VERIFICATION OF REFACTORINGS IN ISABELLE/HOL

a thesis submitted to
The University of Kent
in the subject of computer science
for the degree
of master of science by research.

By
Nik Sultana
April 2008

Contents

List of Figures

Abstract

Refactorings are source-to-source behaviour-preserving program transformations that are used for improving program structure. Programmers refactor code to adapt it when new functionality is added or when the code is being repaired – refactoring serves to keep the code "clean" and more maintainable. Refactoring can also be used as an exploratory technique for understanding source code.

The process of refactoring has been automated through the implementation of tools; these tools assist programmers by handling the consistent application of behaviour-preserving changes to the code. It is desirable that the implementations of refactorings are correct: bugs might otherwise be introduced in refactored programs. The correctness, i.e. behaviour-preservation, of refactoring is traditionally probed by testing the refactored program and not the refactoring implementation directly. Recently, automated testing techniques have been used to test implementations of refactorings directly, but the coverage of testing is partial at best. The verification of refactorings is more challenging but determines whether a refactoring is behaviour-preserving for all possible programs. We study the verification of refactorings using the proof assistant Isabelle/HOL for untyped and typed λ-calculi.

Some of the issues encountered during verification are technical rather than purely theoretical: they relate to the embedding of the programming language in the proof environment. The reasons for our choice of techniques are discussed. We also discuss other practical considerations such as the readability of mechanised refactorings, and the avoidance of computationally expensive refactorings.

Acknowledgements

First and foremost, I am very grateful to Simon Thompson for his invaluable advice, encouragement and support. Many other dissertations start with similar words of thanks, and during the course of this year I have come to appreciate how much these words are indeed merited.

Thanks also go to my examiners, Dr. Stefan Kahrs and Prof. Oege de Moor, for their insightful comments. I am also indebted to other members of the Theoretical Computer Science and the Functional Programming research groups for stimulating discussions. In particular, I thank Eerke Boiten, Olaf Chitil, Rodolfo Gomez, Claus Reinke and Axel Simon for sharing their knowledge and passion for their research. Many other members of the Laboratory, and the friends I met at the infamous Tyler Court A hangar, have helped make my stay at Kent enjoyable as well as productive.

This research was possible thanks to financial support from the Computing Laboratory and the Malta Government Scholarship Scheme through award MGSS/2006/007. Cambridge University Press provided financial support to attend the Federated Logic Conference prior to starting MSc research. Thanks also to Byron Cook and Moshe Vardi for making this possible. I also thank the organisers of the TYPES Summer School 2007 for providing financial assistance to attend the event.

My deepest thanks go to my parents and Anna for their continuous love and support.

Lil ġenituri tiegħi

Preliminaries

This chapter introduces notation, definitions and notions used throughout the dissertation. The λ-calculus and its theory are outlined, some methods for encoding language syntax are introduced and the choice of substitution operation is discussed.

The symbol $\stackrel{\text{def}}{=}$ will be used to denote definitional equality, $\stackrel{\wedge}{=}$ for abbreviations and $\equiv$ for identity. The abbreviation "iff" stands for "if and only if", and "wrt" abbreviates "with respect to".

λ-calculus

The λ-calculus is the surviving fragment of a system that was intended to be a foundation of mathematics. The calculus is considered to be a canonical functional programming language, but it has also had a profound impact on other areas of computer science, for instance automated reasoning, and mathematics. Barendregt (1997) surveys the influence the calculus has had in the first six decades of its existence.

The pure untyped calculus will be described next and we will build on this when defining other languages in Chapters 5 and 6. More details on the calculus can be obtained from the books by Barendregt (1981) and Hankin (1994).

Let V be a denumerable set of variable names $x, x', y, z, \ldots$ ranged over by the metavariable v. We will use the symbol ∇ as a constructor to flag variable occurrences. An *abstraction*, such as $\lambda v \cdot M$, is an expression representing a function. *Application* (or combination) of a function to an argument is usually denoted by juxtaposition in the λ-calculus, but the symbol "$\circ$" will be used explicitly here. The expression occurring on the left of "$\circ$" is called the *operator*, and the expression on the right is the *operand*. Let M and N range over expressions in this language. The grammar of the calculus is as follows:

$$\begin{array}{rcll} M & ::= & \nabla v & \text{Variable} \\ & | & \lambda v \cdot M & \text{Abstraction} \\ & | & M \circ N & \text{Application} \end{array}$$

Let Λ denote the least set induced by this grammar. In $\lambda x \cdot M$, M is called the *body* of the expression and λ binds occurrences of x in M. Variables in an expression that are not bound are said to be *free*; "$y \in FVM$" predicates that y is a free variable in expression M. The operation "BVM" produces the set

containing the names of all variables bound in the expression M. The scope of bound variables extends to as right as possible modulo parenthetical enclosure. Parenthesis are used to disambiguate scopes in linear representations of terms. In terms of precedence, "λ" is weaker than "$\circ$".

To facilitate explanation, we will not explicitly use ∇ to mark variable occurrences in this chapter. However the symbol will be used in the formal developments described in Chapters 5 and 6.

The grammar of *contexts* is obtained by extending the previous grammar with a new clause: "[]", called a *hole*. Let C be a metavariable ranging over this set, then it stands for a *context*. If C stands for a context having n holes then this is reflected by writing C followed by n occurrences of the symbol []. A hole can be filled by expressions in Λ, yielding other contexts (if holes are still present) or expressions in Λ (if all holes have been filled). In general, contexts can have multiple holes, but it is useful to define *single-holed* contexts. Contexts will be used to define contextual-equivalence – a characterisation of congruence – in § 4.1.2.

α-conversion refers to the renaming of bound variables such that their new names are *fresh* – i.e., they do not appear free in the expression. Often this conversion is done implicitly; in order to facilitate reasoning, expressions are identified up to renaming of bound variables. This practice abstracts away name information, retaining the pure *binding structure* – i.e., scope information. The symbol $\equiv_\alpha$ is used to denote identity modulo renaming of bound variables.

Substitution is the canonical transformation operation for expressions in the λ-calculus. It is parametrised by two expressions and a variable, and it transforms the first expression by substituting its free occurrences of the variable for the second expression. We will define different substitution operations later, and justify our choice of operation based on its suitability for the model of refactoring we will use.

Substitution occurs during β-reduction. This reduction may take place when an abstraction (the operator) is applied to some other expression (the operand), and results in the latter being substituted for free occurrences of the bound variable in the body of the operator.

A β-reducible subexpression is called a *β-redex*. When an expression is reduced to a *normal form*, or value, then it cannot be reduced any further. There are different kinds of normal forms and further details will be provided below.

A *reduction strategy* is a commitment to a consistent position in expressions where redexes are picked. For example, one might choose to reduce operands themselves prior to reducing the operator. This is called *applicative-order* reduction. *Normal-order* reduction is an alternative reduction strategy that involves substituting the operands unevaluated into the operator then reducing the whole expression.

β-reduction is the means through which computation takes place in the calculus, and it is crucial that reduction is *deterministic* – in the sense that at most one normal form must exist for any term; the actual definition of what constitutes

a normal form, as previously mentioned, may vary with the calculus' specific definition and use. Values of the "standard" unityped lambda calculus – technically known as the λK-calculus – are terms in *head* normal form (HNF). Terms having such a normal form are said to be *solvable*. The definition and motivation for HNF are given in (Barendregt 1981).

Variable *capture* is a potential side-effect of substitution. Capture occurs when a free variable becomes bound. That is, a free variable in the operand becomes bound when the operand is placed inside the operator. To avoid capture occurring during β-reduction, the reduction rule may be given with a precondition to ensure that capture does not take place – e.g., none of the free variables in the operand appear bound in the operator. Alternatively, the substitution might *rename* bound variables (i.e., perform α-conversion) such that capture is avoided. Different definitions of the substitution operation will be discussed below. A substitution that allows capture will be used throughout the dissertation and to balance this the β-rule will be conditional on non-capture, as shown in the definition below.

Equational theory

The relation $\simeq \subseteq \Lambda \times \Lambda$ formalises the equational semantics of expressions in Λ. It is defined inductively through the following rules. The names of the rules occur on their right and their side-conditions occur on the left. The first set of rules induce the core theory of the λ-calculus.

$$Fresh(z, M)\ \frac{}{\lambda x \cdot M \simeq \lambda z \cdot M[z/x]}\ (\alpha)$$

$$\neg Captures(N, M)\ \frac{}{(\lambda x \cdot M)\ N \simeq M[N/x]}\ (\beta)$$

$$x \notin FV(M)\ \frac{}{\lambda x \cdot (M \circ x) \simeq M}\ (\eta)$$

The second set of rules ensure that $\simeq$ is an equivalence relation:

$$\frac{}{M \simeq M}\ (\textsc{Refl}) \qquad \frac{N \simeq M}{M \simeq N}\ (\textsc{Symm})$$

$$\frac{M \simeq M' \qquad M' \simeq N}{M \simeq N}\ (\textsc{Tran})$$

The last set of rules ensure that $\simeq$ is a compatible relation:

$$\frac{M \simeq M' \qquad N \simeq N'}{M \circ N \simeq M' \circ N'}\ (\textsc{CompApp})$$

$$\frac{M \simeq N}{\lambda x \cdot M \simeq \lambda x \cdot N}\ (\textsc{CompAbs})$$

The equivalence and compatibility properties make $\simeq$ a congruence relation. The rule CompAbs is the rule of weak extensionality, often called ξ. *Fresh(z,M)* predicates that the variable z is fresh in M: a weak definition of this says that z does not appear free in M. Another definition for *Fresh(z,M)* says that z does not appear either free or bound in M; this definition avoids the problem of shadowing (described further down). We will use the latter definition. *Captures(N,M)* means that substituting M for one of the free variables in N might capture free variables in M. These predicates will feature in later chapters and their definitions will be made precise then.

Language encoding

In order to produce machine-checkable proofs about the correctness of metaprograms one must also include a specification of the programming language under study. Some specification techniques lend themselves better to automated reasoning. We discuss this here and describe some techniques. This is covered in more detail in §3.2.

In informal reasoning we usually identify expressions up to α-equivalence, but formal reasoning requires making explicit how this is done and requires the implementation of this conversion. Having concrete variable names makes this more difficult, so this motivated de Bruijn (1972) to develop an anonymous representation for expressions – i.e. removing the names of bound variables from the calculus. In the *de Bruijn indices* representation the abstraction clause stipulates a binding but does not explicitly name the variable being bound. Variable occurrences are natural numbers measuring the distance, in terms of the number of abstractions, from the binding occurrence that binds them. For example, $\lambda x.\lambda y.x$ becomes $\lambda\lambda 2$, and so does $\lambda y.\lambda z.y$. Free variables in de Bruijn indices are treated as if they were bound by binding occurrences outside the expression, so a free variable is recognisable since it is a number greater than the number of λs that precede it. Every time a substitution takes place the indices of free variables need to be "shifted" to preserve their reference to the "external" binding occurrence.

Locally nameless representation (McBride & McKinna 2004) is an improvement of de Bruijn's technique. It is a hybrid approach: bound variables are de Bruijn indices while free variables are named.

The choice of which technique to use largely depends on what one has to use it for; none of the techniques are generally "clearly better" than the rest since each technique seems to have a niche of applications where it is more suitable. The de Bruijn system is criticised because of its poor readability (although it is certainly more readable than combinatory logic). On the other hand, it is more amenable for automation since by using it one avoids having to explicitly perform α-conversion. This is because through de Bruijn indices one does not represent individual expressions but rather their α-equivalence class. Thus when using a de Bruijn representation we do not deal with elements in Λ but with those in the quotient set $\Lambda_{/\equiv_\alpha}$.

The locally-nameless technique is an improvement over de Bruijn indices since shifting free variables is avoided altogether. This is accomplished by treating free and bound variables differently in the grammar itself. We outline more techniques for encoding languages in §3.2.

The following pseudocode snippets demonstrate how Λ might be encoded using classical (i.e., having concrete variable names), de Bruijn, and locally-nameless representations. Names are represented using natural numbers in the following example, but making `Exp` and `ExpLN` parametric to the type of names yields a more abstract definition. `Exp` encodes a language having concrete names, `ExpDB` is the de Bruijn index encoding and `ExpLN` uses locally-nameless representation.

```
data Exp = Var Nat
         | Lam Nat Exp
         | App Exp Exp

data ExpDB = VarDB Nat
           | LamDB ExpDB
           | AppDB ExpDB ExpDB

data ExpLN = FreeLN Nat
           | BoundLN Nat
           | LamLN ExpLN
           | AppLN ExpLN ExpLN
```

Using these datatypes, $\lambda x.\lambda y.(x \circ z)$ might be represented as:

- `Lam 0 (Lam 1 (App (Var 0) (Var 2)))` in `Exp`,
- `LamDB (LamDB (AppDB (VarDB 2) (VarDB 3)))` in `ExpDB`,
- `LamLN (LamLN (AppLN (BoundLN 2) (FreeLN 0)))` in `ExpLN`

Substitution

Substitution plays a central rôle in λ-calculi and its definition is influenced by the encoding of the language. We now describe two problems that might occur in expressions, then describe how variable capture can be mitigated. Further details on this can be obtained from (Hankin 1994, §2.3), (Barendregt 1981, §C) and (Pierce 2002, §6).

These problems might arise from using a name-carrying syntax encoding. The encoding used also affects the definition of the substitution operation, since it might be defined to avoid problems such as:

Shadowing: In terms such as $\lambda x.\lambda x.x$ the second binding of x is said to *shadow* the first. This expression is α-equivalent to $\lambda y.\lambda x.x$, where no shadowing occurs. Shadowing can be confusing and lead to mistakes in expression manipulation.

Variable capture occurs when a previously free variable becomes bound during expression transformation. For example consider the application $(\lambda x.\lambda y.x) \circ y$. We cannot β-reduce this to $\lambda y.y$ since the meaning of the expression would be changed – y becomes bound. Had the binding occurrence instead been z instead of y then the reduct $\lambda z.y$ would be acceptable.

To avoid variable capture we can either ensure that the arguments given to the substitution operation are such that capture cannot take place, or else we would need to rename bound variables to avoid capture. The latter option involves using a *renaming* definition of substitution. Different definitions of substitution will be described in the next section, but we anticipate this by pointing out that the (β) rule in the logic given earlier has the side-condition $\neg Captures(N, M)$. We will not be using a renaming definition of substitution in our work. The inputs to the substitution will come from the inputs that the user gives to the refactoring operation. The user is alerted if using the inputs they provided would entail name capture, otherwise the program is transformed. This model is described in more detail further down.

Shadowing is not a problem semantically, but variable capture is. The most drastic solution to this problem is to use a terse combinatory logic since this would eliminate the need for variables altogether. However, Turner (2006) points out that "It would seem that only a dedicated cryptologist would choose to write other than very small programs directly in combinatory logic". This "variable-free" philosophy has prosperred somewhat however, and developed into richer languages to facilitate programming – an early example of which is FP (Backus 1978). This has also spurred a style of programming called *point-free* programming (Cunha 2005). The language we have in mind however is not restricted to applicative terms alone, and therefore variables must feature in the object language.

The following methods assist in transforming expressions in a name-carrying language encoding while avoiding name capture:

Curry substitution assumes the presence of an infinite list of "next fresh variables" which can be consumed for renaming binding and bound variables in the rand before substitution. This is a *renaming* substitution operation.

Barendregt variable convention assumes that bound and free variables are always distinct in expressions we reason about. That is, reasoning is constrained to a subset of Λ such that for every pair of expressions M and N in this subset we have that $FVM \cap BVN = \emptyset$. Assuming this convention makes renaming unnecessary, since capture is impossible. Shadowing still remains possible, however.

