = do r <- applyRefac (elimPatternsInMod decl) (Just info) fileName
       if (any (flip isExported  exps) (conPNTs decl))
        then do modName <- fileNameToModName fileName
                clients <- clientModsAndFiles modName
                rs <- applyRefacToClientMods (elimPatsInClients modName)
                                             Nothing (map snd clients)
                writeRefactoredFiles isSubRefactor (r:rs)
        else writeRefactoredFiles isSubRefactor $ [r]
  where
  typeCon  = fromJust $ getTypeCon decl
```

```
conNames = map pNTtoPN $ conPNTs decl

sels = map selectors con
  where
  selectors ((HsRecDecl _ _ _ i ts):: HsConDeclP)
    = (pNTtoPN i, map pNTtoName (concatMap fst ts))
  selectors ((HsConDecl _ _ _ i _)::HsConDeclP)
    = (pNTtoPN i, [])

discrs = map (\(x,y)->(x, pNtoName (fromJust y)))
          $ existingDiscriminators mod decl

cons  = map (\(x,y) -> (x, pNtoName (fromJust y)))
         $ filter (\(x,y)->isJust y) $ existingCons mod decl

elimPatternsInMod decl (_, exps , mod)
  = rmPatterns =<< doCreateADTInterface True decl mod exps

elimPatsInClients serverModName (_, _, mod)
  = do --replace exports of data constructors by exports of functions.
       let entsToAdd= map (\x-> pNtoVarEnt (nameToPN x))(findEntsToAdd
                           typeCon sels discrs cons (exportedEnts mod)
       mod'<- addItemsToExport mod Nothing False (Right entsToAdd)
       --replace imports of data constructors by imports of functions.
       mod''<-addToImport serverModName typeCon sels  discrs cons mod'
       rmPatterns mod''

rmPatterns = applyTP (full_tdTP (idTP `adhocTP` inMatch
                                      `adhocTP` inPatBinding
                                      `adhocTP` inExp))
  where
  -- Remove patterns in the formal parameters of a function.
  inMatch (match@(HsMatch loc i ps rhs ds)::HsMatchP)
   |isNothing (find (==(pNTtoName i)) (map snd (discrs++cons)))
   = do (ps', sels, guards) <-rmPatternsInParams conNames match ps
        r'<-replaceVarsBySels sels rhs
        r''<-replacePatsByCons conNames cons r'
        r'''<-if guards/=[]
               then addGuardsToRhs r'' $ fromJust (mkGuard guards)
               else return rhs''
        ds'<- replaceVarsBySels sels ds
        ds'<-replacePatsByCons conNames cons ds'
        return $ (HsMatch loc i ps' r''' ds')
  inMatch m = return m

  -- Remove patterns in the LHS of a pattern binding.
  inPatBinding (pat@(Dec (HsPatBind loc i rhs ds)::HsDeclP))
    |isNothing (find (==(pNTtoName.patToPNT) i)
                     (map snd (discrs++cons)))
```

```
  = do rhs'<-replacePatsByCons conNames cons rhs
       ds'<-replacePatsByCons conNames cons  ds
       return $ (Dec (HsPatBind loc i rhs' ds'))
inPatBinding m = return m

-- Remove patterns in expressions.
inExp (exp@(Exp (HsLambda ps e))::HsExpP)
  = do (ps',sels,guards) <-rmPatternsInParams conNames exp ps
       e'<-replaceVarsBySels sels e
       let e''= if guards/=[] then
                  (Exp (HsCase (fromJust (mkGuard guards))
                   [HsAlt loc0 (nameToPat "True") (HsBody e') []]))
                else e'
           exp'= Exp (HsLambda ps' e'')
       if (exp/=exp') then update exp exp' exp
                      else return exp'
inExp exp@(Exp (HsListComp stmts))
  = do exp'<-applyTP (full_buTP (idTP `adhocTP` inStmt)) exp
       if (exp/=exp') then update exp exp' exp
                      else return exp'
  where
  inStmt (stmt@(HsGenerator loc p e stmts)::HsStmtP)
   = do (p', sels, guards) <-rmPatternsInParams conNames stmt p
        e'<-replaceVarsBySels sels e
        stmts'<-replaceVarsBySels sels stmts
        let stmts''=
             if guards/=[]
              then (HsQualifier (fromJust (mkGuard guards)) stmts')
              else stmts'
        return $ HsGenerator loc p' e' stmts''
  inStmt m = return m

inExp exp@(Exp (HsDo stmts))
  = do exp' <-applyTP (full_buTP (idTP `adhocTP` inStmt)) exp
       if (exp/=exp') then update exp exp' exp
                      else return exp'
  where
  inStmt (stmt@(HsGenerator loc p e stmts)::HsStmtP)
    = do (p', sels, guards) <-rmPatternsInParams conNames stmt p
         e'<-replaceVarsBySels sels e
         stmts'<-replaceVarsBySels sels stmts
         let stmts'' =
              if guards/=[] then
                 (HsLast (Exp (HsCase (fromJust (mkGuard guards))
                  [HsAlt loc0 (nameToPat "True")
                       (HsBody (Exp (HsDo stmts'))) []])))
                else stmts'
         return (HsGenerator loc p' e' stmts'')
  inStmt m = return m
```

```
inExp exp@(Exp (HsCase e alts))
  = do exp'<-applyTP (full_buTP (idTP `adhocTP` inAlt)) exp
       if (exp/=exp') then update exp exp' exp
                      else return exp'
  where
  inAlt (alt@(HsAlt loc p rhs ds)::HsAltP)
    = do (p', sels, guards) <-rmPatternsInParams conNames alt p
         r'<- replaceVarsBySels sels rhs
         r''<-if guards/=[]
               then addGuardsToRhs r' (fromJust (mkGuard guards))
               else return rhs'
         ds'<-replaceVarsBySels sels ds
         return (HsAlt loc p' r'' ds')
inExp m = return m

rmPatternsInParams conNames ast ps
  = do fds <-existingVbls ast
       ps' <- collectPatsInParams fds conNames ps
       resetVal
       (sels, guards) <-rmPatternInParams fds conNames ps
       resetVal
       return (ps',sels, guards)

collectPatsInParams d conNames
  = applyTP (stop_tdTP (failTP `adhocTP` inPat))
  where
  inPat pat@(Pat (HsPParen (Pat (HsPApp i is))))
    | isJust (find (==pNTtoPN i) conNames)
    = inPat' pat Nothing
  inPat pat@(Pat (HsPAsPat i1 (Pat (HsPParen
                       (Pat (HsPApp i2 is))))))
    | isJust (find (==pNTtoPN i2) conNames)
    = inPat' pat (Just (pNTtoName i1))
  inPat (pat@(Pat (HsPApp i is))::HsPatP)
    | isJust (find (==pNTtoPN i) conNames)
    = inPat' pat Nothing
  inPat pat@(Pat (HsPAsPat i1 (Pat (HsPApp i2 is))))
    | isJust (find (==pNTtoPN i2) conNames)
    = inPat' pat (Just (pNTtoName i1))
  inPat pat@(Pat (HsPId (HsCon i)))
    | isJust (find (==pNTtoPN i) conNames)
    = inPat' pat Nothing
  inPat pat@(Pat (HsPParen (Pat (HsPId (HsCon i)))))
    | isJust (find (==pNTtoPN i) conNames)
    = inPat' pat Nothing
  inPat pat@(Pat (HsPAsPat i1 (Pat (HsPId (HsCon i2)))))
    | isJust (find (==pNTtoPN i2) conNames)
    = inPat' pat (Just (pNTtoName i1))
```

```
inPat pat@(Pat (HsPAsPat i1 (Pat (HsPParen
                      (Pat (HsPId (HsCon i2)))))))
  | isJust (find (==pNTtoPN i2) conNames)
  = inPat' pat (Just (pNTtoName i2))

inPat _ = mzero

inPat' pat varName
  = do unless (not (isNestedPattern conNames pat))
                   $ error "Nested patterns exist!"
       (_,d') <- hsFreeAndDeclaredPNs pat
       newVarName <- if isJust varName
                     then return (fromJust varName)
                     else mkNewName "p" (map pNtoName
                                         (d \\d')) Nothing
       update pat (nameToPat newVarName) pat

rmPatternInParams d conNames ps
  = do r<-applyTU (stop_tdTU (failTU `adhocTU` inPat)) ps
       return (concatMap fst r, concatMap snd r)
  where
  inPat (pat@(Pat (HsPApp i is))::HsPatP)
    | isJust (find (==pNTtoPN i) conNames)
    = inPat' (pNTtoPN i) pat Nothing

  inPat (pat@(Pat (HsPId (HsCon i)))::HsPatP)
    | isJust (find (==pNTtoPN i) conNames)
    = inPat' (pNTtoPN i) pat Nothing

  inPat (pat@(Pat (HsPAsPat i1 i2)))
    | case rmPParen i2 of
       (Pat (HsPApp i2' _)) ->
                 isJust (find (==pNTtoPN i2') conNames)
       (Pat (HsPId (HsCon i2')))->
                 isJust (find (==pNTtoPN i2') conNames)
       _ ->False
    = (\pat -> case pat of
                (Pat (HsPApp i2' ps))->
                  inPat' (pNTtoPN i2') pat (Just (pNTtoName i1))
                (Pat (HsPId (HsCon i2')))->
                  inPat' (pNTtoPN i2') pat
                         (Just (pNTtoName i1))) (rmPParen i2)
  inPat _ =mzero

  inPat' conPN pat varName
    = do (_,d') <- hsFreeAndDeclaredPNs pat
         newName <-
           if isJust varName
            then return (fromJust varName)
```

```
            else mkNewName "p" (map pNtoName (d\\d')) Nothing
          let sels' = snd.fromJust
                      $ find (\(x,y) -> x==conPN) sels
              selFuns =mkSelFuns conPN newName "" sels' pat
              guardExps = mkGuardExps newName discrs pat
          return [(selFuns, guardExps)]

isNestedPattern conNames appPat
  = isJust $ find (==True) $ head $ applyTU strategy appPat
  where
  strategy = full_tdTU (constTU []  `adhocTU` inAppPat)

  inAppPat pat | isVarPat pat =return []
  inAppPat pat@(Pat (HsPApp i ps))
    |isJust (find (==pNTtoPN i) conNames)
    = return []
  inAppPat pat@(Pat (HsPId (HsCon i)))
    |isJust (find (==pNTtoPN i) conNames)
    =return []
  inAppPat pat@(Pat (HsPInfixApp _ (HsCon i) _))
    |isJust (find (==pNTtoPN i) conNames)
    = return []
  inAppPat pat@(Pat (HsPRec i _))
    |isJust (find (==pNTtoPN i) conNames)
    = return []
  inAppPat pat@(Pat (HsPAsPat i1 i2))
    = applyTU strategy i2
  inAppPat pat@(Pat (HsPParen p))
    = applyTU strategy p
  inAppPat _ = return [True]

replaceVarsBySels [] p = return p
replaceVarsBySels sels p = applyTP strategy p
  where
  strategy = full_buTP (idTP `adhocTP` inExp)
  inExp exp@(Exp (HsId (HsVar (PNT pn _ _))))
    |isJust (find (==pn) (map fst sels))
    = do let sel = (snd.fromJust)
                   (find (\(x,y)-> x==pn) sels)
         update exp sel exp
  inExp e = return e

replacePatsByCons conNames constrs
  = applyTP (full_tdTP (idTP `adhocTP` inExp))
  where
  inExp exp@(Exp (HsRecConstr i fields))
    | isJust (find (==pNTtoPN i) conNames)
    = do let con = (snd.fromJust)
                   $ find (\(x,y)->x == pNTtoPN i) constrs
```

```
            es = map (\(HsField _ e ) -> e) fields
            exp'= (Exp (HsParen (foldl (\e1 e2->(Exp
                   (HsApp e1 e2))) (nameToExp con) es)))
        update exp exp' exp
  inExp e
    | isJust (find (==(pNTtoPN.expToPNT) e) conNames)
    = do let (PNT pn ty src)=expToPNT e
            con = (snd. fromJust)
                   $ find (\(x,y)-> x==pn) consts
         renamePN pn Nothing con True e

  inExp e = return e

mkGuardExps newVarName discrs (Pat (HsPApp i ps))
  = let discr = snd.fromJust
                $ find (\(x,y) -> x== pNTtoPN i) discrs
        g = Exp (HsApp (Exp (HsId (HsVar (nameToPNT discr))))
                (Exp (HsId (HsVar (nameToPNT newVarName)))))
    in [g]

mkGuardExps newVarName discrs (Pat (HsPId (HsCon i)))
  = let discr = snd.fromJust
                $ find (\(x,y) -> x==pNTtoPN i)  discrs
        g = Exp (HsApp (Exp (HsId (HsVar (nameToPNT discr))))
                (Exp (HsId (HsVar (nameToPNT newVarName)))))
    in [g]

mkGuard [] = Nothing
mkGuard (e:es)
  = Just (foldl (\e1 e2 -> (Exp (HsInfixApp e1 (HsVar
                           (nameToPNT  "&&")) e2))) e es)

mkSelFuns con  newName posfix selectors (Pat (HsPApp i ps))
  = concatMap mkSelFuns' (zip selectors ps)
  where
  mkSelFuns' (sel, pat@(Pat (HsPId (HsVar i))))
    = let f = if posfix ==[]
              then (Exp (HsParen (Exp (HsApp (nameToExp sel)
                                   (nameToExp newName)))))
              else (Exp (HsParen (Exp (HsApp (nameToExp
                    (sel++"."++posfix))(nameToExp newName))))))]
      in [(pNTtoPN i, f)]
  mkSelFuns' (sel, (Pat (HsPParen p)))
    = mkSelFuns' (sel, p)
  mkSelFuns' (sel, pat@(Pat (HsPApp i ps)))
    = mkSelFuns con newName
       (if posfix=="" then sel
          else sel++"."++posfix) selectors pat
  mkSelFuns' (sel, _) = []
```

```
      mkSelFuns _ _ _ _  _ = []

addToImport serverModName typeCon sels discrs constrs mod
  =applyTP (full_buTP (idTP `adhocTP` inImport)) mod
  where
  inImport (imp@(HsImportDecl _ (SN modName _) _ _ h)::HsImportDeclP)
    | serverModName == modName &&  findPN typeCon h
    = case h of
       Nothing        -> return imp
       Just (b, ents) ->
        do let funs=findEntsToAdd typeCon sels discris constrs ents
           if (funs==[]) then return imp
            else addItemsToImport serverModName Nothing (Left funs) imp
  inImport imp = return imp