Variable-capturing substitution

The differences between three different substitution operations are illustrated next. A similar exposition is given in Pierce (2002). We distinguish between the operations using a subscript numeral.

$$M[N/_1x] \stackrel{\text{def}}{=} \begin{cases} y & \text{if } M \equiv y \text{ and } y \neq x \\ N & \text{if } M \equiv y \text{ and } y = x \\ \lambda y \cdot (M'[N/_1x]) & \text{if } M \equiv \lambda y \cdot M' \\ (M'[N/_1x]) \circ (M''[N/_1x]) & \text{if } M \equiv (M' \circ M'') \end{cases} \tag{1}$$

$$M[N/_2x] \stackrel{\text{def}}{=} \begin{cases} y & \text{if } M \equiv y \text{ and } y \neq x \\ N & \text{if } M \equiv y \text{ and } y = x \\ \lambda x \cdot M' & \text{if } M \equiv \lambda y \cdot M' \text{ and } y = x \\ \lambda y \cdot (M'[N/_2x]) & \text{if } M \equiv \lambda y \cdot M' \text{ and } y \neq x \\ (M'[N/_2x]) \circ (M''[N/_2x]) & \text{if } M \equiv (M' \circ M'') \end{cases} \tag{2}$$

$$M[N/_3x] \stackrel{\text{def}}{=} \begin{cases} y & \text{if } M \equiv y \text{ and } y \neq x \\ N & \text{if } M \equiv y \text{ and } y = x \\ \lambda y \cdot M' & \text{if } M \equiv \lambda y \cdot M' \text{ and } y = x \\ \lambda y \cdot (M'[N/_3x]) & \text{if } M \equiv \lambda y \cdot M' \text{ and } y \neq x \\ & \text{and } y \notin FVN \\ \lambda z \cdot ((M'[z/_3y])[N/_3x]) & \text{if } M \equiv \lambda y \cdot M' \text{ and } y \neq x \\ & \text{and } y \in FVN \text{ then use fresh } z \\ & \text{i.e. } z \notin (FVN \cup FVM') \\ (M'[N/_3x]) \circ (M''[N_3/x]) & \text{if } M \equiv (M' \circ M'') \end{cases} \tag{3}$$

The three substitution operations are given in order of sophistication. The first operation does not do what we expect it to: it does not only replace N for free occurrences of x, but replaces N for *all* occurrences of x. The operation is too weak.

The second operation is an improvement over the first: it substitutes N for only free occurrences of x. The difference between the second and third operation is that the third substitution also avoids capturing free variables in N by changing the bound variable to a variable that does not occur freely in N – this ensures that the newly-introduced variable z does not capture free variables in N either.

In the formalisations of refactorings described in Chapters 5 and 6 we will use a substitution operation of the second kind – a *variable-capturing* substitution. This substitution is used since it formalises the operation required: it is up to the user to provide parameters to the refactoring, including the choice of names. The possibility of variable capture is checked prior to applying the transformation, so a substitution is only performed if there is no chance of capture. If the checks fail then the user is alerted and can choose to re-attempt the refactoring using

different arguments, or else they could choose to abort the refactoring altogether. They might need to perform a different refactoring to adapt the program further before re-attempting their original refactoring.

This model will be called *interactive*. An alternative approach would be to *compensate*: the refactoring would rename variables such that the refactoring can go through, then the user is informed of this measure when the refactoring is complete.

In the interactive model the side-conditions of a refactoring will lie explicitly in the definition of the refactoring. This is because it does not rely on sophisticated transformation operations (such as a renaming substitution) where these are not required. These two approaches are explained further in §2.2.1.

By using a variable-capturing substitution one avoids a technical difficulty: one may express formulæ involving substitution as equations. If the renaming substitution is used instead then such formulæ must be expressed using a relation that contains α-congruence. For example, we might want to prove that substituting a term for a variable that does not occur free in an expression is a redundant operation. This is formulated as follows for the second substitution operation:

$$\forall L.\ x \notin FV M \longrightarrow M[L/_2 x] = M$$

The result can then easily be proved by induction on the structure of expressions. The formulation using the third substitution operation is as follows:

$$\forall L.\ x \notin FV M \longrightarrow M[L/_3 x] \equiv_\alpha M$$

This is because the third definition may introduce new variable names, so its output cannot be guaranteed to be identical to the result had renaming been avoided. The result must therefore be shown to be equivalent modulo α-equivalence. Berghofer & Urban (2007) illustrate the difficulty of proving such a result when using a language encoding involving concrete variable names; they call proving the formula shown previously a "tour de force". They then proceed to compare the use de Bruijn indices and that of nominal techniques, described in §3.2, for encoding language syntax in order to mitigate this difficulty.

Part I

Chapter 1

Introduction

Programs are not only written to be executed, but also to be understood – later in time and potentially by people other than the author of the original code. Understanding the program code is a precondition to fixing or updating it, or adapting the code for reuse. Due to the sheer size of programs, writing program code in a style that facilitates comprehension is a very valuable skill. But even well-structured program code might need to be altered at some point in its life (software would not be soft if it was difficult to change) and each change can potentially degrade the structure of the program. Ill-structured programs are more difficult, and expensive, to maintain.

Refactorings are a kind of *program transformation*. Program transformations assist programmers in fulfilling both goals of programs – to be executed and to be understood. For example, program *optimisation* contributes towards a more efficient execution, while refactoring contributes to understanding and maintaining programs. Another kind program transformation, program *derivation*, contributes to the correct construction of programs from their specifications. Various kinds of program transformations have been implemented; this has enabled them to be applied to large programs and has made them accessible to programmers. For example, it is common for compilers to offer options to perform various optimisations during compilation.

Refactorings are procedures that assist programmers in keeping software understandable; they transform program code[1] to improve its clarity without changing its behaviour[2]. They offer a more powerful way to modify program code than changing it one character or line at a time. As with other modifications of program code, refactorings can be undone (rolled back) if they are found to be unsuitable. Refactoring can also be performed on abstract models of software and other artifacts – Mens & Tourwé (2004, §V) include "database schemas, software architectures and software requirements" among the kinds of artifacts that would

[1] Refactoring focuses on *existing* code, as emphasised by the subtitle of the book by Fowler & Beck (1999).

[2] *Behaviour-preservation* is usually taken to mean that for identical inputs, the original and refactored programs return identical outputs. Here we do not consider resource-usage efficiency as being part of a program's behaviour, and therefore might not be preserved by a refactoring.

benefit from refactoring. Here we study only the refactoring of program source code.

Perhaps the simplest refactoring is *renaming a variable*. This is not a mere invocation of a text editor's search-and-replace feature, since it respects scoping rules – recall that a refactoring must preserve program behaviour. More sophisticated examples of refactoring involve transforming a data structure (this might be done to conform to Parnas' principle of *information hiding*) or moving definitions between modules (while respecting the definitions' dependencies and avoiding name-clashes).

Refactorings have provisos to ensure that a program is not transformed if its behaviour will be changed as a result of that transformation. These *side-conditions* are checked in the course of applying the refactoring. We study the static application of refactorings, however *dynamic refactoring* has been studied by Roberts (1999) and others. If the side-conditions are not satisfied then the refactoring behaves like the identity function.

Refactorings have been automated and integrated with program development environments. Some such tools will be described in §2.1.1. *Partially-automated* refactoring tools automate the process of transforming program source code (and checking their side-conditions). *Fully-automated* refactoring tools extend partially-automated tools with the automatic detection of sites in source code which could be refactored and suggest a possible refactoring to use.

Such tools help make software modification manageable – indeed they are necessary for managing large programs since even renaming a variable can become daunting in a moderately-sized body of code.

1.1 Correctness of refactorings

Automated refactorings are useful because they enforce rigour with ease and automate tasks which are tedious and error-prone. However the usefulness of tools also depends on their quality: the tools themselves need to be correct otherwise they might introduce bugs in programs, leading to unspecified behaviour during their execution. Since tools are programs too, if a tool is to be correct then the specification against which it is checked must be correct to start with. In this work we study the correctness of refactorings: refactorings are correct if they preserve behaviour of arbitrary programs.
In order to assess whether a refactoring is correct we could either:

Test : traditionally each application of a refactoring is tested to check that the behaviour of the refactored program is identical to its original form. Recent research has studied testing refactorings directly. This is described further in §2.1.2.

Verify : this involves producing a formal proof that for all possible programs the refactoring does not change program behaviour.

We take the second approach to check refactorings. In turn, the correctness of our proofs will be checked by machine – this will be described in the next section.

Applying a refactoring might not improve a program's structure; this depends on the structure of the original program and the refactoring chosen. What might be considered to be a refactoring for one program might be an obfuscation[3] for another. We will therefore take the partial-automation approach mentioned earlier, and we rely on human guidance in applying the refactorings.

1.2 Objective

We seek to study the decisions that need to be made when verifying a refactoring *mechanically*, and the choices available for each decision. Two of the decisions which must be made early concern the *logic* and the *proof assistant* used. Sometimes these two decisions have to be made together since many proof assistants provide a single logic in which to work. There are many varieties of proof assistant to pick from, and some characteristics will be described in Chapter 3. We have chosen to work with the Isabelle proof assistant. This is a *generic* proof assistant, that is, it is designed to host a number of embedded object logics; of these we will use HOL, a higher-order logic based on Church's simply-typed theory.

To verify a refactoring we need to define the following in the proof assistant's input language:

- the programming language for which refactorings are being studied
- metalinguistic operations (e.g. substitution)
- program equivalence relation: the original and refactored programs must stand in this relation.

In later chapters we verify a number of refactorings for two languages and discuss improvements to the approaches taken. Through this we seek to contribute to the development of high-assurance correctness proofs for refactorings. Assurance is partly drawn from the correctness of the proof environment used (i.e. the proof checker must be correct and the logic used needs to be sound). The environment might also implement a facility to extract the refactoring from the proof, thus realising the implementation through the verification effort. This approach might seem to be high-effort and expensive, but can be justified since, as with all metaprograms, bugs in refactorings can leads to the propagation of bugs to other software.

[3]Like a refactoring, an *obfuscation* is a behaviour-preserving source-to-source transformation. Unlike a refactoring, an obfuscation is intended to make programs harder to understand (for security purposes). Drape (2004) describes data-oriented program code obfuscation, including correctness and efficiency criteria.

1.3 Organisation

The next chapter provides more background on refactoring, including an outline of past research and implementations of refactoring tools. Then in **Chapter 3** we describe mechanical theorem proving, in particular Isabelle/HOL, and techniques for encoding programming languages.

Chapter 4 discusses reasoning apparatus for program equivalence and describes the refactorings verified in this dissertation. **Chapter 5** presents the verification of a number of refactorings for a simple extension of the λ-calculus, and **Chapter 6** presents the verification of a type-based refactoring in an extension of PCF (Plotkin 1977).

Chapter 7 surveys other work to verify refactorings and outlines work done to verify other kinds of program transformations. **Chapter 8** concludes the dissertation by reflecting on observations made in previous chapters, particularly Chapters 5 and 6, and suggests directions for future work.

Chapter 2

Refactoring

The chapter starts with a chronological outline of research on refactoring. This is followed by descriptions of refactoring tools. In the second part of the chapter, refactoring is discussed more abstractly, two models of refactoring are described, followed by Robert's idea of using "postconditions" to improve the composition of refactorings.

2.1 Background

The observation that restructuring software makes it more reusable and benefits its maintenance motivated the practice of refactoring. The requirements of software might evolve during the software's lifetime, pressing the software to evolve too. Refactoring assists in adapting software for reuse or in adapting it prior to applying improvements/corrections (or possibly after such an activity). Opdyke (1992) and Griswold (1991) develop these observations in their theses, described in §7.

Opdyke and Griswold studied this idea for different languages: C++ and Scheme respectively. The nature of the programming language used influences refactoring since program analysis is part of the refactoring process: refactoring a program in a language is at least as complex as the analysis the refactoring needs to perform. This is because refactoring must check whether transforming a program structurally will indeed preserve its behaviour. Opdyke suggested structuring refactorings into **preconditions** and **transformations**, and Griswold studied the merits of combining information from two graph-based representations of programs to reason about behaviour-preservation.

Roberts et al. (1997) developed the Refactoring Browser: a tool for refactoring Smalltalk programs. It was the first refactoring tool implemented. Roberts (1999) contributed the idea of **postconditions** to reason about the composition of refactorings, and also that of **dynamic refactoring** (in which program analysis and program transformation performed at runtime). The nature of the language he used made it necessary for some refactorings to be applied dynamically: analysis is difficult to perform statically since Smalltalk is dynamically scoped and dynamically typed. Code can be transformed at runtime too because of Smalltalk's

reflection facility.

Fowler & Beck (1999) wrote the first book on refactoring and concentrated on refactoring Java programs. The book takes a practical approach and provides practical advice on integrating refactoring in one's code development routine. From the start the authors emphasise the need for testing. They advocate carrying out small, incremental code updates and testing them in tandem. *Unit testing* was a key component of this approach. This echoes Roberts et al. (1997)'s message about the importance of having good test suites, and that the results are only as good as the test suites themselves. Fowler's book contains over 70 refactorings for Java, specified (as one would expect) as precondition checks to make and transformations to apply.

Perhaps one of the reasons that refactoring became so popular is that it is compatible with the pervading idea that to some extent program code is in constant flux. That is, one never assumes code to be complete since it might be subjected to continuous modification in response to an evolved design, or to be reused elsewhere.

In recent years there was a strong push to treat refactoring more formally. This contributed to the possibility of verifying refactoring transformations rather then relying solely on the partial nature of testing. The need for testing is not likely to be eliminated with verified refactorings, at least in the near future, but it contributes to limiting the sources of errors. The goal of verification is to prove that refactorings are indeed behaviour-preserving program transformations.

Mens et al. (2005) formalised refactorings as graph transformations and the behaviour-preservation property as graph-property preservation. Cornélio (2004) studied the correctness problem from a (algebraic) "laws of programming"-based refinement view. Ettinger focuses on studying the use of *program slicing*, a technique originally used in debugging, for refactoring. Slicing involves extracting the subprogram (called "slice") that affects a particular variable in a program. This is difficult to automate but Ettinger suggests a new technique which he calls *sliding* and proves its suitability for refactoring (i.e. its behaviour-preservation wrt programs). These verification efforts, together with other related work, are surveyed in more detail in Chapter 7.

A large part of the work done in refactoring addressed object-oriented programming languages. As a result, one comes across many refactorings that address features of OO-languages such as type-based refactorings related to the object hierarchy, e.g., pushing methods up or down the hierarchy. However refactoring in the context of other paradigms has been pursued for several years too (cf. Griswold's work on Scheme described earlier).

Li (2006) wrote her doctoral thesis on the design and implementation of HaRe, a refactoring tool for the functional programming language Haskell. Li's work is described in more detail in §7.1. This work was produced in the context of the *Refactoring Functional Programs*[1] project at the University of Kent. The work described here is a continuation of this project.

[1]partially supported by EPSRC grant GR/R75052/01

2.1.1 Automation

Compilers were programmers' first tools and enabled them to encode abstractly and freed them from the need to target a specific machine. Similarly, refactoring tools enable programmers to devote more attention to the design of a program by facilitating its modification.

Manual refactoring is problematic: its repetitive nature makes it unsuitable to be performed by humans since it is highly vulnerable to error. This characteristic also makes it suitable for automation by machine since machines enforce rigour with ease. Many refactoring tools have been developed and some of these tools will be described next.

Refactoring Browser was the first refactoring tool. In the course of its development Roberts et al. (1997) had to answer questions such as "what are the characteristics of such a tool?" and "what makes it usable?" for the first time. For example, the authors sought to hasten analysis so that the programmers would not be discouraged from using this tool due to an inhibiting time-cost. They also worked to integrate the tool tightly with the program development environment to ease programmer access to the tool's functionality.

Garrido implemented refactorings as executable specifications in Maude and verified them. This was the first time refactorings were certified in this manner. Her work is described further in §7.4. Her implementations do not form part of a tool yet.

JunGL is a scripting language for specifying refactorings by declaratively describing their side-conditions and transformations in a language-generic manner. It is intended to enable programmers to create and alter refactorings with relative ease.