-------------------------------------------------------------------------
-- Refactoring: creating the ADT module interface.
-------------------------------------------------------------------------
createADTMod fileName row col
  = do info@(_,_,mod,_) <- parseSourceFile fileName
       clients <- clientModsAndFiles =<< fileNameToModName fileName
       clientInfo <- mapM parseSourceFile (map snd clients)
       case locToTypeDecl fileName row col mod of
        Left errMsg -> putStrLn errMsg
        Right decl  -> do condChecking decl clientInfo
                          createADT' decl info fileName False
  where
  condChecking decl clientInfo
    = let r = findPNs (conName decl)
              (map (\ (_, _, mod,_) -> hsModDecls mod) clientInfo)
      in case r of
          True -> error ("Some of the data constructors declared by "
                         ++ "this datatype are used by at least one "
                         ++ " of the client modules! ")
          False -> return ()

createADT' decl info fileName isSubRefactor
  = do r <-applyRefac (doCreateADTMod decl) (Just info) fileName
       let typeCon = fromJust $ getTypeCon decl
       clients <-clientModsAndFiles =<< fileNameToModName fileName
       clientInfo <- mapM parseSourceFile (map snd clients)
       rs <- applyRefacToClientMods (createADTInClients typeCon)
                             (Just clientInfo) (map snd clients)
       writeRefactoredFiles isSubRefactor $ (r:rs)


createADTInClients typeCon (_, _, mod)
  = rmSubEntsFromExport typeCon mod
```

```
doCreateADTMod decl@(Dec (HsDataDecl _ _ _ cons _)) (_, exps, mod)
  = doCreateADTInterface False decl mod exps


doCreateADTInterface forElimPat
                     decl@(Dec (HsDataDecl _ _ _ cons _)) mod exps
  = addToExport (typeCon:(sels++discs++constrs))
  where
  typeCon = fromJust $ getTypeCon decl

  conNames = map pNTtoPN $ conPNTs decl

  discs = map (fromJust.snd) $ existingDiscriminators mod decl

  constrs = map (fromJust.snd) $ existingCons mod decl

  sels = concatMap sels' cons
    where
    sels' ((HsRecDecl _ _ _ i ts):: HsConDeclP)
      = map pNTtoPN (concatMap fst ts)
    sels' ((HsConDecl _ _ _ i _) :: HsConDeclP)
      = []

addToExport pns
 =let filteredExps = map fromJust
                      $ filter isJust (map fromEntToEntE exps)
      modName = ModuleE $ hsModName mod
  in case hsModExports mod of
      Nothing  -> if forElimPat then return mod
                  else addItemsToExport  mod Nothing
                        True (Right entsToBeExported)
      -- There are explicitly exported entities.
      Just exports ->
       case isJust (find (==modName) exports) of
            -- The whole module is implicity exported.
         True-> if forElimPat then return mod
                 else do let e = [sNtoName (hsModName mod)]
                         mod' <- rmItemsFromExport mod (Left (e, pns))
                         addItemsToExport mod' Nothing True
                                     (Right entsToBeExported)
            -- Individual entities are explicitly exported
         False -> do let entsToAdd = map pNtoVarEnt
                                    $ (pns \\(findEnts pns exports))
                     addItemsToExport mod Nothing False (Right entsToAdd)
    where
    entsToBeExported
      = nub $ map fromJust
            $ filter isJust (map fromEntToEntE exps)
```

```
    fromEntToEntE  (_, Ent modName (HsCon con) (Type _))
      = if sNtoName con == pNtoName typeCon
         then Just $ EntE (Abs (nameToPNT (sNtoName con)))
         else Just $ EntE (AllSubs (nameToPNT (sNtoName con)))
    fromEntToEntE  (_, Ent modName (HsVar var) _)
      = Just $ EntE (Var (nameToPNT (sNtoName  var)))
    fromEntToEntE  _   = Nothing

    findEnts pns ents
      = filter (\pn->any (\e->case e of
                           ModuleE _ -> False
                           EntE e'   -> match pn e') ents) pns
      where  match pn (Var pnt)        = pNTtoPN pnt == pn
             match pn (Abs pnt)        = pNTtoPN pnt == pn
             match pn (AllSubs pnt)    = pNTtoPN pnt == pn
             match pn (ListSubs pnt _) = pNTtoPN pnt == pn

exportedEnts (HsModule _ _ (Just ents) _ _)
  = map fromJust $ filter isJust
      ( map (\e ->case e of
                  (EntE ent) ->Just ent
                  _          ->Nothing) ents)
exportedEnts (HsModule _ _ Nothing _ _) = []

findEntsToAdd typeConPN sels discrs constrs ents
  = nub $ concatMap (match typeConPN) ents
  where
  match typeConPN (AllSubs pnt)
    | pNTtoPN pnt == typeConPN
    = concatMap snd sels ++ map snd (discrs++constrs)
  match typeConPN (ListSubs pnt idents)
    | pNTtoPN pnt == typeConPN
    = let r1 = concatMap (\(dataCon, funs) ->
                if elem (pNtoName dataCon) identNames
                 then funs
                 else []) sels
          r2 = concatMap (\(dataCon, fun) ->
                if elem (pNtoName dataCon) identNames
                 then [fun]
                 else []) (discris++constrs)
      in r1++r2
    where
    identNames= map identToName  idents
    identToName (HsVar i) = pNTtoName i
    identToName (HsCon i) = pNTtoName i
   match _ _ = []

-- From textual selection to the internal (AST) representation
-- of the data type declaration.
```

```
locToTypeDecl::String->Int->Int->HsModuleP->Either String HsDeclP
locToTypeDecl fileName row col mod
  = case locToTypeDecl' of
      Nothing   ->Left "Invalid cursor position. " ++
                       "Please place cursor at the beginning " ++
                       "of the type constructor name!"
      Just decl ->Right decl
  where
  locToTypeDecl'
    = find (definesTypeCon (locToPNT fileName (row,col) mod))
           (hsModDecls mod)

  definesTypeCon pnt (Dec (HsDataDecl loc c tp _ _))
    = isTypeCon pnt && (findPNT pnt tp)
  definesTypeCon pnt _ = False

-- From type constructor name to the internal (AST) representation
-- of the data type declaration.
findDataTypeDecl::String->HsModuleP->Either String HsDeclP
findDataTypeDecl typeCon mod
  = case find definesTypeCon (hsModDecls mod) of
      Nothing   -> Left "Datatype declaration can not be found!"
      Just decl'-> Right decl'
  where
  definesTypeCon (Dec (HsDataDecl loc c tp _ _))
    = typeCon == pNTtoName (head (hsPNTs tp))
  definesTypeCon _ = False

-- Fetch the declared type constructor.
getTypeCon::HsDeclP->Maybe PName
getTypeCon decl@(Dec (HsDataDecl l c tp cons d))
  = Just $ pNTtoPN $ ghead "getTypeCon"
         $ filter (\(PNT _ (Type _) _)->True) (hsPNTs tp)
getTypeCon _ = Nothing

-- Get the PNT representation of data constructors
-- declared by a data type declarations.
conPNTs (Dec (HsDataDecl _ _ _ cons _))
  = map conPNT cons

conPNT (HsRecDecl _ _ _ i _) = i
conPNT (HsConDecl _ _ _ i _) = i
```

Appendix F

The HaRe API

The HaRe API, with a brief description of each function, is shown in this appendix. The complete documentation of this API is also available from our *Refactoring Functional Programs* project webpage [91]. In what follows, some frequently used type synonyms are given before the API.

F.1 Some Type Synonyms

```
data NameSpace
  = ValueName
  | ClassName
  | TypeCon
  | DataCon
  | Other

type HsDeclP = HsDeclI PNT

type HsPatP = HsPatI PNT

type HsExpP = HsExpI PNT

type HsMatchP = HsMatchI PNT HsExpP HsPatP [HsDeclP]

type HsModuleP = HsModuleI ModuleName PNT [HsDeclI PNT]

type HsImportDeclP = HsImportDeclI ModuleName PNT

type HsExportEntP = HsExportSpecI ModuleName PNT

type RhsP = HsRhs HsExpP
type GuardP = (SrcLoc, HsExpP, HsExpP)

type HsAltP = HsAlt HsExpP HsPatP [HsDeclP]
```

```
type HsStmtP = HsStmt HsExpP HsPatP [HsDeclP]

type HsFieldP = HsFieldI PNT HsExpP

type HsTypeP = HsTypeI PNT

type EntSpecP = EntSpec PNT

type HsConDeclP = HsConDeclI PNT HsTypeP [HsTypeP]

type ENT = Ent Id

type InScopes = Rel QName (Ent Id)

type Exports = [(Id, Ent Id)]

type SimpPos = (Int, Int)
```

F.2 Program Analysis Functions

F.2.1 Import and Export Analysis