HaRe is a refactoring tool for Haskell, a functional programming language. HaRe also provides an API which programmers can use to extend its catalogue of refactorings.

Haskell is a statically-typed functional language with a Hindley-Milner typing system, and it also supports ad hoc polymorphism through *type classes*. Its other language features include static scoping, and *monads* to organise the semantics (in separating pure from effectful computation). HaRe works for the full Haskell'98 language specification. It has been integrated with the most popular IDEs among Haskell programmers. One of the practical features implemented in HaRe is code-layout preservation (including comments, indentation, etc) to make the program easily recognisable after refactoring.

Development on HaRe is being succeeded with the development of *Wrangler* – a refactoring tool for Erlang. Unlike Haskell, Erlang is highly used in industrial settings. Therefore it is necessary that tools for Erlang must not

only be experimental – they must be of very good quality if they are to be used. Erlang is a functional and concurrent programming language, and it is very different from Haskell in many respects. The experience acquired in developing HaRe is now being employed in developing Wrangler.

2.1.2 Testing and verification

Traditionally, programmers were advised to test refactored programs to ensure that behaviour was preserved. This form of testing is called *regression testing*.

Fowler advocated producing "self-checking code", based on a principle that "classes should contain their own tests". This practice is not specific to object-oriented programming. It is considered to be good practice to write test code incrementally, in tandem with adding features to a program. Test code is accumulated while the program is being written, and the tests can be combined together to test more features together. This approach is called *unit testing* and makes use of a framework to write *test cases*. The test cases are then collected into *test suites*. The framework facilitates writing and running the test cases quickly and easily, and the tests themselves contribute towards documenting the program. In this approach one distinguishes between *failures* (a failed test) and *errors* (a fault for which there was no test). Neither should be produced when running the tests after applying a refactoring.

Using this testing approach, the assurance of behaviour-preservation is only as good as the test suites' coverage. The process of generating test suites has also been automated. QuickCheck is a popular automatic testing tool that generates test cases for a program from a specification of its properties. The test cases are generated randomly and are used to find bugs by showing that the program does not satisfy the specification.

Recently research has been made into applying testing tools to the implementations of refactorings directly, rather than to the refactored programs. Daniel et al. (2007) test refactoring engines (implementations of refactorings) by generating abstract syntax trees to use as input to refactoring engines and check whether behaviour is preserved. The approach taken by Li & Thompson (2007) involves using QuickCheck and writing a specification consisting of properties that should be preserved by a refactoring. QuickCheck will then attempt to perform random refactorings on a program supplied by the tester and checks if the specification is violated.

Despite the availability of better tools and methods to test refactorings, testing cannot prove a refactoring correct. This can be proved by *verifying* the refactoring: formally proving that it is behaviour-preserving for all programs. This comes at the expense of increased difficulty, especially for realistic programming languages. This approach involves formalising the semantics of a programming language and proving the refactorings correct modulo these semantics. The features of the programming language greatly influence the ease with which refactorings can be verified. Chapter 7 surveys research on the verification of refactorings.

2.2 Notions

Since the first thesis on refactoring (Opdyke 1992), a refactoring has been defined to consist of a set of side-conditions and a program transformation. The side-conditions are conjoined into a precondition to program transformation: unless all the side-conditions are satisfied, the transformation is not effected.

Let $\simeq$ be a semantic equivalence relation over programs and T an endofunction on programs (i.e., the transformation operation). The requirement for refactorings to be behaviour-preserving intuitively suggests the following formulation:

$$\lambda p.\ \mathsf{if}\ (T\,p \simeq p)\ \mathsf{then}\ (T\,p)\ \mathsf{else}\ p$$

That is, if the transformed program is equivalent to the original program then transform the program otherwise return it unchanged. However, as widely known, the antecedent is undecidable for nontrivial languages (unless their theory is inconsistent), so refactorings cannot be implemented in this manner. Let Q be an effective predicate defined over programs. If Q is sufficient for $(T\,p) \simeq p$ then the following formulates a behaviour-preserving program transformation:

$$\lambda p.\ \mathsf{if}\ (Q\,p)\ \mathsf{then}\ (T\,p)\ \mathsf{else}\ p$$

Then in order to verify a refactoring one must prove that:

$$\forall p.\ (Q\,p) \longrightarrow (T\,p) \simeq p$$

The correctness of the refactorings verified in Chapters 5 and 6 is formulated in this manner.

One could say that refactorings "come in pairs" since a refactoring and its inverse are both refactorings. The inverse of a refactoring is used implicitly when *undo*ing a refactoring. However the specification of a refactoring often favours moving in one direction since the side-conditions are defined over the original program rather than the refactored program. The reason for this is that defining side-conditions over a transformed program might be disadvantageous: if the transformed program fails the check then the effort spent transforming it would have been wasted. This will be discussed further in later chapters.

2.2.1 Interact vs. Compensate

Refactorings can be implemented to behave in two ways: they can *interact* with the user to ensure program behaviour-preservation, or the user might have to *compensate* for additional changes effected by the refactoring. These two models were described briefly in the Preliminaries chapter.

Both models will be illustrated using automata having the IDE as their initial state. The compensating (also known as *conservative*) model works thus: if preconditions fail then the transformation may *still* take place but only after some other code has been transformed such that the overall transformation is

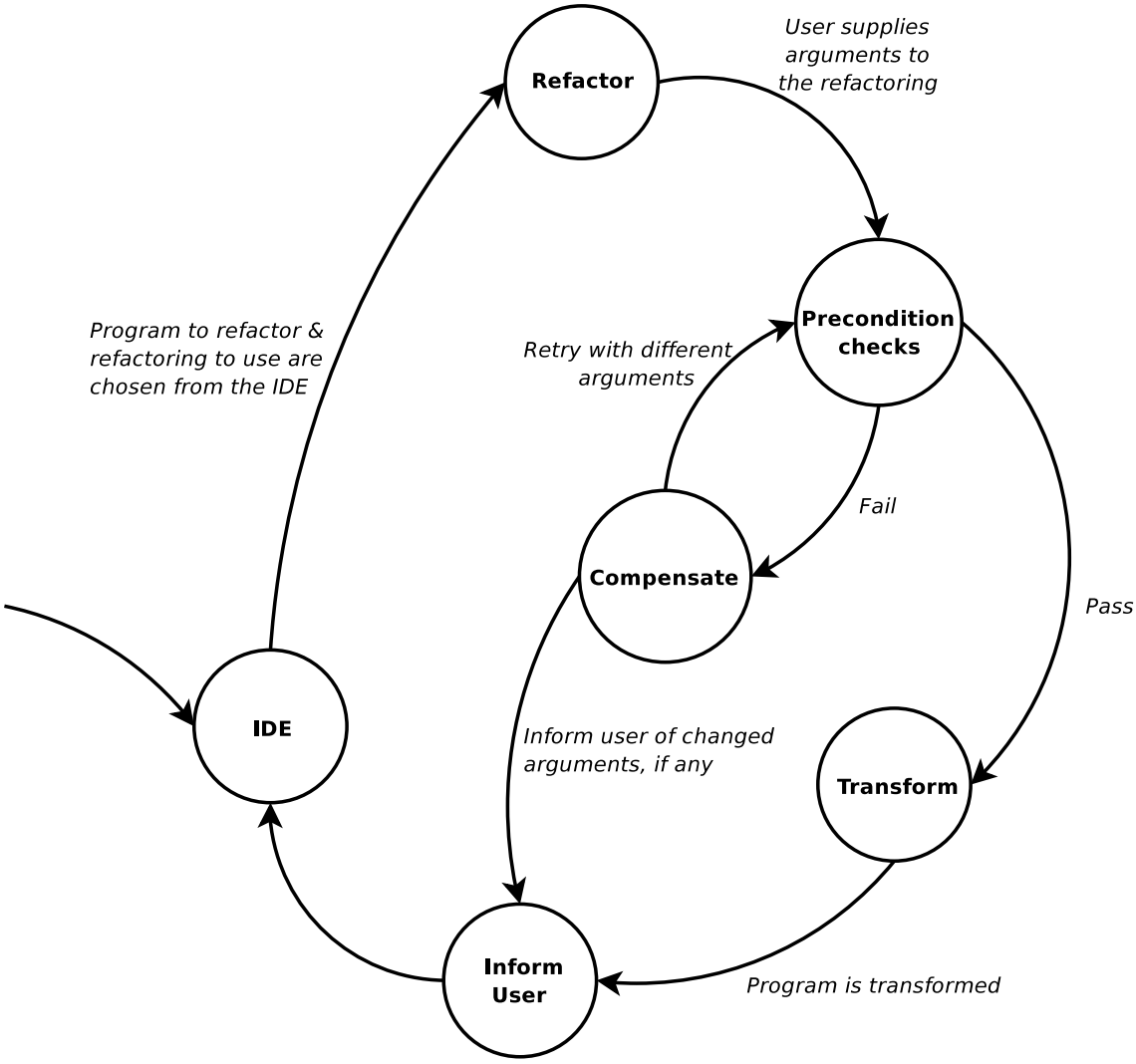


Figure 1: Compensation-based refactoring

behaviour-preserving. The user is informed about this change after the refactoring is complete and might then have to transform the code again (e.g. to rename some variable name that was picked by the refactoring tool).

This model is illustrated in Figure 1. Note that the refactoring tool is invoked from the IDE and is passed a program and the chosen refactoring as parameters. The user provides any further arguments needed by the refactoring, and the refactoring tool runs the refactoring.

The interactive approach, modelled in Figure 2, uses less automation and more user involvement. Should the side-condition checks fail then the user is asked to choose between aborting the refactoring or else providing different arguments so that the refactoring may be attempted afresh.

The interactive approach requires that more assumptions are explicit and

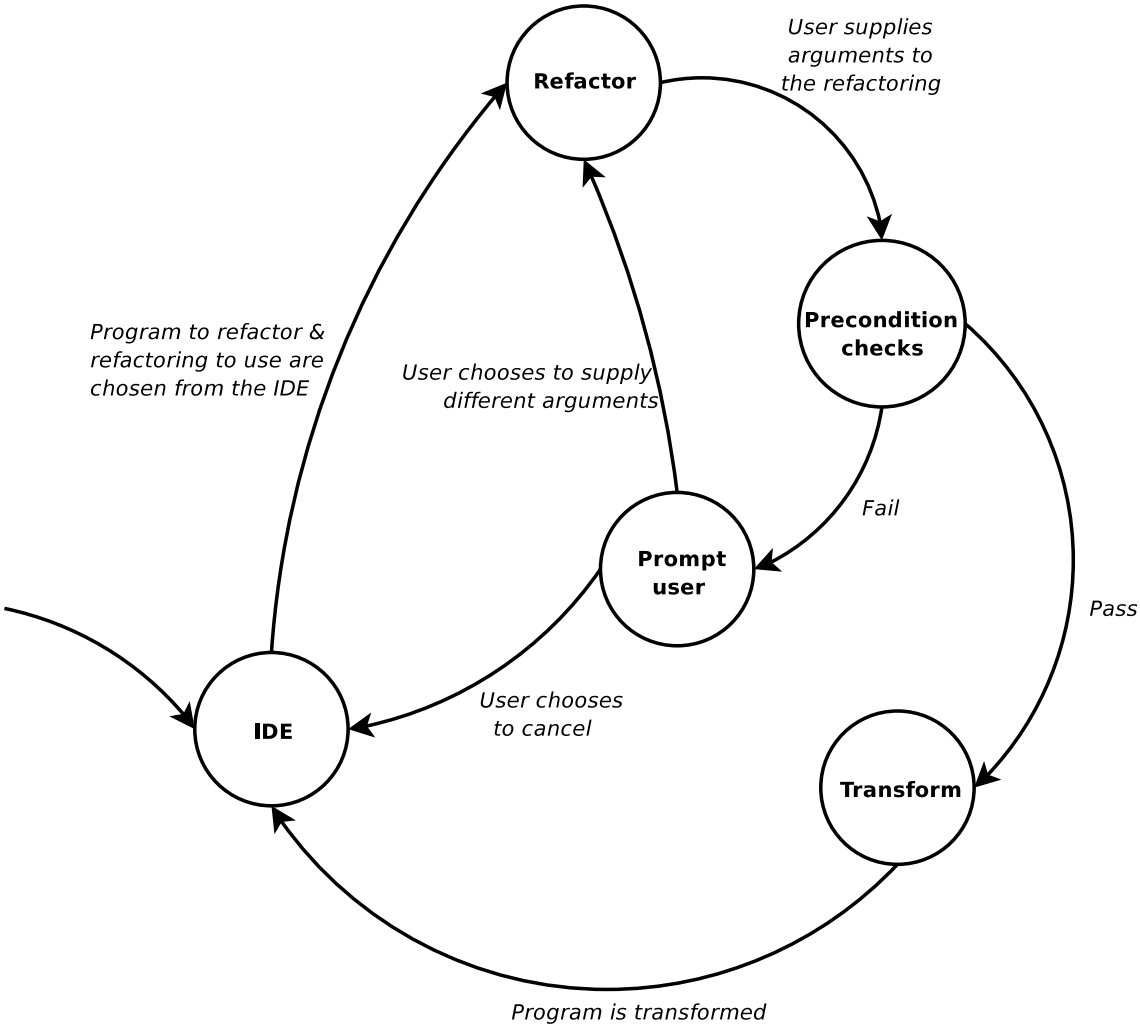


Figure 2: Interactive refactoring

checked before the transformation is attempted. The compensation approach discharges some assumptions and transferring them elsewhere: for instance, it would require verifying an algorithm that produces fresh variable names relative to a specific expression.

We favour the interactive model because it avoids leaving too much to the machine. Automation is useful only to a certain extent; automating the wrong aspects runs the danger of becoming a nuisance to the user. The compensating model can be seen as an extension of the interactive model with further automation. The interactive model affords the user greater control.

2.2.2 Composition of refactorings

Roberts (1999) contributed the idea of postconditions to facilitate the composition of refactorings. Since refactorings are endofunctions it is natural to imagine their composition to be ordinary function composition.

Compound refactorings are useful to programmers since they reliably automate complex program transformations. This saves the programmer having to manually apply the various smaller refactorings in a composite refactoring. Many useful refactorings can be constructed this way – indeed Kniesel & Koch (2004) see this as an accessible method of constructing correct refactorings. Their work is described in §7.6.

Let $\fatsemi$ denote (sequential) function composition, and let r_1 and r_2 be two refactorings. We can define a new compound refactoring, r, as follows:

$$r \triangleq r_1 \fatsemi r_2$$

Refactoring r consists of applying refactoring r_1 then refactoring the result using r_2 . Recall that $\triangleq$ is being used to denote abbreviations. However this definition of refactoring composition is naïve, since it might be the case that when applying r only r_1 is carried out. That is, if after refactoring using r_1 the resulting program does not satisfy the side-conditions of r_2 then r would be equivalent to r_1. In order to ensure that the compound refactoring is applied whole we can collect the side-conditions together at the start so that a program is accepted for refactoring only if it will be refactored by each elementary refactoring in turn.

Continuing this example, let the respective side-conditions be denoted as Q_1 and Q_2 and the respective transformations be T_1 and T_2. The compound refactoring is expressed as follows:

$$r \triangleq \lambda p.\, \mathsf{if}\ ((Q_1\ p) \wedge (Q_2(T_1\ p)))\ \mathsf{then}\ ((T_1 \fatsemi T_2)p)\ \mathsf{else}\ p$$

This definition ensures that programs given to r_2 from r_1 pass the side-condition checks. However this compound refactoring would be improved if its side-conditions could be formulated to address the original program, rather than the transformed program or an intermediate step – if possible we would like to eliminate side-conditions like $Q_2(T_1\ p)$. Roberts (1999) put forward the idea of deriving refactorings' *postconditions* and using them to discharge preconditions of refactorings they are composed with.

Let P be a predicate. Establishing that P is a postcondition of T_1 entails proving the theorem $\forall p.\ P(T_1\, p)$. This formula may be weakened since it is unnecessarily strong: we are not interested in *all* programs being transformed by T_1 but only in a bounded universal quantification. That is, we are only interested in programs that satisfy the side-conditions of r_1. Therefore we can instead prove the weaker theorem $\forall p.\ (Q_1\, p) \longrightarrow P(T_1\, p)$.

Let P_1 be a postcondition for refactoring r_1. If r_2 has Q_2 as $q_1 \wedge \ldots \wedge q_n$ and if it can be shown that $P_1 \longrightarrow q_i \wedge \ldots \wedge q_m$ then we can weaken Q_2 by dropping the conjuncts $q_i \wedge \ldots \wedge q_m$. Ideally $Q_2(T_1\ p)$ can be shown to be implied by Q_1 since in practice this would allow us to avoid having to transform programs before checking them. It is desirable to avoid checking transformed programs since if the check fails then the transformation would have been wasted effort. Having the side-conditions of a compound refactoring defined in terms of the original program, rather than intermediate steps, improves the performance of refactoring.

Chapter 3

Computer-assisted theorem proving

Automated reasoning support is crucial for large formal developments since computers assist in both checking and organising the development. Such tools, called *proof assistants*, have been used for both general mathematics and also specifically in computer science to verify models of software and hardware.

Proof assistants often provide services beyond proof checking by offering a suite of tools that automate parts of the proof development, for example, solvers or proof search tools. This facilitates proof development by carrying out deduction steps on the user's behalf – for instance by term rewriting. Depending on the system used, it might be up to the user to ensure that such a procedure would terminate by not adding rules causing it to diverge. The proof tools might support different modes of input; for example, proofs might be written in a stylised fragment of natural language or as a sequence of *tactics* (instructions that transform the proof state). A proof assistant might also interface with other tools – for instance, interactive tools might invoke automatic tools as oracles to solve parts of a problem, or might mark up the formalisation for LaTeX rendering. Some proof assistants also provide facilities for extracting and optimising computational content from proofs.

There are a variety of proof tools available; Wiedijk (2006) compiled proofs of the irrationality of $\sqrt{2}$ checked by 17 principal proof assistants. Some of these tools address a specific niche or have distinct characteristics. For example they might be intended for educational use rather than for large developments, or they might be especially suitable for computer science-related formalisations. Some tools have gathered sizable communities or accumulated a large quantity of contributed mechanisations.

Usage of proof tools is increasing steadily; this is partly due to maturing tools and the increasing power of commodity computers, better user interfaces and better support through an active community. Aydemir et al. (2005) sought to find out the current state of affairs with regards to the ease of mechanising results about programming language theory – in particular they use the language $F_{<:}$ (System F extended with subtyping) as a kind of "litmus test" for tools and techniques. This initiative stems from a hypothesis that mechanically verified

software will become more commonplace.

This chapter will provide some background to mechanical theorem proving, focusing on Isabelle/HOL, and will discuss the use of such tools for reasoning about programming languages.

3.1 Development of proof tools

In 1954, Davis (2001) implemented the first theorem prover, using Presburger's decision algorithm. The AUTOMATH project started in 1968 and was one of the first large projects for mechanising mathematics; techniques invented during its development are still in use today. Development on the system LCF (Paulson 1987) started in 1972 and addressed the program logic described by Scott (1993) (circulated as an unpublished manuscript since 1969). LCF has had many descendants, many of which replaced the original logic by more expressive systems. The system Mizar (Rudnicki 1992) started in 1973 and was intended for the mechanisation of mathematics. Unlike AUTOMATH and LCF, the Mizar project is still active. One of the Mizar system's characteristics is its input language: a stylised fragment of English used in mathematics. Each of these three systems was innovative and had a large influence on future systems.

The number of tools available makes choosing a proof tool more difficult. While the comparison in (Wiedijk 2006) is based on proof scripts, reviews such as that by Grioen & Huisman (1998) examine several aspects of the tools from a practical angle. Such a "consumer's report" tabulates criteria on which tools were compared and helps to inform newcomers of the characteristics of specific tools.

In the work described in this report we use Isabelle/HOL: an embedding in Isabelle of HOL (a system of classical higher-order logic based on Church's simple type theory and used in the HOL proof environment (Gordon & Melham 1993)). Isabelle is described in more detail in the next section. Like many other proof tools, Isabelle is heavily influenced by LCF. The connection between LCF and Isabelle has been outlined by Gordon (2000); we summarise this next. In §3.1.4 we explain why Isabelle is suitable for our work.

3.1.1 LCF

LCF was a first order logic of domains originally described by Scott in 1969, but the name ("Logic for Computable Functions") is due to Milner. In this logic types corresponded to domains. While in Stanford, Milner started the project to develop a computer-based proof checker for LCF and the resulting system was later known as Stanford LCF. After moving to Edinburgh, Milner continued improving Stanford LCF and thus Edinburgh LCF was created. One of the significant differences between Edinburgh LCF and its predecessor was its extensibility: the system was defined within a meta-language, the functional programming language ML, designed for the sole purpose of hosting it.

Since then ML has evolved into an independent programming language (Milner 1997). ML fragmented into different dialects due to its popularity, but in the late 1980s it was standardised by means of a canonical definition. Moreover, its definition was formal: ML was the first programming language of its size having a formal definition.

One of the innovative features of ML was its type system (known as the Hindley-Milner type system); this was used to specify an abstract datatype "thm" of theorems. Each theorem was a value in this type, and proofs were terms that constructed this type. Users carried out proofs by using *tactics* (higher-order functions applied to simplify the proof goal). Checking the proof entailed type-checking the proof term; this is decidable for the Hindley-Milner system.

It is impressive that although ML was designed to accommodate LCF it had a profound effect on computer science in several ways. ML influenced the designs of various other functional programming languages. Moreover, ML and its descendants (e.g. OCAML) still serve as meta-languages to descendants of LCF (e.g. HOL, Isabelle, Coq).

When a proof tool is described as being *LCF-style* it means that the tool is implemented as a library in a (statically typed) programming language and it uses an abstract datatype of theorems which guarantees soundness. Initially users had to embed their proof developments in the meta language too: this involves writing ML code and making calls to the proof tool. Gradually the underlying ML layer was concealed from users of the proof assistant by enabling all the contents of a theory to be encapsulated in a theory file. Some assistants also offer facilities for users to implement tactics in the theory file itself rather than resorting to the metalanguage. Tools in the LCF tradition also tend to adhere to the *de Bruijn principle*: they rely on a small "trusted core" (kernel) which can be checked manually for soundness, and some of these tools produce proof objects that can be checked by independent tools. This principle was suggested by Nicolaas de Bruijn, the architect of the AUTOMATH system mentioned earlier.

3.1.2 HOL

The Cambridge LCF system was an improvement on the Edinburgh system. In a separate project concerned with studying hardware verification (rather than programs) the LCF system was tweaked to support a "Logic of Sequential Machines" that was devised by Mike Gordon. The resulting system was called LCF-LSM. LCF was not suitable for reasoning about hardware since extra results concerning strictness and definedness had to be proved; these were relevant for software but less so for hardware. For the first version of HOL, LCF was adapted to use a classical higher-order logic and the HOL system inherited LCF's efficient implementation of its algorithms. The HOL system had several offspring and is still being developed. Proof developments in HOL are not restricted to hardware verification; indeed higher-order logic has been found suitable for very diverse formalisations. The HOL system's influence is also appreciable from how an informal gathering of HOL users is now an international conference (TPHOLs).

3.1.3 Isabelle

Isabelle was intended to be an extensible LCF-style proof environment that would lessen the amount of ML code required to embed a new object logic. It provided a logical framework consisting of a minimal intuitionistic higher-order logic (referred to as Isabelle/Pure). Various logics have been embedded in Isabelle, e.g., FOL, Constructive Type Theory, LCF, HOL, etc. The most developed embedding is HOL, a classical higher order logic based on the logic used in the system HOL (described previously). Isabelle provides several facilities at the metalogical level and these facilities (e.g. unification) are inherited by object logics or can be instantiated (e.g. simplifier, program extraction (Berghofer 2003)).

Initially Isabelle was used as an ML library and users divided their formalisations into "definitions" and "proofs"; definitions were put in a "theory" file and proofs were written as ML code. In more modern versions of Isabelle the proofs migrated to the theory file. Isabelle users need not code or know ML unless they need to implement tactics or other new functionality in Isabelle. Theory files are more user-centric: they are less cluttered and easier to read than raw ML code since they hide the lower language levels. Users can write proofs using different styles:

Procedural style involves using tactics to refine goals into simpler subgoals. Using this style involves backwards reasoning most of the time.

Declarative style involves using a stylised fragment of natural language (overlapping with the vernacular of mathematics) to construct proofs by forward reasoning. This proof style is more natural for mathematics and more readable than tactic scripts. It also enables more modular proof development and proofs are more easily changed and reused. In Isabelle this mode is called *Isar* (for "Intelligible semi-automatic reasoning") and is inspired by the input language used by Mizar. It is described in detail by Wenzel (2002) and a compact overview is given by Nipkow (2003).

The suitability of which proof style to use depends on the nature of the formalisation; for example, backwards reasoning seems more suitable for tasks such as verifying microprocessors (Gordon 2000, §7.3). Another form of proof style involves building *proof terms*, but Isabelle does not allow the user to manipulate proof terms directly. However due to work by Berghofer (2003) proof terms can be extracted from Isabelle proofs.

As with other LCF-style systems, proof checking is done by means of type-checking and is decidable. Users can build proofs interactively by querying the proof state and simplifying goals (backward reasoning), or else generating facts from other facts until the theorem is proved (forward reasoning). Isabelle has a larger kernel than the HOL system since Isabelle's kernel also includes higher-order-unification code, but using Berghofer (2003)'s proof-term-extraction mechanism the proofs can be checked by a system with a much smaller core. This improves assurance of the proofs' correctness. Technical details of Isabelle are

provided in Paulson (1994) and Nipkow et al. (2002) provide an accessible tutorial on Isabelle/HOL.

3.1.4 Choosing Isabelle/HOL

It would have been possible to work in other environments but Isabelle/HOL was a very appealing candidate. Like some other systems, it is a mature product and has undergone many years of development. This development was also directed at its usability. For instance, it has a Proof General (Aspinall 2000) interface and it also comes with a suite of tools to manage the organisation and presentation of formal developments. Further work (Haftmann et al. 2005) was done to improve the presentation of LaTeX scripts generated from theory files.

One of the reasons for Isabelle's maturity is that two sites (University of Cambridge and Technische Universtät München) pooled their resources and invested in its development; this in turn facilitated the gathering of a sizable community of Isabelle users.

We chose Isabelle/HOL also because of the following reasons:

- It is a "safe" system (i.e. its design employs the de Bruijn principle); this is desirable for the verification of metaprogramming systems such as refactorings.
- Various other related work (on the verification of program transformations) has been done using Isabelle/HOL. Using Isabelle/HOL facilitated comparison and learning from related work.
- The availability of packages such as the nominal datatype (Berghofer & Urban 2007) is particularly mature in Isabelle. This affords greater flexibility when choosing techniques to use.

3.2 Language encoding

Irrespective of the proof environment chosen to host a formalisation, one could choose between different techniques to encode the language.

The languages specified in Chapters 5 and 6 will be encoded using first-order abstract syntax and concrete variable names. This enables reasoning about names and is suitable for our purpose. There are many other techniques to pick from. Some of these techniques were described in the Preliminaries section. As explained then, one cannot really speak about a technique being "better" than another but rather that it might be more suitable to solve a particular problem Some more techniques are outlined next.

Berkling keys is a hybrid method due to Klaus Berkling that indexes *named* variable occurrences with the distance to their binding occurrence. The distance is relative to each name; this is achieved through having "binding

contexts" for each scope and variables with the same name must specify which (embedded) binding context they refer. These indexes (keys) serve to "protect" the names from capture during substitution: the names themselves are not changed but the variables are made to refer to the appropriate context (similar to the "shift" operation in de Bruijn indices). This technique is described further in (Reinke 1997).

Higher-order abstract syntax (Pfenning & Elliot 1988) exploits a meta-language to define binders in the object language in terms of binders in the meta-language. Such an embedding is *shallow* due to the dependence of the object language on the meta-language for some of its clauses. This technique had been developed for embedding deductive systems.

Explicit Substitutions : Substitution is classically an *implicit* operation, i.e. it is a metalinguistic operation. The key idea in explicit-substitution calculi (or $\lambda\sigma$-calculi) is that substitution becomes part of the language and avails itself to explicit reasoning.

The $\lambda\sigma$-calculi (Abadi et al. 1990) provide a theory of substitutions along with the usual theory for λ-calculi. The syntax in these calculi is composed of two categories: expressions and substitutions. Substitutions can be composed, appended to substitution-lists and applied to expressions.

McKinna & Pollack (McKinna & Pollack 1993) emulate the Barendregt convention by separating free and bound variable names. This approach is called *locally named*.

Gordon/Melham axioms (A. D. Gordon & T. Melham 1996) are five axioms (validated as theorems using the HOL system) for nominal reasoning about α-equivalent terms in the untyped lambda calculus.

Nominal Logic (Gabbay & Pitts 1999) is recent work that seeks to introduce the notion of "fresh" variables in the logic thus avoiding many of the problems associated with name-based methods. In this theory the central operation is *swapping* (permuting variables) rather than substitution. This work has been partially implemented for Coq and for Isabelle (Urban & Tasson 2005).

Part II

Chapter 4

Verifying refactorings mechanically

This chapter discusses our choice of programming language and method to verify refactorings. In the next two chapters a number of refactorings will be verified. These refactorings will be described at the end of this chapter and in the next chapters we will focus on the verification itself.

4.1 Reasoning about programs

4.1.1 Languages

In order to verify refactorings for a language we must first formalise that language's semantics. Not all programming languages have a formal definition and this inhibits reasoning about the correctness of their programs. The correctness of algorithms that manipulate programs in the language cannot be checked either, and the usual approach in this case is to formalise a subset of the language.

Refactorings have been studied for various languages and in the chapters that follow we will focus on studying refactorings for functional programming languages. Functional languages have contributed to the precise understanding of programs and have also benefited greatly from research made into other language-paradigms. This cross-fertilisation has led to multi-paradigmatic languages which offer improved flexibility and expressiveness to programmers compared to early programming languages.

The semantics of functional languages are usually easier to formalise and the programs in this paradigm are considered to be particularly amenable to mathematical reasoning. Rather than being pure doctrine there seems to be evidence for this: the first non-toy language to have a formal definition was a a functional language – Standard ML (Milner 1997). It is also likely to be the first language of its size to have a fully mechanised definition (Lee et al. 2007). Other than being of purely academic interest, functional languages are also useful (cf. the oft-cited

manifesto by Hughes (1989)) and have been applied to several non-academic problems. Despite the suitability of functional programs for mathematical analysis, they were isolated from practical use for a while because it was difficult to combine their inherent purity (in terms of computation) with the real-world through input/output. This was addressed in the thesis by Gordon (1994). The λ-calculus is considered to be a canonical functional language and serves as a springboard to more sophisticated languages (when enriched with additional primitive notions). The pure calculus is far removed from any recognisable programming language. A reasonable compromise involves bringing the λ-calculus closer to a "real" programming language by extending it with more primitive features. In Chapters 5 and 6 we verify refactorings for two extensions of the λ-calculus.

4.1.2 Program equivalence

There are many criteria by which programs can be considered to be equivalent. Some definitions for equivalence between programs will be described next. Wherever reference is made to a theorem in this section but no citation is provided then a reference to the treatise by Barendregt (1981) is intended by default. Let the set of programs be the set of closed λ-terms, denoted by Λ_0.