```
-- Process the in-scope relation returned from the module analysis,
-- and return a list of four-element tuples. Each tuple contains an
-- identifier name, the identifier's namespace, the name of the module
-- in which the identifier is defined, and the identifier's qualifier.
inScopeInfo :: InScopes
            -> [(String, NameSpace, ModuleName, Maybe ModuleName)]

-- Process the export relation returned from the module analysis, and
-- return a list of  three-element tuples. Each tuple contains an
-- identifier name, the identifier's namespace, and the name of the
-- module in which the identifier is defined.
exportInfo :: Exports
           -> [(String, NameSpace, ModuleName)]

-- Return True if the identifier is exported.
isExported :: PNT -> Exports -> Bool

-- Return True if an identifier is explicitly exported by the module.
isExplictlyExported :: PName -> HsModuleP -> Bool

-- Return True if the module is exported by itself either by omitting
-- the export list or by specifying the module name in its export list.
modIsExported :: HsModuleP -> Bool
```

F.2.2 Variable Analysis

```
-- Collect the identifiers (in PName representation) in a syntax phrase.
hsPNs :: Term t => t -> [PName]

-- Collect the identifiers (in PNT representation) in a syntax phrase.
hsPNTs :: Term t => t -> [PNT]

-- Collect the data constructors that occur in a syntax phrase. The first
-- list in the result contains the data constructors declared in other
-- modules, and the second list contains the data constructors declared
-- in the current module.
hsDataConstrs :: Term t
              => ModuleName   -- The name of the module that contains t.
              -> t            -- The given syntax phrase.
              ->([PName], [PName])

-- Collect the type constructors and class names that occur in a syntax
-- phrase. The first list in the result contains the type constructor/
-- classes declared in other modules, and the second list contains
-- the ones declared in the current module.
hsTypeConstrsAndClasses :: Term t
                        => ModuleName
                        -> t
                        -> ([PName], [PName])

-- Collect the type variables declared in the given syntax phrase.
hsTypeVbls :: Term t => t -> [PName]

-- Collect the class instance names of the specified class which occur
-- in a given syntax phrase. In the result, the first list contains the
-- class instances declared in other modules, and the second list
-- contains the class instance names declared in the current module.
hsClassMembers :: Term t
               => String -> ModuleName -> t -> ([PName], [PName])

-- Collect the free and declared variables in the given syntax phrase.
-- The first list in the result contains the free variables, and the
-- second list contains the declared variables.
hsFreeAndDeclaredPNs :: (Term t, MonadPlus m)
                     => t -> m ([PName], [PName])

-- Given syntax phrases t1 and t2, if t1 occurs in t2, then return those
-- variables which are declared in t2, and accessible to t1, otherwise
-- return [].
hsVisiblePNs :: (Term t1, Term t2, FindEntity t1, MonadPlus m)
             => t1 -> t2 -> m [PName]

-- Return all the possible qualifiers of an identifier in a module.
hsQualifier :: PNT -> InScopes -> [ModuleName]
```

```
-- Return the identifier(s) defined by a function/pattern binding.
definedPNs :: HsDeclP -> [PName]

-- The HasNameSpace class.
class HasNameSpace t where
Methods
  -- Return an entity's name space information.
  hasNameSpace :: t -> NameSpace
Instances
  HasNameSpace PNT
  HasNameSpace ENT
```

F.2.3 Property Checking

```
-- Return True if a string is a lexically valid variable name.
isVarId :: String -> Bool

-- Return True if a string is a lexically valid constructor name.
isConId :: String -> Bool

-- Return True if a string is a lexically valid operator name.
isOperator :: String -> Bool

-- Return True if the given identifier (represented by PName) is a
-- top-level identifier.
isTopLevelPN :: PName -> Bool

-- Return True if the given identifier is qualified.
isQualifiedPN :: PName -> Bool

-- Return True if an identifier is a function name defined in the
-- syntax phrase given by the second parameter.
isFunPN :: Term t => PName -> t -> Bool

-- Return True if an identifier is defined by a pattern binding in
-- the syntax phrase given by the second parameter.
isPatPN :: Term t => PName -> t -> Bool

-- Return True if an identifier is a type constructor.
isTypeCon :: PNT -> Bool

-- Return True if a declaration declares a type signature.
isTypeSig :: HsDeclP -> Bool

-- Return True if a declaration is a function definition.
isFunBind :: HsDeclP -> Bool

-- Return True if a declaration is a pattern binding.
```

```
isPatBind :: HsDeclP -> Bool

-- Return True if a declaration is declares a simple pattern binding.
isSimplePatBind :: HsDeclP -> Bool

-- Return True if a declaration is a complex pattern binding.
isComplexPatBind :: HsDeclP -> Bool

-- Return True if a declaration is a function/pattern definition.
isFunOrPatBind :: HsDeclP -> Bool

-- Return True if a declaration defines a class.
isClassDecl :: HsDeclP -> Bool

-- Return True if a declaration defines a class instance.
isInstDecl :: HsDeclP -> Bool

-- Return True if a function is a directly recursive function.
isDirectRecursiveDef :: HsDeclP -> Bool

-- Return True if the two given syntax phrases refer to the same
-- occurrence in the code.
sameOccurrence ::(Term t, Eq t) => t -> t -> Bool

-- Return True if the declaration declared the specified identifier,
-- or its type signature.
defines :: PName -> HsDeclP -> Bool

-- Return True if the given syntax phrase contains any free variables.
hasFreeVars :: Term t => t -> Bool

-- Return True if the first syntax phrase is part of the second one.
findEntity :: (FindEntity a, Term b) => a -> b -> Bool

-- Find the declarations that define the specified entities.
definingDecls :: [PName]      -- The entities.
              -> [HsDeclP]           -- A collection of declarations.
              -> Bool                   -- Include the type signature or not.
              -> Bool                   -- Check the local declarations or not.
              -> [HsDeclP]

-- Return True if the identifier is used in the RHS of a definition.
isUsedInRhs::(Term t) => PNT -> t -> Bool

-- The HsDecls class.
class Term t => HsDecls t where
Methods
  -- Return the declarations that are directly enclosed in the given
  -- syntax phrase.
```

```
  hsDecls :: t -> [HsDeclI PNT]

  -- Replace the directly enclosed declaration list with the given
  -- declaration list.
  replaceDecls :: t -> [HsDeclI PNT] -> t

  -- Return True if the identifier is declared in the given syntax phrase.
  isDeclaredIn :: PName -> t -> Bool

-- Return True if the given name is in scope and can be used unqualified.
isInScopeAndUnqualified :: String -> InScopes -> Bool.
```

F.2.4 Modules and Files

```
-- Return the client module and file names of the given module.
clientModsAndFiles ::(PFE0_IO err m, IOErr err, HasInfixDecls i ds,
                      QualNames i m1 n, Read n, Show n)
                   => ModuleName
                   -> PFEOMT n i ds ext m [(ModuleName, String)]

-- Return the server module and file names of the given module.
serverModsAndFiles ::(PFE0_IO err m, IOErr err, HasInfixDecls i ds,
                      QualNames i m1 n, Read n, Show n)
                   => ModuleName
                   -> PFEOMT n i ds ext m [(ModuleName, String)]