Definition 4.1.1 *A term t is said to be* solvable*, denoted by $t\swarrow$ if it can be reduced to head normal form, otherwise it is said to be* non-solvable*, denoted by $t\nwarrow$.*
The term $(\lambda x.\ x\,x)(\lambda x.\ x\,x)$ is the canonical non-solvable term and it is usually abbreviated by the symbol Ω. Non-solvable terms are not considered to have any meaning and are usually equated with Ω.

As a weak definition of equivalence, we can consider equivalent all solvable terms, and equate all non-solvable terms.

Definition 4.1.2 Solvability equivalence*, denoted by $\equiv\!\downarrow$, is the largest equivalence between terms such that $s \equiv\!\downarrow t$ iff $s\swarrow$ and $t\swarrow$, or $s\nwarrow$ and $t\nwarrow$.*

As previously explained there are many definitions of equivalence that can be used. The definition we have just seen is not useful in practice, and the next definition is a better candidate.

Definition 4.1.3 *For a reduction relation R we define* R-convertibility*, denoted by $=_R$, as the reflexive, transitive and symmetric closure of R such that $s =_R t$ iff $(s,t) \in R^{*=}$.*

In order to derive theorems of convertibility a proof system is defined as for $\alpha\beta\eta$-convertibility in the Preliminaries chapter.

A weaker equivalence abstracts away the structure of terms and focuses on their *behaviour*. It is intuitive to formalise this equivalence in terms of contexts; this was first done by Morris (1968). The definition requires programs to reduce

to the same normal form *in all contexts*, or else diverge in all contexts. In fact, we do not actually require normalisable terms to converge to the same normal form since if two terms converge to different values then there will be a context that distinguishes them. This is made precise in the following definition.

Definition 4.1.4 Contextual equivalence*, denoted by* $\equiv_C$*, is the equivalence between terms such that* $s \equiv_C t$ *iff* $\forall C.\ C[s]\!\downarrow \longleftrightarrow C[t]\!\downarrow$.

Quantification over all contexts makes this equivalence difficult to prove. It turns out that we do not need to check all contexts, but only specific contexts. This is due to the Context Lemma, described next.

Definition 4.1.5 *An* applicative context *is single-holed context with a hole at head position and applied to any number of terms (e.g.* [], ([] t), ([] $t\ t'$)*, etc, for arbitrary* $t, t', \ldots$*, are applicative contexts).*

Lemma 4.1.6 (Context Lemma) *Let* C_A *range over applicative contexts, then* $s \equiv_C t$ *iff* $\forall C_A.\ C_A[s]\!\downarrow \longleftrightarrow C_A[t]\!\downarrow$.

This approach is described in detail by Abramsky (1990), and Pierce (1997) provides an overview of operational program equivalence. The Context Lemma captures the intuitive notion of programs behaving in the same manner "for all inputs".

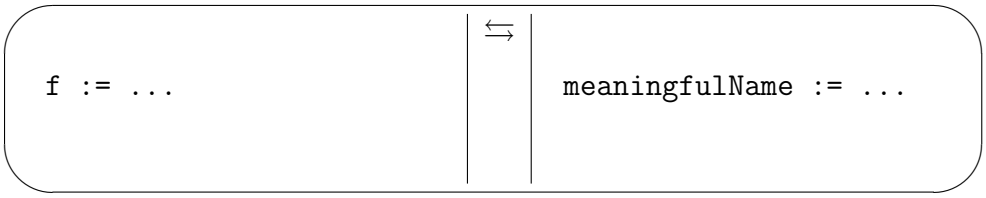
Note that convertibility is an *intensional equivalence* (i.e. convertible terms encode the same algorithm) and behavioural equivalence is an *extensional equivalence* (i.e. related terms fulfil the same function). We will use an intensional equivalence in the formalisations described in the next two chapters, but we discuss other work based on extensional equivalence further down and in §7.7.

4.2 Refactorings verified

In the following two chapters we describe the verification of the refactorings "**extract a definition**" and "**enlarge definition type**". The first refactoring is composed of several smaller refactorings which are verified separately and then used to verify the compound refactoring. We will describe all the mechanised refactorings below using pseudocode fragments to illustrate their transformation. The refactorings will be formalised and their side-conditions explained in more detail in the following chapters.

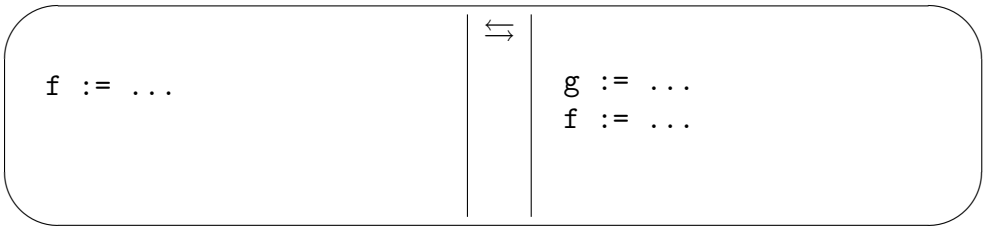
In the following code fragments the left pane illustrates the original program and the right pane suggests the transformation effected by the refactoring. The two panes are separated by the symbol "$\leftrightarrows$" to suggest that the refactoring can be applied in both directions. The code fragments do not necessarily show all the transformations effected by a refactoring but the full transformation will be described in the accompanying explanation.

4.2.1 Rename a definition



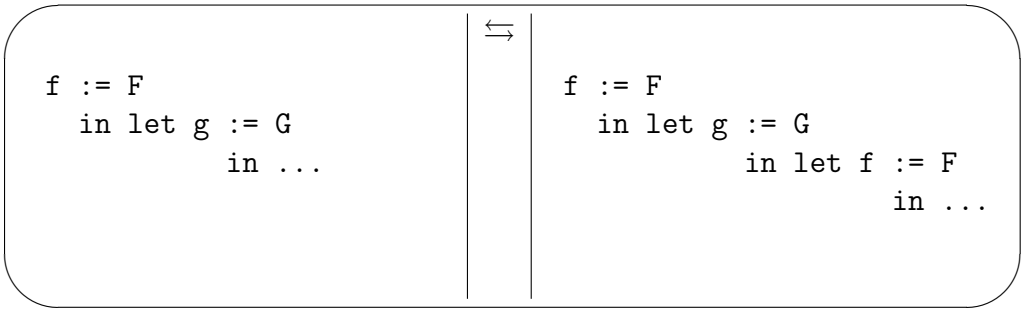
This refactoring simply renames a definition. The new name chosen must not appear free in the body of the expression otherwise the renaming would lead to name capture. The new name must not already be defined in the same scope otherwise this would lead to name clash. All the call sites of `f` must be updated to use the new name. This refactoring is verified on page 45.

4.2.2 Add/drop a redundant definition



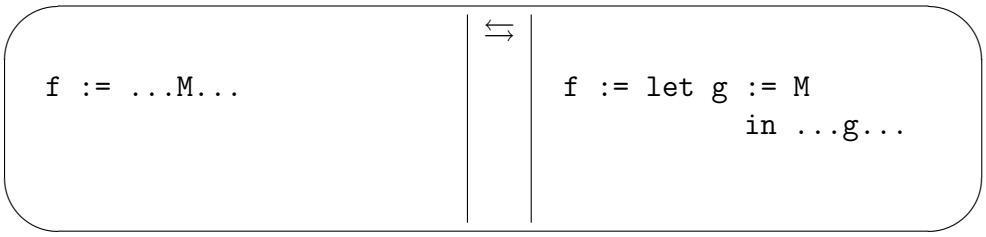
This refactoring adds or removes a definition; the variable bound to this definition (i.e. the name of the definition) must not appear free in the body of the expression and must not clash with other definitions in the same scope. This refactoring is verified on page 46.

4.2.3 Demote a definition



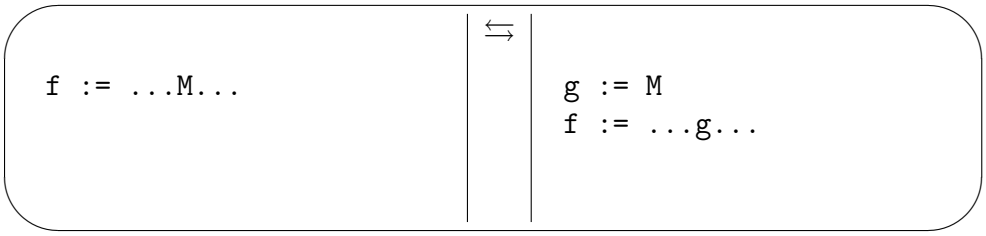
This refactoring reproduces the outermost definition inside the definition directly below it. We look at a particular instance of this refactoring; as mentioned by (Li 2006, §2.8) it is not uncommon to find varying definitions of similar refactorings. This refactoring is verified on page 48.

4.2.4 Declare/inline a local definition



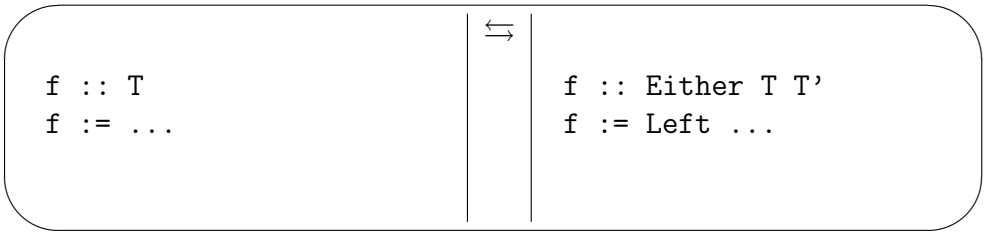
This refactoring produces a local definition from a subexpression, or inlines a definition in all its calling sites (i.e. in all free occurrences of the variable with which it is bound). This refactoring is verified on page 53.

4.2.5 Extract a definition



"Extract a definition" is a non-trivial, compound refactoring which we define by composing the previous three refactorings using the transitivity rule. This refactoring is verified on page 56.

4.2.6 Enlarge definition type



"Enlarge the definition type" is a type-based refactoring that transforms a definition of a certain type into a coproduct with the original term as a left injection. This refactoring might be useful for adapting code prior to extending its functionality to make use of the broader type.

Although the refactoring itself is straightforward, its verification requires prior proof obligations concerning the type system. In particular, the Substitution Lemma needs to be proved – this establishes that substitution is type-sound. This refactoring is verified on page 68.

4.3 Techniques

The proofs of the theorems described in the next two chapters have been checked by Isabelle – specifically, the proofs have been checked using Isabelle2005. The theory files are available on the accompanying CD. The LaTeX scripts for the next two chapters have also been generated using Isabelle's suite of tools and its presentation facilities. This helps ensure consistency between the mechanisation and the presentation of the results. As well as verifying the refactorings we have sought to communicate them (i.e. their side-conditions and the transformation they effect) in a compact yet understandable manner.
We now discuss the techniques used in the next two chapters.

Language definition

We embed the languages in the next two chapters in different ways and make observations on the effect this has on proofs. As described in §3.2, the technique used to encode both languages is name-carrying first-order abstract syntax. In the second formalisation we discuss a slight weakening of the side-conditions to admit more realistic programs for refactoring. We will also discuss increasing the level of abstraction by anonymising the language's syntax and the price this brings on soundness since it precludes being able to catch problems related to name capture (there would be no names to reason about). On the other hand this could significantly simplify the proof process and afford more time to be spent on verifying other aspects of the refactoring, such as type-soundness. Such an approach would be comparatively "lightweight".

Proving equivalence

The semantics of programs will be described equationally (since only the equational fragment of the program logic is required for the task at hand) and results will usually be proved by induction on the structure of expressions. Reduction steps are done explicitly using the (β) and transitivity rules. Since the logic is embedded in HOL we use the quantifiers and connectives at the HOL-level to formalise side-conditions of the rules of the equational logic. For the second language we have used a logic described in the book by Gunter (1992).

An alternative approach would have involved specifying a weaker relation than convertibility, as described in §4.1.2. This would have required different mathematical machinery, including the prior development of a tractable method to prove equivalence coinciding with the weak equivalence. Gordon (1994) and Pitts (1995) provide examples of this approach. Through this other approach one could verify more transformations, but verifying transformations using extensional equivalence is generally more challenging.

Extensional equivalence is usually formalised as contextual equivalence, but due to the quantification over all contexts the method is intractable for even simple languages. Finding the right induction hypothesis to use in an induction on

the structure of contexts is not easy. More tractable methods have been sought, and one solution borrows from Labelled Transition Systems and the Calculus of Communicating Systems. Using this method programs are related using a bisimulation relation. This equivalence is then proved to imply contextual equivalence (Pitts (1995) uses this method for PCF extended with streams). Coinductive reasoning is used to prove programs equivalent using this more tractable method. Moreover, this technique is also ideal for reasoning about languages in which nonterminating systems or potentially infinite data is defined. Glesner et al. (2007) mechanise proofs using this method, as described further in §7.7.

Chapter 5

"Extract a definition"

5.1 Introduction

The goal of this chapter is to verify the *extract/inline a definition* refactoring. Since it is a compound refactoring, leading up to its verification will be the verification of some other refactorings that are its intermediate steps. Reflecting the relationship between compound and elementary refactorings, the correctness proofs of the elementary refactorings are lemmata to the proof for the compound refactoring.

As is common with formal developments, a large number of lemmata are generated in order to support higher-level reasoning. We concentrate on describing the definitions and main theorems here.

We rely on the proof assistant to check the proofs and, assuming inference to be sound, if there are any bugs then they must be in the definitions we use. The presentation given here is consistent with the actual proof development since this chapter was generated from the scripts.

The presentation at times reveals the underlying systems in which this work was embedded. We will reason about programs using an equational logic embedded in HOL, which in turn is embedded in Isabelle's logical framework. HOL's connectives and quantifiers are used in the specification of theorems, leaving the embedded logic very simple.

A simple programming language will be defined in the next section. Expressions in this language are distinguished from terms in the metalanguage through the use of a sans serif typeface for metalanguage terms. For example, consider the term:

$$\mathsf{if}\ (x \in DVTop\,(letrec\,d \cdot e))\ \mathsf{then}\ (letrec\,(d[M/x]) \cdot e)\ \mathsf{else}\ letrec\,(d[M/x]) \cdot (e[M/x])$$

In this term, "*if..then..else*" is a term in the metalanguage, "$(letrec\ d \cdot e)$" is a term in the object language, and "$(x \in DVTop\,(letrec\,d \cdot e))$" is a proposition in HOL.

We proceed as follows: the programming language we study will be described, and various definitions will be provided to formalise notions required to assert the correctness of refactorings. Several structural refactorings will be verified and

some of these refactorings will then be composed together into the "extract a definition" refactoring; the proof of correctness for this refactoring will also be a composition of the proofs for its constituent refactorings. An experimental style is adopted in the formalisation: different techniques are tried and their outcomes compared. In §5.4.2 we discuss improvements to the approach taken to define the language – e.g., the (de)merits of the definition used for fixpoint expressions. These observations will be discussed at the end of the chapter and used to improve the approach taken in the next chapter.

5.2 Metatheory

5.2.1 Language

The language defined below is the λ-calculus extended with recursive definitions. Conventionally, `letrec` is used to declare recursive definitions. The language is defined as two syntactic categories: that of expressions and that of declarations. These are mutually recursive. The language is inspired from that used by Li (2006).

Let V be a denumerable set of variables, ranged over by metavariables x, y, z. We will use metavariables M, N, L, e to range over expressions and D, d, dec to range over definitions. Metavariables might appear indexed or primed. The language of programs is the least set obtained inductively from the following grammar:

$$\begin{array}{rcll} M & ::= & \nabla x & \text{Variable} \\ & | & \lambda x \cdot M & \text{Abstraction} \\ & | & M \circ N & \text{Application} \\ & | & \mathit{letrec}\ D \cdot M & \text{Definition} \end{array}$$

The language of definitions is defined next. The two grammars are mutually dependent.

$$\begin{array}{rcll} D & ::= & \varepsilon & \text{Empty definition} \\ & | & x := M & \text{Single definition} \\ & | & D \parallel D' & \text{Parallel definitions} \end{array}$$

A parallel definition is malformed if and only if two definitions for the same variable are provided in parallel; only unambiguous definitions are well-formed.