-- Return True if the given module name exists in the project.
isAnExistingMod ::(PFE0_IO err m, IOErr err, HasInfixDecls i ds,
                   QualNames i m1 n, Read n, Show n)
                => ModuleName
                -> PFEOMT n i ds ext m Bool

-- From file name to module name (assume that a file only contains
-- one module).
fileNameToModName ::(PFE0_IO err m, IOErr err, HasInfixDecls i ds,
                     QualNames i m1 n, Read n, Show n)
                  => String
                  -> PFEOMT n i ds ext m ModuleName
```

F.3 Program Transformation

F.3.1 Adding a Syntax Phrase

```
-- Adding a declaration to the declaration list of the given syntax phrase
-- If the second argument is Nothing, then the declaration will be added to
-- the beginning of the declaration list, but after the data type
-- declarations is there is any.
addDecl :: MonadState (([PosToken], Bool), t1) m
```

```
        => t            -- The AST.
        -> Maybe PName
        -> ([HsDeclP], Maybe [PosToken])
        -> Bool         -- The declaration is top-level or not.
        -> m t

-- Add an import declaration to a module.
addImportDecl :: MonadState (([PosToken], Bool), t1) m
              => HsModuleP
              -> HsImportDeclP
              -> m HsModuleP

-- Add entities (given by the third argument) to the explicit entity list
-- in the declaration importing the specified module.
addItemsToImport :: MonadState (([PosToken], Bool), t1) m
                 => ModuleName
                 -> Maybe PName
                 -> Either [String] [EntSpecP]
                 -> HsModuleP
                 -> m HsModuleP

-- Add entities to the export list of a module.
addItemsToExport :: MonadState (([PosToken], Bool), t1) m
                 => HsModuleP
                 -> Maybe PName
                 -> Bool
                 -> Either [String] [HsExportEntP]
                 -> m HsModuleP

-- Add entities to the hiding list of an import declaration which
-- imports the specified module.
addHiding :: MonadState (([PosToken], Bool), t1) m
          => ModuleName
          -> HsModuleP
          -> [PName]
          -> m HsModuleP

-- Add a guard expression to the RHS of a function definition (or a
-- pattern binding).
addGuardsToRhs :: MonadState (([PosToken], Bool), t1) m
               => RhsP
               -> HsExpP
               -> m RhsP

-- Add parameters to a function definition (or simple pattern binding).
addParamsToDecls
   :: (MonadPlus m, (MonadState (([PosToken], Bool),(Int,Int)) m)
   => [HsDeclP]-> PName -> [PName]-> m [HsDeclP]
```

F.3.2 Removing a Syntax Phrase

```
-- Remove the declaration (and the type signature if the second
-- parameter is True) that defines the given identifier.
rmDecl :: MonadState (([PosToken], Bool), t1) m
       => PName -> Bool -> [HsDeclP] -> m [HsDeclP]

-- Remove the type signature that defines the given identifier's
-- type from the declaration list.
rmTypeSig :: MonadState (([PosToken], Bool), t1) m
          => PName -> [HsDeclP]-> m [HsDeclP]

-- Remove the specified items from the entity list in the import
-- declarations of a module.
rmItemsFromImport :: MonadState (([PosToken], Bool), t1) m
                  => HsModuleP -> [PName] -> m HsModuleP

-- Remove the specified entities from the module's exports. The entities
-- can be specified by either entity names or ASTs.
rmItemsFromExport :: MonadState (([PosToken], Bool), t1) m
                  => HsModuleP
                  -> Either ([ModuleName], [PName]) [HsExportEntP]
                  -> m HsModuleP  The result.

-- Remove the sub entities of the specified type constructor or class
-- from the exports.
rmSubEntsFromExport :: MonadState (([PosToken], Bool), t1) m
                    => PName -> HsModuleP -> m HsModuleP

-- Remove the first n parameters of an identifier in an expression.
rmParams :: (MonadPlus m, MonadState (([PosToken], Bool), t1) m)
         => PNT -> Int -> HsExpP -> m HsExpP

-- Unqualify the uses of the given identifiers.
rmQualifier :: (MonadState (([PosToken], Bool), t1) m, Term t)
            => [PName] -> t -> m t

-- The Delete class.
class (Term t, Term t1) => Delete t t1 where

Method
 -- Delete the occurrence of the given syntax phrase.
 delete :: (MonadPlus m, MonadState (([PosToken], Bool), t2) m)
        => t        -- The syntax phrase to delete.
        -> t1       -- The contex where the syntax phrase occurs.
        -> m t1     -- The result.
Instances
  Term t => Delete HsExpP t
  Term t => Delete HsPatP t
  Term t => Delete HsImportDeclP t
```

F.3.3 Updatng a Syntax Phrase

```
-- The Update class.
class (Term t, Term t1) => Update t t1 where

Methods
 -- Update the occurrence of a syntax phrase in the given scope by
 -- another syntax phrase of the same type.
 update:: (MonadPlus m, MonadState (([PosToken], Bool), t2) m)
       => t -> t -> t1 -> m t1
Instances
  Term t => Update HsExpP t
  Term t => Update PNT t
  Term t => Update HsMatchP t
  Term t => Update HsPatP t
  Term t => Update [HsPatP] t
  Term t => Update [HsDeclP] t
  Term t => Update HsDeclP t
  Term t => Update HsImportDeclP t
  Term t => Update HsExportEntP t


-- Rename each occurrence of the given identifier with an automatically
-- created new name if the identifier is declared in syntax phrase.
autoRenameLocalVar :: (MonadPlus m, Term t)
                   => Bool       -- False means only modifying the AST.
                   -> PName -> t -> m t


-- Rename the occurrences of the given identifier with given the new
-- name and qualifier.
renamePN ::((MonadState (([PosToken], Bool), t1) m),Term t)
         => PName                -- The identifier to be renamed.
         -> Maybe ModuleName     -- The new qualifier.
         -> String               -- The new name.
         -> Bool                 -- False means only modifying the AST.
         -> t                    -- The syntax phrase
         -> m t
```

F.4 Some Miscellaneous Functions

F.4.1 Parsing and Writing

```
-- Parse and scope analyse a Haskell module.
parseSourceFile :: ... => FilePath
                -> m (InScopes, Exports, HsModuleP, [PosToken])

-- Write the refactored program to files.
writeRefactoredFiles :: ...
```

```
                    => Bool      -- True means sub-refactoring.
                    -> [((String,Bool),([PosToken],HsModuleP))]
                    -> m ()
```

F.4.2 From Textual Selection to AST Representation

```
-- Find the identifier (in PNT format) whose start position is the
-- given location in the specified file.
locToPNT :: Term t
         => String       -- The file name.
         -> Int          -- The row number.
         -> Int          -- The column number.
         -> t            -- The syntax phrase.
         -> Maybe PNT

-- Given the syntax phrase (and the token stream), find the
-- largest-leftmost expression contained in the region specified
-- by the start and end position.
locToExp :: Term t
         => SimpPos         -- Start position.
         -> SimpPos         -- End position.
         -> [PosToken]      -- The token stream.
         -> t               -- The AST.
         -> Maybe HsExpP
```

F.4.3 Combinators for Applying a Refactoring

```
-- Apply a transformation to a Haskell module. If the
-- module information is provided, this function will
-- use the provided information, otherwise it will
-- parse and analysis the module to get the information.
applyRefac::(PFE2MT (PFE0State HsName.Id) Names.QName ds
                    (PFE2Info (SN HsName.Id), ext) m)
          => ((InScopes,Exports,HsModuleP) -> HsModuleP)
          -> Maybe (InScopes,Exports,HsModuleP,[PosToken])
          -> String
          -> m ((String, Bool), ([PosToken], HsModuleP))

-- Apply a transformation to a collection of modules if
-- the information of these modules is explicitly given,
-- otherwise apply the transformation to all the client
-- modules of the module contained in the specified file.
applyRefac::(PFE2MT (PFE0State HsName.Id) Names.QName ds
                    (PFE2Info (SN HsName.Id), ext) m)
          => ((InScopes,Exports,HsModuleP) -> HsModuleP)
          -> Maybe [(InScopes,Exports,HsModuleP,[PosToken])]
          -> String
          -> m [((String, Bool), ([PosToken], HsModuleP))]
```

F.4.4 Creating New Names

```
-- Create a new name basing on the given prefix, starting postfix
-- integer and the collection of names which can not be taken by
-- the new name. Suppose the old name is f, then the new name
-- will have a format of f_i, where i is an integer.
mkNewName :: String        -- The prefix.
          -> Maybe Int     -- The possible posfix.
          -> [String]
          -> String
```

F.4.5 Converting Between AST Representations

```
-- From PNT to PName representation of an identifier.
pNTtoPN :: PNT -> PName

-- From PNT to string representation of an identifier.
pNTtoName :: PNT -> String

-- From PName to string representation of an identifier.
pNtoName :: PName -> String

-- From expression to PNT representation of an identifier. A default
-- PNT value is returned if the expression is not a single identifier.
expToPNT :: HsExpP -> PNT

-- From pattern to PNT representation of an identifier. A default
-- PNT value is returned if the pattern is not a single identifier.
patToPNT :: HsPatP -> PNT

-- From string to PNT representation of an identifier. Default
-- location and name space information is used (the same applies
-- to the following functions in this sub-section).
nameToPNT :: String -> PNT

-- From string to PName representation of an identifier.
nameToPN :: String -> PName

-- From string to expression representation of an identifier.
nameToExp :: String -> HsExpP

-- From string to pattern representation of an identifier.
nameToPat :: String -> HsPatP
Compose a pattern from a String.

-- From PName to expression representation of an identifier.
pNtoExp :: PName -> HsExpP

-- From PName to pattern representation of an identifier.
pNtoPat :: PName -> HsPatP
```

```
-- Transform a complex function definition (or pattern binding) into
-- a simple function definition (or pattern binding) using case and
-- conditional expressions. A function definition (or pattern binding)
-- is simple if it has only one equation, no guards, and all of its
-- formal parameters are simple variables.
simplifyDecl :: Monad m => HsDeclP -> m HsDeclP
```

F.4.6 Regarding to Locations

```
-- Change the absolute define locations of local variables to relative
-- locations in the given AST.
toRelativeLocs :: Term t => t -> t