5.2.2 Predicates and operations

Most of the formal definitions we will use are presented in this section. Elsewhere in the document, a fixpoint combinator is defined in §5.4.2 and another substitution operation is defined in §5.4.4. All these definitions will play a part in deriving the correctness proof for "extract a definition". Note that since the grammars of expressions and definitions are mutually dependent, many predicates and operations defined over programs are also defined in mutually-dependent pairs (i.e.

for expressions and definitions respectively). As a result of this, recursive definitions are given as mutually-recursive pairs of definitions, and proofs using those definitions are done by simultaneous induction.

Definition 5.2.1 Top-level declared *variables are* `letrec`*-bound variables of which binding instances occur in the outermost subterm or occur directly below other top-declared variables.*

```
DVTopd ε           = ∅
DVTopd x:=M        = {x}
DVTopd (d1 ‖ d2)   = DVTopd d1 ∪ DVTopd d2
DVTop ∇i           = ∅
DVTop λi·e         = ∅
DVTop (e1 ∘ e2)    = ∅
DVTop letrec d·e   = DVTopd d
```

As anticipated, note that this definition is split into two definitions $DVTopd$ and $DVTop$ given inductively on definitions and expressions respectively. We will use the naming convention of appending 'd' to the names of notions concerning definitions.

Definition 5.2.2 *Free variables*

```
FVd ε           = ∅
FVd x:=M        = FV M - {x}
FVd (d1 ‖ d2)   = FVd d1 ∪ FVd d2 - DVTopd (d1 ‖ d2)
FV ∇i           = {i}
FV λi·e         = FV e - {i}
FV (e1 ∘ e2)    = FV e1 ∪ FV e2
FV letrec d·e   = FV e ∪ FVd d - DVTopd d
```

In the last clause of the previous definition recall that set difference binds more weakly than union.

Definition 5.2.3 *λ-bound variables*

```
BVd ε           = ∅
BVd x:=M        = BV M
BVd (d1 ‖ d2)   = BVd d1 ∪ BVd d2
BV ∇i           = ∅
BV λi·e         = BV e ∪ {i}
BV (e1 ∘ e2)    = BV e1 ∪ BV e2
BV letrec d·e   = BV e ∪ BVd d
```

Definition 5.2.4 *letrec-bound variables*

```
DVd ε          = ∅
DVd x:=M       = {x} ∪ DV M
DVd (d1 ‖ d2)  = DVd d1 ∪ DVd d2
DV ∇i          = ∅
DV λi·e        = DV e
DV (e1 ∘ e2)   = DV e1 ∪ DV e2
DV letrec d·e  = DV e ∪ DVd d
```

Many of the definitions are given by primitive recursion as above. Some, such as the next definition, are defined non-recursively.

Definition 5.2.5 *Fresh variables:*

$$\texttt{Fresh}\ x\ M \stackrel{def}{=} x \notin \texttt{FV}\ M \cup \texttt{BV}\ M \cup \texttt{DV}\ M$$

Such non-recursive definitions are more abstract and are comparatively more difficult to use for mechanical proofs. The proof assistant employs a term rewriting engine (called the "simplifier") that can assist in unifying, pattern-matching and rewriting clauses from definitions given by primitive recursion. This is of great help in automating parts of proofs, but the nature of a definition might preclude such automatic mechanisms from being used and it is up to the human user then to arrive at the results manually. In fact we will not use *"Fresh"* since it is not amenable to this automation. Further down another predicate is defined by primitive recursion and proved to be logically equivalent to *"Fresh"*. We will use the refined predicate from that point onwards in the development for the automation benefit it brings.

The predicate for variable capture is defined next; it plays a central rôle in the rest of this chapter. We will discuss an improved definition of this predicate in the next chapter.

Definition 5.2.6 *Variable capture:*

$$\texttt{Captures}\ M\ N \stackrel{def}{=} \exists v{\in}\texttt{FV}\ N.\ v \in \texttt{BV}\ M \vee v \in \texttt{DV}\ M$$

The next definition is used when converting parallel definitions into a sequence of definitions – for example, $letrec\ (d_1 \parallel d_2) \cdot M$ becomes $letrec\ d_1 \cdot (letrec\ d_2 \cdot M)$. This transformation cannot be done if d_1 invokes a definition in d_2. The predicate defined next expresses dependency between definitions: a definition $D1$ depends on another definition $D2$ if the former invokes definitions found in the latter. This predicate is used to ensure that definitions given in parallel can indeed be nested in the manner described above; if the definitions are mutually recursive then they must be nested in one another in order to prevent *letrec*-bound variables from becoming free or from becoming bound by different binding occurrences.

Definition 5.2.7 *Dependent definitions:*

$$\texttt{Dep D1 D2} \stackrel{def}{=} \texttt{DVTopd D2} \cap \texttt{(FVd D1 - DVTopd D1)} \neq \emptyset$$

Next we define a substitution operation for this language. As explained in the Preliminaries chapter we will use a substitution operation that permits variable capture.

Definition 5.2.8 *Variable-capturing substitution*

```
ε[M/x]           = ε
y:=N[M/x]        = if x = y then y:=N else y:=N[M/x]
(d1 ‖ d2)[M/x]   = if x ∈ DVTopd (d1 ‖ d2) then d1 ‖ d2
                   else d1[M/x] ‖ d2[M/x]
∇i[M/x]          = if x = i then M else ∇i
λi·e[M/x]        = if x = i then λi·e else λi·e[M/x]
(e1 ∘ e2)[M/x]   = e1[M/x] ∘ e2[M/x]
letrec d·e[M/x]  = if x ∈ DVTop letrec d·e then letrec d·e
                   else letrec d[M/x]·e[M/x]
```

The operation defined next nests a definition within another definition. This is used in nesting mutually-recursive parallel definitions inside each other to preserve their meaning and to prepare the parallel definition to be changed into a sequence of definitions as explained earlier when introducing the *Dep* predicate.

Definition 5.2.9 *Local definition: inserting a definition within another definition's scope.*

```
ε ≪ d'          = ε
y:=N ≪ d'       = y:=letrec d'·N
d1 ‖ d2 ≪ d'    = (d1 ≪ d') ‖ (d2 ≪ d')
```

The next predicate is defined for recursive single definitions; this predicate is used when converting a *letrec* expression into an expression in the pure (*letrec*-free) λ-calculus. The rules for performing this conversion will be given in the logic defined further down.

Definition 5.2.10 *Recursive definitions: a definition is considered to be recursive if the defined variable occurs free in the defined expression.*

```
Rec x:=M = x ∈ FV M
```

Definition 5.2.11 *Definitions are well-formed if and only if parallel definitions define different variables. Expressions are well-formed if and only if all definitions they contain are well-formed.*

```
WFPredd ε           = True
WFPredd y:=N        = WFPred N
WFPredd (d1 ‖ d2)   = WFPredd d1 ∧ WFPredd d2 ∧ DVd d1 ∩ DVd d2 = ∅
WFPred ∇i           = True
WFPred λi·e         = WFPred e
WFPred (e1 ∘ e2)    = WFPred e1 ∧ WFPred e2
WFPred letrec d·e   = WFPredd d ∧ WFPred e
```

Definition 5.2.12 *Subexpressions and subdefinitions*

$$
\begin{array}{lll}
M \subseteq_\Lambda \ \varepsilon & = & \mathit{False} \\
M \subseteq_\Lambda \ x{:=}N & = & M = N \vee M \subseteq_\Lambda N \\
M \subseteq_\Lambda \ d1 \parallel d2 & = & M \subseteq_\Lambda \ d1 \vee M \subseteq_\Lambda \ d2 \\
M \subseteq_\Lambda \nabla x & = & M = \nabla x \\
M \subseteq_\Lambda \lambda x{\cdot}N & = & M = \lambda x{\cdot}N \vee M \subseteq_\Lambda N \\
M \subseteq_\Lambda N \circ N' & = & M = N \circ N' \vee M \subseteq_\Lambda N \vee M \subseteq_\Lambda N' \\
M \subseteq_\Lambda \mathit{letrec}\ d{\cdot}N & = & M = \mathit{letrec}\ d{\cdot}N \vee M \subseteq_\Lambda N \vee M \subseteq_\Lambda \ d
\end{array}
$$

We now give another definition for a variable-freshness predicate. Unlike the previous definition, its characteristic function will be defined recursively: the new definition is less abstract. We prove the two definitions equivalent on page 44.

Definition 5.2.13 *Fresh variable predicate*

$$
\begin{array}{lll}
n \sharp \varepsilon & = & \mathit{True} \\
n \sharp x{:=}N & = & x \neq n \wedge n \sharp N \\
n \sharp d1 \parallel d2 & = & n \sharp d1 \wedge n \sharp d2 \\
n \sharp \nabla x & = & n \neq x \\
n \sharp \lambda x{\cdot}N & = & n \neq x \wedge n \sharp N \\
n \sharp N \circ N' & = & n \sharp N \wedge n \sharp N' \\
n \sharp \mathit{letrec}\ d{\cdot}N & = & n \sharp d \wedge n \sharp N
\end{array}
$$

5.2.3 Logic

Equational logic is a fragment of FOL in which judgements are equations. We embed this logic in the proof assistant's logic (HOL) and use it to reason in the theory of program equations. A refactoring is behaviour-preserving iff the original and refactored programs stand in the relation $\simeq$ induced by the following rules.

λ-rules

$$\frac{z \sharp M}{\lambda x{\cdot}M \simeq \lambda z{\cdot}M[\nabla z/x]}\ \text{ALPHA}$$

$$\frac{\neg\ \mathit{Captures}\ M\ N}{\lambda x{\cdot}M \circ N \simeq M[N/x]}\ \text{BETA} \qquad \frac{x \notin \mathit{FV}\ M}{\lambda x{\cdot}M \circ \nabla x \simeq M}\ \text{ETA}$$

Equivalence rules

$$M \simeq M\ \text{REFL} \qquad \frac{M \simeq N}{N \simeq M}\ \text{SYMM} \qquad \frac{M \simeq M' \qquad M' \simeq N}{M \simeq N}\ \text{TRAN}$$

Compatibility rules

$$\frac{M \simeq M' \qquad N \simeq N'}{M \circ N \simeq M' \circ N'}\ \text{compApp} \qquad \frac{M \simeq N}{\lambda x{\cdot}M \simeq \lambda x{\cdot}N}\ \text{compAbs}$$

letrec-rules

$$\texttt{letrec}\ \varepsilon{\cdot}M \simeq M\ \text{GARBAGE}$$

$$\texttt{letrec}\ d1 \parallel d2{\cdot}M \simeq \texttt{letrec}\ d2 \parallel d1{\cdot}M\ \text{EXCHANGE}$$

$$\frac{\neg\ \texttt{Rec}\ x{:=}N}{\texttt{letrec}\ x{:=}N{\cdot}M \simeq \lambda x{\cdot}M \circ N}\ \text{cNRec}$$

$$\frac{\texttt{Rec}\ x{:=}N}{\texttt{letrec}\ x{:=}N{\cdot}M \simeq \lambda x{\cdot}M \circ (\texttt{fix} \circ \lambda x{\cdot}N)}\ \text{cRec}$$

$$\frac{\neg\ \texttt{Dep}\ d1\ d2}{\texttt{letrec}\ d1 \parallel d2{\cdot}M \simeq \texttt{letrec}\ d1{\cdot}\texttt{letrec}\ d2{\cdot}M}\ \text{NEST}$$

$$\frac{\texttt{Dep}\ d1\ d2 \qquad \texttt{Dep}\ d2\ d1}{\texttt{letrec}\ d1 \parallel d2{\cdot}M \simeq \texttt{letrec}\ d1 \ll d2{\cdot}\texttt{letrec}\ d2 \ll d1{\cdot}M}\ \text{nestRec}$$

Compatability rules are only provided for the lambda-fragment of the language; they are not provided for letrec terms since a letrec term is meaningful only if it is convertible to a lambda term (this extension of the λ-calculus is consistent).

5.3 Some lemmata

The results discussed next support the lemmata proved later, which in turn support correctness proofs of refactorings.

5.3.1 Equivalence between freshness definitions

Some definitions are easy to read while others lend themselves better to automation by the proof assistant. The definition for `Fresh z M` is more compact and clear, and the definition for `z ♯ M` is more detailed and usable by the prover's automatic procedures. However, the two definitions are logically equivalent and therefore interchangeable.

In order to prove this logical equivalence we need to define freshness over definitions (in `Fresh z M` the metavariable M ranges over expressions and not over definitions). Since the grammars of expressions and definitions are mutually-dependent, formulæ about properties of the language must be stated conjoined together.

Definition 5.3.1 *Variable freshness defined over definitions:*
$\mathit{Freshd}\ x\ e \stackrel{def}{=} x \notin \mathit{FVd}\ e \cup \mathit{BVd}\ e \cup \mathit{DVd}\ e$

We prove the two definitions equivalent next. Note that '=' is HOL's bi-implication connective.

Lemma 5.3.2 (equivFreshDefns)

$$\forall M\ z.\ (z \sharp M) = \mathit{Fresh}\ z\ M$$

Proof sketch The left-to-right direction is proved by simultaneous structural induction on M and D in $(z \sharp M \longrightarrow \mathit{Fresh}\ z\ M) \wedge (z \sharp D \longrightarrow \mathit{Freshd}\ z\ D)$ and projecting the first conjunct. The opposite direction uses the same method but performed on $(\mathit{Fresh}\ z\ M \longrightarrow z \sharp M) \wedge (\mathit{Freshd}\ z\ D \longrightarrow z \sharp D)$

∎

Refining the definition of the original predicate to take advantage of automatic procedures will facilitate proofs of theorems relying on its definition. Adapting definitions is not uncommon in mechanical reasoning: for instance if a definition is given recursively on the first argument but a proof requires induction on a different argument, an equivalent definition defined inductively on the second argument is sought.

5.3.2 Other basic properties

Various proofs of peripheral yet crucial properties are given throughout the formal development to support the larger theorems. These lemmata include routine results such as Lemma 5.3.3. It asserts that renaming a non-recursive definition will not make it recursive.

Lemma 5.3.3 (rename_orth_rec)

$$z \sharp \mathit{letrec}\ x{:=}N{\cdot}M \wedge \neg\ \mathit{Rec}\ x{:=}N \Longrightarrow \neg\ \mathit{Rec}\ z{:=}N[\nabla z/x]$$

Proof sketch By induction on the structure of N. The proof also uses the formulæ below (proved by simultaneous induction on expressions and declarations and projecting the formula concerning expressions):

- $\forall z\ N.\ z \sharp N \longrightarrow z \notin \mathit{FV}\ N$
- $\forall x\ z\ N.\ z \sharp N \wedge x \notin \mathit{FV}\ N \wedge x \neq z \longrightarrow z \notin \mathit{FV}\ (N[\nabla z/x])$

∎

Note that despite the absence of HOL quantifiers the formula of lemma *(rename_orth_rec)* is universal since the free variables are implicitly universally quantified.

Earlier on we mentioned that occasionally meta-parts of the logic will show through: note the $\Longrightarrow$ in the previous formula. It denotes Isabelle's logical implication (i.e. at the logical framework level); HOL's material implication is denoted by $\longrightarrow$. The formula was originally entirely in HOL, but was converted to this meta-form to enable this theorem to be used directly as a rule. This will help explain the occurrences of the different implication symbols in the scripts. Theorems will occasionally be displayed in rule form to improve readability (e.g. in §5.4.1).