-- Remove source the location information.
rmLocs :: Term t => t -> t

-- Return the identifier's define location.
defineLoc :: PNT -> SrcLoc

Return the identifier's source location.
useLoc :: PNT -> SrcLoc
```

Bibliography

[1] The Emacs Editor. `http://www.gnu.org/software/emacs/`.

[2] The Haskell Editing Survey. `http://www.cs.kent.ac.uk/projects/refactor-fp/surveys/haskell-editors-July-2002-summary.txt`.

[3] The Vim Editor. `http://www.vim.org/`.

[4] The XEmacs Editor. `http://www.xemacs.org/`.

[5] A. V. Aho and R. Sethi and J. D. Ullman. *Compilers: Princiles, Techniques, and Tools*. Addison-Wesley, 1986.

[6] A. W. Appel and D. B. MacQueen. Standard ML of New Jersey. In *3rd International Symposium on Programming Language Implementation and Logic Programming, LNCS 528*, August 1991. pp. 1-13.

[7] L. Augustsson. HBC – The Chalmers Haskell Compiler. `http://www.math.chalmers.se/~augustss/hbc/hbc.html`, 1999.

[8] R. C. Backhouse. An Exploration of the Bird-Meertens Formalism. Technical Report CS 8810, 1988.

[9] M. Balazinska, E. Merlo, M. Dagenais, B. Lage, and K. Kontogiannis. Advanced Clone-Analysis to Support Object-Oriented System Refactoring. In *WCRE '00: Proceedings of the Seventh Working Conference on Reverse Engineering (WCRE'00)*, page 98. IEEE Computer Society, 2000.

[10] H. P. Barendregt. *The Lambda Calculus – Its Syntax and Semantics*, volume 103 of *Studies in Logic and the Foundations of Mathematics*. North-Holland, 1984.

[11] F. L. Bauer, M. Broy, B. Möller, P. Pepper, M. Wirsing, et al. *The Munich Project CIP. Vol. I: The Wide Spectrum Language CIP-L.* Number 183 in Lecture Notes on Computer Science. Springer Verlag, Berlin, Heidelberg, New York, Berlin, 1985.

[12] F. L. Bauer, H. Ehler, R. Horsch, B. Möller, H. Partsch, O. Paukner, and P. Pepper. *The Munich Project CIP. Vol. II: The Transformation System CIP-S, LNCS 292*, volume II. Springer Verlag, Berlin, Heidelberg, New York, Berlin, 1987.

[13] I. D. Baxter, A. Yahin, L. M. Moura, M. Sant'Anna, and L. Bier. Clone Detection Using Abstract Syntax Trees. In *ICSM*, pages 368–377, 1998.

[14] K. Beck. *Extreme Programming Explained: Embrace Change.* Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 2000.

[15] K. Bennett and M. Ward. Formal Methods to Aid the Evolution of Software. *Journal of Software Engineering and Knowledge Engineering*, 5(1):25–47, 1995.

[16] R. W. Bowdidge. *Supporting the Restructuring of Data Abstractions through Manipulation of a Program Visualization.* PhD thesis, University of California, San Diego, November 1995. Tech. Rep. No. CS95-457.

[17] J. Brant and D. Roberts. The Refactoring Browser. `http://st-www.cs.uiuc.edu/users/brant/Refactory/`, 1994.

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

[19] C. Nguyen-Viet. Transformation in HaRe. Technical report, Computing Laboratory, University of Kent, Canterbury, Kent, UK, December 2004.

[20] A. Church. *The Calculi of Lambda-Conversion*, volume 6 of *Annals of Mathematical Studies*. Princeton University Press, Princeton, 1951. (second printing, first appeared 1941).

[21] M. Cinnide. *Automated Application of Design Patterns: A Refactoring Approach*. PhD thesis, University of Dublin, Trinity College, 2000.

[22] M. Cinnide and P. Nixon. Composite Refactoring for Java Programs. Technical report, Dept. of Computer Science, Univ. College Dublin, 2000.

[23] J. R. Cordy. Comprehending Reality - Practical Barriers to Industrial Adoption of Software Maintenance Automation. In *IWPC*, pages 196–206, 2003.

[24] J. R. Cordy, T. R. Dean, and N. Synytskyy. Practical Language-independent Detection of Near-miss Clones. In *CASCON '04: Proceedings of the 2004 conference of the Centre for Advanced Studies on Collaborative research*, pages 1–12. IBM Press, 2004.

[25] 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.

[26] O. de Moor, R. Ettinger, and M. Verbaere. Nate – A Slicing-based Refactoring Tool. `http://web.comlab.ox.ac.uk/oucl/research/progtools/progtools/nate/nate.html`.

[27] O. de Moor and G. Sittampalam. Higher Order Matching for Program Transformation. In *FLOPS '99: Proceedings of the 4th Fuji International Symposium on Functional and Logic Programming*, pages 209–224, London, UK, 1999. Springer-Verlag.

[28] I. S. Diatchki, M. P. Jones, and T. Hallgren. A Formal Specification for the Haskell 98 Module System. In *ACM Sigplan Haskell Workshop*, 2002.

[29] S. Ducasse, M. Rieger, and S. Demeyer. A Language Independent Approach for Detecting Duplicated Code. In *ICSM*, pages 109–118, 1999.

[30] M. Erwig and D. Ren. A Rule-based Language for Programming Software Updates. *SIGPLAN Notices*, 37(12):88–97, 2002.

[31] M. Erwig and D. Ren. Programming Type-Safe Program Updates. In *ESOP*, pages 269–283, 2003.

[32] Martin Erwig. Programs Are Abstract Data Types. In *ASE '01: Proceedings of the 16th IEEE International Conference on Automated Software Engineering*, page 400, Washington, DC, USA, 2001. IEEE Computer Society.

[33] Martin Erwig and Deling Ren. Monadification of functional programs. *Sci. Comput. Program.*, 52(1-3):101–129, 2004.

[34] M. Fowler. Refactoring Home Page. `http://www.refactoring.com`.

[35] 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/`.

[36] A. Garrido. Software Refactoring Applied to C Programming Language. Master's thesis, University of Illinois at Urbana-Champaign, 2000.

[37] A. Garrido and R. Johnson. Refactoring C with Conditional Compilation. In *ASE*, pages 323–326, 2003.

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

[39] W. G. Griswold. *Analysis and Transformation of Source Code by Parsing and Rewriting,* . PhD thesis, Univ. of Washington, Dept. of CS and Engineering, 1991. Tech. Rep. No. 91-08-04.

[40] W. Guttmann, H. Partsch, W. Schulte, and T. Vullinghs. Tool Support for the Interactive Derivation of Formally Correct Functional Programs. *Journal of Universal Computer Science*, 9(2):173–188, February 2003.

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

[42] J. R. Hindley. The Principal Type-scheme Of An Object in Combinatory Logic. *Trans. American Math. Soc*, 146:29–60, 1969.

[43] P. Hudak, J. Peterson, and J. H. Fasel. A Gentle Introduction to Haskell. Tutorial, 1997.

[44] Paul Hudak. Conception, Evolution, and Application of Functional Programming Languages. *ACM Comput. Surv.*, 21(3):359–411, 1989.

[45] J. Hughes. Why Functional Programming Matters. *Computer Journal*, 32(2):98–107, 1989.

[46] J. Hughes. The Design of a Pretty-printing Library. In J. Jeuring and E. Meijer, editors, *Advanced Functional Programming*, pages 53–96. Springer Verlag, LNCS 925, 1995.

[47] I. D. Baxter. Using transformation systems for software maintenance and reengineering. In *ICSE '01: Proceedings of the 23rd International Conference on Software Engineering*, pages 739–740, Washington, DC, USA, 2001. IEEE Computer Society.

[48] IntelliJ IDE – Java IDE With Refactoring Support. `http://www.jetbrains.com/idea/`.

[49] J.Kort and R. Lämmel. Parse-Tree Annotations Meet Re-Engineering Concerns. In *Proceedings of the International Workshop on Source Code Analysis and Manipulation (SCAM 2003)*, Amsterdam, September 2003. IEEE Computer Society Press.

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

[51] S. P. Jones. A Pretty Printer Library in Haskell, Version 3.0, 1997.

[52] S. P. Jones. Compiling Haskell by Program Transformation: A Report from the Trenches. In *ESOP*, pages 18–44, 1996.

[53] S. P. Jones, L. Augustsson, D. Barton, B. Boutel, W. Burton, J. Fasel, K. Hammond, R. Hinze, P. Hudak, J. Hughes, T. Johnsson, M. Jones, J. Launchbury, E. Meijer, J. Perterson, A. Reid, C. Runciman, and P. Wadler. *Haskell 98 Language and Libraries: the Revised Report*. Cambridge University Press, 2003.

[54] K. Angelov and S. Marlow. Visual Haskell: A Full-featured Haskell Development Environment. Submitted to the Haskell Workshop 2005, 2005.

[55] Y. Kataoka, M. D. Ernst, W. G. Griswold, and D. Notkin. Automated Support for Program Refactoring Using Invariants. In *ICSM 2001, Proceedings of the International Conference on Software Maintenance*, pages 736–743, Florence, Italy, November 6–10, 2001.

[56] P. Klint. A Meta-environment for Generating Programming Environments. *ACM Trans. Softw. Eng. Methodol.*, 2(2):176–201, 1993.

[57] Günter Kniesel and Helge Koch. Static Composition of Refactorings. *Sci. Comput. Program.*, 52(1-3):9–51, 2004.

[58] B. Korel and J. Laski. Dynamic Program Slicing. *Information Processing Letters*, 29(3):155–163, 1988.

[59] W. F. Korman. Elbereth: Tool Support for Refactoring Java Programs. Msc thesis, Department of Computer Science and Engineering, University of California, San Diego, 1998.

[60] J. Kort and R. Lämmel. A Framework for Datatype Transformation. In B.R. Bryant and J. Saraiva, editors, *Proceedings Language, Descriptions, Tools, and Applications (LDTA'03)*, volume 82 of *ENTCS*. Elsevier, April 2003. 20 pages.

[61] R. Lämmel. Reuse by Program Transformation. In Greg Michaelson and Phil Trinder, editors, *Functional Programming Trends 1999*. Intellect, 2000. Selected papers from the 1st Scottish Functional Programming Workshop.

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

[63] R. Lämmel and S. Peyton Jones. Scrap Your Boilerplate: a Practical Design Pattern for Generic Programming. *ACM SIGPLAN Notices*, 38(3):26–37, March 2003. Proceedings of the ACM SIGPLAN Workshop on Types in Language Design and Implementation (TLDI 2003).

[64] R. Lämmel and S. Peyton Jones. Scrap More Boilerplate: Reflection, Zips, and Generalised Casts. In *Proceedings of the ACM SIGPLAN International Conference on Functional Programming (ICFP 2004)*, pages 244–255. ACM Press, 2004.

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

[66] R. Lämmel and J. Visser. Design Patterns for Functional Strategic Programming. In *Proc. of Third ACM SIGPLAN Workshop on Rule-Based Programming RULE'02*, Pittsburgh, USA, October5 2002. ACM Press. 14 pages.

[67] R. Lämmel and J. Visser. Typed Combinators for Generic Traversal. In *Proc. Practical Aspects of Declarative Programming PADL 2002*, volume 2257 of *LNCS*, pages 137–154. Springer-Verlag, January 2002.

[68] M. Schmidt-Schauß and M. Huber. A Lambda-Calculus with letrec, case, constructors and non-determinism. *CoRR*, cs.PL/0011008, 2000.

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

[70] S. Marlow. Hatchet: A Type Checking and Inference Tool for Haskell 98.

[71] K. Maruyama. Automated Method-extraction Refactoring by Using Block-based Slicing. In *SSR '01: Proceedings of the 2001 symposium on Software reusability*, pages 31–40, New York, NY, USA, 2001. ACM Press.

[72] J. Meacham and N. Winstanley. DrIFT – A Type Sensitive Preprocessor for Haskell. `http://repetae.net/john/computer/haskell/DrIFT/`.

[73] L. Meertens. Algorithmics – Towards Programming as a Mathematical Activity. In *CWI Symposium on Mathematics and Computer Science, CWI Monographs vol. 1, North-Holland*, 1986.

[74] T. Mens, S. Demeyer, and D. Janssens. Formalising Behaviour Preserving Program Transformations. In *ICGT '02: Proceedings of the First International Conference on Graph Transformation*, pages 286–301. Springer-Verlag, 2002.

[75] R. Milner. A Theory of Type Polymorphism in Programming. *Journal of Computer and System Sciences*, 17(3):348–375, 1978.

[76] I. Moore. Automatic Inheritance Hierarchy Restructuring and Method Refactoring. In *OOPSLA*, pages 235–250, 1996.

[77] J. Nordlander. *Reactive Objects and Functional Programming*. PhD thesis, Chalmers University of Technology, Gteborg, Sweden, 1999.

[78] W. Opdyke and R. Johnson. Refactoring: An Aid in Designing Application Frameworks and Evolving Object-Oriented Systems. In *Symposium on Object-Oriented Programming Emphasizing Practical Applications (SOOPPA)*, September 1990.

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

[80] P. Hudak and S. P. Jones and P. Wadler and B. Boutel and J. Fairbairn and J. Fasel and M. Guzman and K. Hammond and J. Hughes and T. Johnsson and D. Kieburtz and R. Nikhil and W. Partain and J. Peterson. Report on the programming language Haskell: a non-strict, purely functional language version 1.2. *SIGPLAN Not.*, 27(5):1–164, 1992.

[81] PacSoft. Programatica: Integrating Programming, Properties, and Validation. `http://www.cse.ogi.edu/PacSoft/projects/programatica/`.

[82] PacSoft. Programatica's Type Checker. `http://www.cse.ogi.edu/PacSoft/projects/programatica/tools/base/TI/`.

[83] H. Partsch and R. Steinbrüggen. Program Transformation Systems. *ACM Computing Surveys*, 15(3), September 1983.

[84] H. A. Partsch. *Specification and Transformation of Programs: a Formal Approach to Software Development*. Springer-Verlag New York, Inc., New York, NY, USA, 1990.

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

[86] R. E. Harper and D. MacQueen and R. Milner. Standard ML. Technical report, March 1986. (ECS-LFCS-86-2).

[87] R. Plasmeijer and M. V. Eekelen. The Concurrent Clean Language Report (version 2.0). Technical report, University of Nijmegen, March 2001.

[88] R. Tischler and R.Schaufler and C. Payne. Static Analysis of Programs as an Aid to Debugging. In *Proceedings of the ACM SIGSOFT/SIGPLAN software engineering symposium on High-level debugging*, pages 155–158, 1983.

[89] R. Virding and C. Wikstrom and M. Williams. *Concurrent Programming in ERLANG (2nd ed.)*. Prentice Hall International (UK) Ltd., Hertfordshire, UK, UK, 1996.

[90] R.Bird and O. de Moor. *Algebra of programming*. Prentice-Hall, Inc., Upper Saddle River, NJ, USA, 1997.

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

[92] D. Rémy. Using, Understanding, and Unraveling the OCaml Language. In Gilles Barthe, editor, *Applied Semantics. Advanced Lectures. LNCS 2395.*, pages 413–537. Springer Verlag, 2002.

[93] D. Roberts, J. Brant, and R. 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/`.

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

[95] W. W. Royce. Managing the Development of Large Software Systems: Concepts and Techniques. In *ICSE '87: Proceedings of the 9th international conference on Software Engineering*, pages 328–338, Los Alamitos, CA, USA, 1987. IEEE Computer Society Press.

[96] C. Ryder. The Medina Metrics Library for Haskell. `http://www.cs.kent.ac.uk/~cr24/medina/`, 2002.

[97] S. Horwitz and T. W. Reps. The Use of Program Dependence Graphs in Software Engineering. In *International Conference on Software Engineering*, pages 392–411, 1992.

[98] S. Thompson and C. Reinke. A Case Study in Refactoring Functional Programs. In Roberto Ierusalimschy, Lucilia Figueiredo, and Marcio Tulio Valente, editors, *VII Brazilian Symposium on Programming Languages*, pages 1–16. Sociedade Brasileira de Computacao, May 2003.

[99] Inc. Semantic Designs. Clone Doctor: Software Clone Detection Removal . `http://www.semdesigns.com/Products/Clone/`.

[100] T. Sheard. Generic Unification via Two-Level Types and Parameterized Modules. In *ICFP'01, Firenze, Italy*, September 2001. expanded version submitted to JFP.

[101] F. Simon, F. Steinbruckner, and C. Lewerentz. Metrics Based Refactoring. In *CSMR*, pages 30–38, 2001.

[102] G. Sittampalam and O. de Moor. MAG - A Small Transformation System for Haskell. `http://web.comlab.ox.ac.uk/oucl/research/areas/progtools/mag/`.

[103] F. Tip. A Survey of Program Slicing Techniques. *Journal of Programming Languages*, 3(3), 1995.

[104] F. Tip, A. Kieżun, and D. Bäumer. Refactoring for Generalization Using Type Constraints. In *Object-Oriented Programming Systems, Languages, and Applications (OOPSLA 2003)*, pages 13–26, Anaheim, CA, USA, November 6–8, 2003.

[105] L. Tokuda and D. Batory. Evolving Object-Oriented Applications with Refactorings. Technical Report CS-TR-99-09, University of Texas, Austin, March 1, 1999.

[106] T. Tourwé and T. Mens. Identifying Refactoring Opportunities Using Logic Meta Programming. In *CSMR*, pages 91–100, 2003.

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

[108] David Turner. Miranda: A Non-strict Functional Language with Polymorphic Types. In *In J.-P. Jouannaud, editor, 2nd Functional programming languages and computer architecture, LNCS 201*, pages 1–16, September 1985.

[109] M. G. J. van den Brand and J. J. Vinju. Generation by Transformation in ASF+SDF. In *GPCE Workshop on Software Transformation Systems (STS)*, 2004. Position paper [Slides on "Combining Formalisms for Software Transformation" ./slides/sts.pdf,PDF of the position paper./papers/sts-brandvinju.pdf].

[110] Jurgen Vinju. *Program Restructuring to Aid Software Maintenance*. PhD thesis, CWI, 2005.

[111] E. Visser. HsOpt – An Optimizer for the Helium Compiler. `http://www.stratego-language.org/twiki/bin/view/Stratego/HsOpt`.

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

[113] E. Visser. Program Transformation with Stratego/XT: Rules, Strategies, Tools, and Systems in StrategoXT-0.9. In C. Lengauer et al., editors, *Domain-Specific Program Generation*, volume 3016 of *Lecture Notes in Computer Science*, pages 216–238. Spinger-Verlag, June 2004.

[114] A. von Mayrhauser and A. M.Vans. Program comprehension during software maintenance and evolution. *Computer*, 28(8):44–55, 1995.

[115] D. G. Waddington and B.Yao. High Fidelity C++ Code Transformation. In J. Boyland and G. Hedin, editors, *Proceedings of the 5th workshop on Language Descriptions, Tools and Applications (LDTA 2005)*, 2005.

[116] P. Wadler. Monads for Functional Programming. In *Advanced Functional Programming, First International Spring School on Advanced Functional Programming Techniques-Tutorial Text*, pages 24–52, London, UK, 1995. Springer-Verlag.

[117] M. Weiser. Program Slicing. In *ICSE '81: Proceedings of the 5th International Conference on Software engineering*, pages 439–449, Piscataway, NJ, USA, 1981. IEEE Press.

[118] M. Weiser. Program Slicing. *IEEE Trans. Software Eng.*, 10(4):352–357, 1984.

[119] Z. M. Ariola and M. Felleisen and J. Maraist and M. Odersky and P. Wadler. A Call-by-Need Lambda Calculus. In *Proceedings of 22nd Annual ACM SIGACT-SIGPLAN Symposium on Principles of Programming Languages (POPL)*, pages 233–246, 1995.

[120] Z. M. Ariola and S. Blom. Lambda Calculi plus Letrec. Technical report, July 1997.