5.4 Refactorings

All the refactorings verified below involve definitions in some way. Most of the correctness proofs for refactorings will follow the same pattern: we first prove the case for a non-recursive definition, then prove the recursive case, and combine the two in the main result. The proofs for non-recursive and recursive cases are symmetric, in the sense that they differ only in the `letrec`-to-λ interpretation rule (i.e. using rule cNRec rather than cRec or vice versa) and the other tactics are mostly the same (except for instantiations of the intermediate terms in tran since these reflect the different λ-expressions that recursive and non-recursive *letrec* expressions are convertible, to using the logic defined earlier).

Perhaps the most interesting aspect of the behaviour-preservation theorems are refactorings' side-conditions. Side-conditions constrain which programs are transformed by the refactoring. They must be sufficient for behaviour-preservation, but if they are too strong the usefulness of the refactoring is diminished: it would mean that there are many programs which *could* be admitted for behaviour-preserving transformation but which are not.

5.4.1 Rename a definition

This is one of the simplest refactorings. While it does not contribute to verifying the "extract a definition" refactoring, its simplicity lets us focus on the verification rather than the refactoring itself.

Formulæ will occasionally be displayed in rule form to improve their readability. The non-recursive case of this refactoring is formulated as the following lemma:

Lemma 5.4.1 (rename_nonRec)

$$\frac{z \sharp \texttt{letrec } x{:=}N \cdot M \wedge \neg\ \texttt{Rec } x{:=}N}{\texttt{letrec } x{:=}N \cdot M \simeq \texttt{letrec } z{:=}N[\nabla z/x] \cdot M[\nabla z/x]}$$

Proof sketch Apply cNRec to either side and relate the resulting using TRAN. Apply compApp to either side yielding two subgoals:

- $\lambda x \cdot M \simeq \lambda z \cdot (M[\lambda z/x])$ is proved using ALPHA
- $N[\nabla z/x] = N$ is proved since $x \notin FVN$ (derived from unfolding the assumption $\neg$ `Rec x:=N`) and thus $N \simeq N$ by REFL

∎

Lemma 5.4.2 (rename_Rec)

$$\frac{\texttt{z} \sharp \texttt{letrec x:=N·M} \wedge \texttt{Rec x:=N}}{\texttt{letrec x:=N·M} \simeq \texttt{letrec z:=N[}\nabla\texttt{z/x]·M[}\nabla\texttt{z/x]}}$$

Proof sketch Similar to previous proof, except that cRec is used instead of cNRec, and the second subgoal is proved by applying compApp again and using REFL to close the branch $fix \simeq fix$, and ALPHA for closing the second branch $\lambda x \cdot N \simeq \nabla z \cdot (N[\lambda z/x])$.

∎

The two lemmas are now combined to prove the overall theorem; it asserts that irrespective of whether a definition is recursive, it can be renamed without the program's meaning being changed if the new name is fresh. This is proved for arbitrary variables and programs, as is clear from the formal definition as a HOL formula:

Theorem 5.4.3 (rename_a_definition)

$\forall$ `z x N M. z` $\sharp$ `letrec x:=N·M` $\longrightarrow$ `letrec x:=N·M` $\simeq$ `letrec z:=N[`∇`z/x]·M[`∇`z/x]`

Proof sketch By case analysis on `Rec x:=N`; either case is immediate from Lemmas 5.4.1 and 5.4.2 respectively.

∎

5.4.2 Add/drop a redundant definition

As refactorings become more complex the details of definitions made earlier (§5.2.2) become more important since the complexity of the refactoring realises the subtle implications of a definition. The "rename a definition" refactoring is a very basic refactoring and consequently does not really "stress-test" our definitions; it is a very easy refactoring to verify.

The refactoring we seek to prove at the end of this chapter, i.e. "extract a definition", is very different in this regard since it is rather complex. Consequently it will test our definitions better. From this point on we will be working

towards verifying that refactoring, starting by verifying its elementary refactorings. "Add/drop a redundant definition" is a component refactoring of "extract a definition".

In the logic given earlier we have used the constant *fix* in the definition of the rule cRec, without having defined *fix*. We now define this constant in the programming language.

The constant *fix* needs to be a closed expression in our language. We consider two definitions, compare them and choose one.

Definition 5.4.4 *Concrete fixpoint combinator:*
`fix` $\stackrel{def}{=}$ `λ0·λ1·∇0 ∘ (∇1 ∘ ∇1) ∘ λ1·∇0 ∘ (∇1 ∘ ∇1)`

Definition 5.4.5 *Abstract fixpoint combinator:*
`fix' f x` $\stackrel{def}{=}$ `λf·λx·∇f ∘ (∇x ∘ ∇x) ∘ λx·∇f ∘ (∇x ∘ ∇x)`

The first definition fixes the names of variables the fixpoint combinator uses. The second definition could only be used under the assumption that `f` $\neq$ `x`, and using either definition requires that ¬ `Captures fix M` to be able to β-reduce `fix ∘ M`. We will pick the first definition since it requires less side-conditions, but the nature of this solution indicates that *fix* had better be defined primitively rather than as a constant. This involves including *fix* in the language's definition and adding a rule in the logic to unwind *fix* expressions: $fix \circ M \simeq M \circ (fix \circ M)$. We will take this approach in the language formalised in the next chapter, but use this constant definition as a compromise in the current language.

Theorem 5.4.6 (`add_drop_a_redundant_definition'`)

```
∀ h N L.
   h ∉ FV L ∧ ¬ Captures L (fix ∘ λh·N) ∧ ¬ Captures L N ⟶
   L ≃ letrec h:=N·L
```

Proof sketch By case analysis on `Rec x:=N` and using cNRec and cRec as before to convert the expressions in each case into pure λ-expressions. The proof then relies on proving the following lemma (by simultaneous structural induction on M and D):

```
(x ∉ FV M ⟶ M[N/x] = M) ∧ (x ∉ FVd D ⟶ D[N/x] = D)
```

∎

Refining side-conditions

We now discuss the side-conditions of this refactoring and how they can be minimised. The first side-condition (`h ∉ FV L`) ensures that the chosen h does not capture any variables in L; this seems reasonable. Replacing it with `h ♯ L` would have been a stronger alternative since this proposition would require h not to appear at all in L.

The second and third side-conditions ensure that the terms related by $\simeq$ are β-convertible (i.e. they satisfy the side-conditions of rule BETA). The third side-condition, `¬ Captures L N`, cannot be made smaller but the second side-condition, `¬ Captures L (fix ∘ λh·N)` can be decomposed further. We could split it into `¬ Captures L fix` and `¬ Captures L λh·N`. We could refine `¬ Captures L λh·N` further into `¬ Captures L N` (and by idempotence of conjunction we drop this assumption since it already appears in the side-conditions). To do this we could use the following lemma:

Theorem 5.4.7 (*nocap_subterm_fix*)

`¬ Captures L N ∧ FV (fix ∘ λh·N) ⊆ FV N ⟹ ¬ Captures L (fix ∘ λh·N)`

Proof sketch `FV (fix ∘ λh·N) ⊆ FV N` is proved by induction on the structure of N. The rest of the formula is proved by unfolding the definition of *Captures* and reasoning set-theoretically.

∎

The improved specification of the refactoring is given next.

Theorem 5.4.8 (*add_drop_a_redundant_definition''*)

`h ∉ FV L ∧ ¬ Captures L N ⟶ L ≃ letrec h:=N·L`

Proof sketch The proof uses the same approach as for Theorem 5.4.6 and uses Lemma 5.4.7.

∎

Reducing side-conditions avoids wasteful computation when the refactoring is applied to a program since less computation is needed to fulfil sufficiency checks.

We will use observations such as these to improve our work, especially in the the formalisation of the second language. These observations will be summarised at the end of the chapter.

5.4.3 Demote a definition

As done previously, we prove the non-recursive and recursive cases (wrt `h`) as separate lemmata and combine them in the main theorem.

We now define a predicate over definitions to detect variable capture; the one we have so far works only for expressions. As previously mentioned, definitions occur in such pairs due to the mutually recursive syntactic categories. We will be requiring this definition for further proofs, such as the lemma shown below.

Definition 5.4.9 *Variable-capture predicate for definitions:*
`Capturesd d n` $\stackrel{\text{def}}{=}$ $\exists$ `v∈FV n. v ∈ BVd d ∨ v ∈ DVd d`

Lemma 5.4.10 (CapSubsumesDVT)

$$(\forall N\ f.\ \neg\ \mathit{Captures}\ N\ \nabla f \longrightarrow f \notin \mathit{DVTop}\ N) \wedge$$
$$(\forall D\ f.\ \neg\ \mathit{Capturesd}\ D\ \nabla f \longrightarrow f \notin \mathit{DVTopd}\ D)$$

Proof sketch We first prove the lemma *(DVTthenDV)*, formulated below, by simultaneous induction on the structure of M and D:

$$(x \in \mathit{DVTop}\ M \longrightarrow x \in \mathit{DV}\ M) \wedge (x \in \mathit{DVTopd}\ D \longrightarrow x \in \mathit{DVd}\ D)$$

The proof proceeds by unfolding *Captures* and *Capturesd*, using the lemma *(DVTthenDV)* and reasoning set-theoretically.

∎

Various routine lemmata are needed in preparation to support the refactoring's verification. For example, the following lemma establishes that variables do not become free as a result of substitution.

Lemma 5.4.11 (`free_inheritance2`)

$$(h \neq f \wedge f \notin \mathit{FV}\ M \wedge f \notin \mathit{FV}\ N \longrightarrow f \notin \mathit{FV}\ (M[N/h])) \wedge$$
$$(h \neq f \wedge f \notin \mathit{FVd}\ D \wedge f \notin \mathit{FV}\ N \longrightarrow f \notin \mathit{FVd}\ (D[N/h]))$$

Proof sketch Simultaneous induction on the structure of M and D.

∎

Some of the lemmata serve to lessen the side-conditions by exploiting dependencies between them. Recall that this was also done for the previous refactoring to avoid wasteful computation when applying the refactoring. For example, the next lemma uses properties of small terms to prove properties of larger terms:

Lemma 5.4.12 (`noCapDefs`)

$$(\neg\ \mathit{Captures}\ L\ N \wedge \neg\ \mathit{Captures}\ L\ M \longrightarrow \neg\ \mathit{Captures}\ L\ \mathit{letrec}\ h{:=}N{\cdot}M) \wedge$$
$$(\neg\ \mathit{Capturesd}\ D\ N \wedge \neg\ \mathit{Capturesd}\ D\ M \longrightarrow \neg\ \mathit{Capturesd}\ D\ \mathit{letrec}\ h{:=}N{\cdot}M)$$

Proof sketch Simultaneous induction on L and D, unfolding *Captures* and *Capturesd*, and set-theoretical reasoning.

∎

Because of the sequence of β-reductions, propositions like the one shown below started appearing as the side-conditions of this refactoring. That is, the β-reduction can only be performed if this side-condition is satisfied.

$$\neg\ \mathit{Captures}\ (L[N/h])\ M$$

Notice that this is a property of the *transformed* program and if the property is not satisfied then the effort spent pre-checking and transforming the program would have been wasted. In order to avoid wasted computation the side-conditions need to be concerned solely with the original program. This requires further lemmata to keep the side-conditions pertinent to the original term. This "adaptation" of side-conditions is performed by the following lemmata :

Lemma 5.4.13

(i) (*noCapPreservation*) *:*

$$(\neg\ \mathit{Captures}\ L\ M \wedge \neg\ \mathit{Captures}\ N\ M \longrightarrow \neg\ \mathit{Captures}\ (L[N/h])\ M) \wedge$$
$$(\neg\ \mathit{Capturesd}\ D\ M \wedge \neg\ \mathit{Captures}\ N\ M \longrightarrow \neg\ \mathit{Capturesd}\ (D[N/h])\ M)$$

(ii) (*noCapPreservation'*)*:*

$$(\neg\ \mathit{Captures}\ L\ N \wedge \neg\ \mathit{Captures}\ L\ M \longrightarrow \neg\ \mathit{Captures}\ L\ (M[N/h])) \wedge$$
$$(\neg\ \mathit{Capturesd}\ D\ N \wedge \neg\ \mathit{Capturesd}\ D\ M \longrightarrow \neg\ \mathit{Capturesd}\ D\ (M[N/h]))$$

Proof sketch Simultaneous induction on L and D, unfolding *Captures* and *Capturesd*, and set-theoretical reasoning.

∎

Specifying the side-conditions in terms of the original, rather than the transformed, program is another recommendation we make at the end of the chapter to implement computationally economical, apart from correct, refactorings. The previous lemmata contribute to proving the following lemma:

Lemma 5.4.14 (*noCapPresLemma*)

$$\frac{\neg\ \mathit{Captures}\ L\ N \wedge \neg\ \mathit{Captures}\ N\ N \wedge \neg\ \mathit{Captures}\ L\ M \wedge \neg\ \mathit{Captures}\ N\ M}{\neg\ \mathit{Captures}\ (L[N/h])\ (\mathtt{letrec}\ h{:=}N{\cdot}M[N/h])}$$

Proof sketch Rather than proving this from first principles we have opted to prove it by combining smaller lemmas to facilitate the proof. Constituent lemmas include *(noCapPreservation)* and *(noCapPreservation')* described above.

∎

The non-recursive case of this refactoring is proved next:

Lemma 5.4.15 (*lift_or_demote_NRec*)

$\neg$ *Captures letrec f:=letrec h:=N·M·L N* $\wedge$
h $\neq$ *f* $\wedge$
$\neg$ *Captures L M* $\wedge$
$\neg$ *Captures N M* $\wedge$ $\neg$ *Rec h:=N* $\wedge$ $\neg$ *Captures L* ∇*h* $\wedge$ $\neg$ *Captures N* ∇*f* $\Longrightarrow$
letrec h:=N·letrec f:=M·L $\simeq$ *letrec h:=N·letrec f:=letrec h:=N·M·L*

Proof sketch As in previous refactorings, both sides of the equivalence relation are transformed using rule BETA and kept in relation using TRAN. Rules COMPAPP and COMPABS are used to decompose both expressions in relation simultaneously, and lemmata such as *(noCapPreservation)*, described earlier, are used to satisfy the side-condition of BETA and avoid having to prove results about the *Captures* predicate from first principles.

∎

The side-conditions are explained in more detail next:

- $\neg$ *Rec h:=N* is the case considered by this lemma.
- *h* $\neq$ *f* since if *h = f* then the conclusion of the theorem becomes *letrec f:=N·letrec f:=M·L* $\simeq$ *letrec f:=N·letrec f:=letrec f:=N·M·L*. This is not valid if *f* is recursive (since occurrences of *f* in *M* would be substituted with *N* instead of *M*, and this is incorrect unless we break the assumption of arbitrarity of *N* and *M*). We could restrict the refactoring to transform non-recursive definitions, and add that as a side-condition, but we will explore the more general alternative here.
- $\neg$ *Captures (letrec f:=letrec h:=N·M·L) N* (this is required for the RHS *letrec h:=N·letrec f:=letrec h:=N·M·L* to β-reduce). This side-condition also implies $\neg$ *Captures letrec f:=M·L N* (which serves the same purpose, except that this time it is for the LHS), and that N, M and L do not capture N. Since the *letrec* unfolds to a term which must β-reduce, capture is forbidden by the side-condition of the rule BETA; this side-condition is inherited as a side-condition of the refactoring, as in previous refactorings.
- The side-conditions $\neg$ *Captures L M* $\wedge$ $\neg$ *Captures N M* together with the assumption $\neg$ *Captures L N* $\wedge$ $\neg$ *Captures N N* (derived from the previous side-condition through lemma *simplerCap*)[1] are needed to imply $\neg$ *Captures (L[N/h]) (letrec h:=N·M[N/h])* $\wedge$ $\neg$ *Captures (L[N/h]) M* (using lemma *noCapPresLemma*), to enable us to specify the refactoring only in terms of the program to be transformed, rather than the transformed program.

[1]This lemma states that: $\neg$ *Captures letrec f:=letrec h:=N·M·L N* $\Longrightarrow$ $\neg$ *Captures L N* $\wedge$ $\neg$ *Captures N N*

- ¬ `Captures L` ∇`h` and ¬ `Captures N` ∇`f` are needed to preserve (non)recursion.

The specification of the refactoring's recursive case is similar (differing only in the complement assumption `Rec h:=N`), and includes the side-condition ¬ `Captures fix` ∇`f`. The proof follows the same approach used in the non-recursive case.

Lemma 5.4.16 (*lift_or_demote_Rec*)

$\neg$ *Captures letrec f:=letrec h:=N·M·L N* $\wedge$
h $\neq$ *f* $\wedge$
$\neg$ *Captures L M* $\wedge$
$\neg$ *Captures N M* $\wedge$
Rec h:=N $\wedge$ $\neg$ *Captures L* ∇*h* $\wedge$ $\neg$ *Captures N* ∇*f* $\wedge$ $\neg$ *Captures fix* ∇*f* $\Longrightarrow$
letrec h:=N·letrec f:=M·L $\simeq$ *letrec h:=N·letrec f:=letrec h:=N·M·L*

As in previous refactorings, the two cases are combined to verify the refactoring.

Theorem 5.4.17 (*lift_or_demote'*)

$\neg$ *Captures letrec f:=letrec h:=N·M·L N* $\wedge$
h $\neq$ *f* $\wedge$
$\neg$ *Captures L M* $\wedge$
$\neg$ *Captures N M* $\wedge$ $\neg$ *Captures L* ∇*h* $\wedge$ $\neg$ *Captures N* ∇*f* $\wedge$ $\neg$ *Captures fix* ∇*f*
$\Longrightarrow$
letrec h:=N·letrec f:=M·L $\simeq$ *letrec h:=N·letrec f:=letrec h:=N·M·L*

Proof sketch By case analysis on *Rec h:=N*; each case is immediate from *lift_or_demote_NRec* and *lift_or_demote_Rec* respectively.

∎

5.4.4 Declare/Inline a local definition

We now define a different substitution operation. The substitution operation defined earlier substitutes variables for terms, the new one does the converse. We could have used the old operation in this refactoring, but it will be interesting to see both types of substitution used to explore how different program transformations interact without changing program behaviour.

Definition 5.4.18 *Subterm-for-variable substitution*

ε*[x:M]*	=	ε
y:=N[x:M]	=	*if x = y then y:=N else y:=N[x:M]*
(d1 ‖ d2)[x:M]	=	*if x* $\in$ *DVTopd (d1 ‖ d2) then d1 ‖ d2*
		else d1[x:M] ‖ d2[x:M]
∇*i[x:M]*	=	*if M =* ∇*i then* ∇*x else* ∇*i*
λ*i·e[x:M]*	=	*if M =* λ*i·e then* ∇*x else if x = i then* λ*i·e else* λ*i·e[x:M]*
(e1 $\circ$ *e2)[x:M]*	=	*if M = e1* $\circ$ *e2 then* ∇*x else e1[x:M]* $\circ$ *e2[x:M]*
letrec d·e[x:M]	=	*if M = letrec d·e then* ∇*x else*
		if x $\in$ *DVTop letrec d·e then letrec d·e*
		else letrec d[x:M]·e[x:M]

As with the substitution operation defined in §5.2.2, we will need lemmata, such as Lemma 5.4.13, that shift the focus of properties of a transformed program to the original version. For the new substitution operation we prove the following lemma.

Lemma 5.4.19 (*capImp*)

(¬ Captures M N ⟶ ¬ Captures (M[g:N]) N) ∧
(¬ Capturesd D N ⟶ ¬ Capturesd (D[g:N]) N)

Proof sketch Simultaneous structural induction on M and D, unfolding *Captures* and *Capturesd* and reasoning set-theoretically.

∎

The lemma proving the non-recursive case of this refactoring is proved next.

Lemma 5.4.20 (*declare_or_inline_NRec*)

¬ Rec f:=M ∧
¬ Rec g:=N ∧
g ≠ f ∧ g ♯ M ∧ N ⊆Λ M ∧ ¬ Captures M N ∧ f ∉ FV N ∧ f ∉ DVTop N ⟹
letrec f:=M·L ≃ letrec f:=letrec g:=N·M[g:N]·L

Proof sketch As for Lemma 5.4.15.

∎

The side-conditions for this case are the following:

- *¬ Rec f:=M* indicates the specific case being proved.
- *¬ Rec g:=N* is assumed since we inline definition *g:=N* simply by replacing occurrences of g with N. This naïve substitution does not preserve recursion – note that with the definition *g:=N* removed after recursion, free occurrences of g in N will not have the same definition to recur on, thus changing the meaning of the expression.
- *g ≠ f* since *g = f* would make the transformed program recursive, since *f ∈ FV (M[g:N])*.
- *g ♯ M* since we assume g to be a fresh name.
- *N ⊆Λ M* expresses that we expect the newly-declared definition to be extracted from the main expression.
- *¬ Captures M N* is needed since it implies *¬ Captures (M[g:N]) N*, which is required for a β-reduction to go through.
- *f ∉ FV N* since we require that the definition not be recursive; if *f ∈ FV N* then definition *f* would be recursive, contradicting the principal assumption of this case since N is a subexpression of M. *f ∈ FV N* might come about if *N* is extracted from below a binding of *f*, thus freeing its occurrences in *N*.

- $f \notin DVTop\ N$ is also related to preserving non-recursion: we require that $\neg\ Rec\ f{:=}letrec\ g{:=}N{\cdot}M[g{:=}N]$. Therefore we require that $f \notin FV\ N$ and $f \notin FV\ (M[g{:=}N])$. The former appears as the previous side-condition. In the case that $f \notin DVTop\ M$ then we require that $f \notin DVTop\ (M[g{:=}N])$ since this would guarantee that $f \notin FV\ (M[g{:=}N])$. This is proved by assuming that $f \notin DVTop\ N$, therefore this assumption is inherited by the refactorings since it could not be discharged.

The second side-condition could be removed if the inlining operation handled recursive definitions differently by changing them into fixpoint expressions. However, *fix* is not part of the core language and generating a *fix* expression would require generating names. This might make the refactored program harder to read, so we avoid this approach in spite of the generality it provides.

The side-conditions of the recursive case's lemma are mostly the same, the only exception is that $Rec\ f{:=}M$ replaces $\neg\ Rec\ f{:=}M$. The side-conditions used to preserve meaning in the non-recursive case ($g \neq f$, $f \notin FV\ N$, and $f \notin DVTop\ N$) act to preserve the meaning in the recursive case since if they are not satisfied then the recursion would be done on a different expression rather than the one intended. The lemma for the recursive case is formulated next.

Lemma 5.4.21 (*declare_or_inline_Rec*)

$$Rec\ f{:=}M \wedge \neg\ Rec\ g{:=}N \wedge g \neq f \wedge g \sharp M \wedge N \subseteq_\Lambda M \wedge \neg\ Captures\ M\ N \wedge f \notin FV\ N \wedge f \notin DVTop\ N \Longrightarrow letrec\ f{:=}M{\cdot}L \simeq letrec\ f{:=}letrec\ g{:=}N{\cdot}M[g{:=}N]{\cdot}L$$

These two cases are combined to prove the main theorem, but the side-conditions will be changed slightly:

- The side-conditions $g \neq f \wedge g \sharp M$ are abbreviated using $g \sharp f{:=}M$
- $\neg\ Captures\ M\ N \wedge f \notin FV\ N$ are abbreviated by $\neg\ Capturesd\ f{:=}M\ N$ (this abbreviation is proved sound using Lemma 5.4.19)

The refactoring's correctness is formulated next. Its proof combines the non-recursive and recursive cases described previously.

Theorem 5.4.22 (*declare_or_inline*)

$$\neg\ Rec\ g{:=}N \wedge g \sharp f{:=}M \wedge N \subseteq_\Lambda M \wedge \neg\ Capturesd\ f{:=}M\ N \wedge f \notin DVTop\ N \Longrightarrow letrec\ f{:=}M{\cdot}L \simeq letrec\ f{:=}letrec\ g{:=}N{\cdot}M[g{:=}N]{\cdot}L$$

5.4.5 Extract/Inline a definition

This refactoring is composed of the last three refactorings that have been verified. It is proved by transitivity steps through these three refactorings. It involves the following steps:

1. `letrec f:=M·L` is the original expression, and is changed to
2. `letrec f:=letrec g:=N·M[g:N]·L` by `declare_or_inline`, and
3. `letrec g:=N·letrec f:=letrec g:=N·M[g:N]·L` using `add_drop_a_redundant_definition`, and finally to
4. `letrec g:=N·letrec f:=M[g:N]·L` by using `lift_or_demote'`.

As any compound refactoring, this refactoring inherits the side-conditions of its constituent refactorings. It is not always obvious which refactorings the side-conditions originate from since, as we saw earlier, the side-conditions might need to be adapted. Moreover, further adaptation of the side-conditions may be necessary in order to "interface" between the constituent refactorings – i.e. proving that the output of a refactoring in a composite always satisfies a precondition of the successive refactoring. Roberts calls these "postconditions" and recall that they serve to lessen the number of potentially wasteful checks made on programs after they have been transformed. This was described in §2.2.2.

For example, applying "lift or demote" required the satisfaction of these preconditions:

$$\neg\ \mathit{Captures\ letrec\ f{:=}letrec\ g{:=}N{\cdot}M[g{:}N]{\cdot}L\ N}\ \wedge$$
$$g \neq f\ \wedge$$
$$\neg\ \mathit{Captures\ L\ (M[g{:}N])}\ \wedge$$
$$\neg\ \mathit{Captures\ N\ (M[g{:}N])}\ \wedge$$
$$\neg\ \mathit{Captures\ L}\ \nabla g\ \wedge\ \neg\ \mathit{Captures\ N}\ \nabla f\ \wedge\ \neg\ \mathit{Captures\ fix}\ \nabla f$$

Note that the first, third and fourth conjuncts are propositions concerning a (intermediate) transformed program. To avoid potentially wasteful computations we proved the implication of the first conjunct from a property of the original program:

Lemma 5.4.23 (*postProperty*)

$$\frac{\neg\ \mathit{Captures\ letrec\ f{:=}letrec\ g{:=}N{\cdot}M{\cdot}L\ N}}{\neg\ \mathit{Captures\ letrec\ f{:=}letrec\ g{:=}N{\cdot}M[g{:}N]{\cdot}L\ N}}$$

This proof for the "extract_a_definition" refactoring is not split into two cases, but is proved by repetitively invoking the transitivity rule and using the proofs of its constituent refactorings.

Theorem 5.4.24 (*extract_a_definition*)

$$
\begin{array}{l}
g \notin FV\ L\ \wedge \\
\neg\ Rec\ g{:=}N\ \wedge \\
g \sharp f{:=}M\ \wedge \\
N \subseteq_{\Lambda} M\ \wedge \\
\neg\ Captures\ fix\ \nabla f\ \wedge \\
\neg\ Captures\ L\ \nabla g\ \wedge \\
\neg\ Captures\ N\ \nabla f\ \wedge \\
\neg\ Captures\ L\ M\ \wedge \\
\neg\ Captures\ N\ M\ \wedge \\
\neg\ Captures\ letrec\ f{:=}letrec\ g{:=}N{\cdot}M{\cdot}L\ N\ \wedge \\
\neg\ Captures\ L\ (M[g{:=}N])\ \wedge\ \neg\ Captures\ N\ (M[g{:=}N])\ \longrightarrow \\
letrec\ f{:=}M{\cdot}L \simeq letrec\ g{:=}N{\cdot}letrec\ f{:=}M[g{:=}N]{\cdot}L
\end{array}
$$

5.5 Conclusion

This exercise provided a glimpse at some of the issues which one immediately encounters when verifying refactorings mechanically in this manner. We have verified a number of refactorings, ranging from trivial and elementary to non-trivial and compound. A refactoring can have multiple formulations (as pointed out by Li in her thesis in §2.8 when discussing the "design space" of refactoring formulations), but we have attempted to provide sufficient motivation to justify the formulation of refactorings verified here.
Possible improvements to this work include:

Object language Clearly the language we have used here is extremely simplistic and far removed from a usable programming language. It would be interesting to apply this approach for a more sophisticated language. Exploring the tractability of using this approach on a non-applicative language would be an interesting exercise too.

Weaker assumptions Side-conditions are a very interesting part of the correctness formulæ since they guide the implementation of *correct* (behaviour-preserving) refactorings. Unless one is careful it is possible to end up with assumptions stronger than needed. This is elaborated further in the next chapter.

This verification effort has served to empirically gather a collection of observations that can help improve our next attempt. We have emphasised the need to specify refactorings in such a way that they are not only behaviour-preserving but also computationally economical. This is discussed further in the next chapter.

Chapter 6

"Enlarge definition type"

6.1 Introduction

We now consider a larger system resembling PCF (Plotkin 1977) extended with unit and sum types. The language has the following types: natural numbers (as the only base types), function space, sums and unit. The typing rules, operation definitions and proof system are presented below and used to verify a type-based refactoring. Non-recursive and recursive definition clauses are not part of the core language but are "syntactic sugaring". We will see that a substantial amount of work goes into type-system-related results, while the correctness proof of the refactoring is comparatively straightforward.

6.2 Metatheory

6.2.1 Language

We use the same conventions used when defining the previous languages, but have two more metavariables, T and t, both ranging over types. The grammar of the language is the following:

$$
\begin{array}{rcl}
M & ::= & \nabla x \\
 & | & \lambda x : T \cdot M \\
 & | & M \circ N \\
 & | & fix\ x : T \cdot M \\
 & | & unity \\
 & | & zero \\
 & | & succ\ M \\
 & | & pred\ M \\
 & | & ifz\ L\ M\ N \\
 & | & inL_T\ M \\
 & | & inR_T\ M \\
 & | & \langle M \Leftarrow x \rangle L \langle y \Rightarrow N \rangle
\end{array}
$$

The clause *fix* $x : T \cdot M$ binds x in M and is unfolded recursively to solve the fixpoint equation $x = M\,x$. This unfolding is captured by the rule FP in the logic presented in §6.2.5. *zero* is a constant of the type of natural numbers, and *succ* and *pred* are unary functions in that type. *unity* denotes the only value inhabiting the unit type. *ifz* is a ternary function and evaluates to either its second or third arguments depending on whether its first argument is *zero*. The last clause stands for "case of" expressions: if L is a left injection then the left branch is evaluated, and similar for the right branch. Note that it binds x in M and binds y in N.

Note that the language is explicitly typed; uniqueness of types is proved as part of the groundwork further down. The system of types is defined next. *Gnd* is a category of ground types; here it contains a single element Nt.

$$\begin{array}{rcl} T & ::= & \gamma Gnd \\ & | & T \to T' \\ & | & Unit \\ & | & T + T' \end{array}$$

We formalise a *typing context* as a finite map from variables to types. We will use Γ and G as metavariables ranging over type contexts. The operations *extend* and *contract* are used to manipulate the typing context. "extend" extends a context with a type for a variable, and "contract" deletes the typing of a variable in the typing context. Formally:

Definition 6.2.1 `extend` Γ `x t` $\stackrel{def}{=}$ λ`v. if v = x then Some t else` Γ `v`

Definition 6.2.2 `contract` Γ `x` $\stackrel{def}{=}$ λ`v. if v` $\neq$ `x then` Γ `v else None`

Contexts are extended safely by first deleting the typing of the variable to be typed and then extending the context. We abbreviate this extension operation using the following definition.

Definition 6.2.3 Γ`, x:t` $\stackrel{def}{=}$ `extend (contract` Γ `x) x t`

6.2.2 Type system

Type judgements will be written $\Gamma \rhd M :: T$ and their rules are:

$$\frac{G\ x = Some\ t}{G \rhd \nabla x :: t} \quad \text{TVAR}$$

$$\frac{G, x{:}t1 \rhd M :: t2}{G \rhd \lambda x{:}t1 \cdot M :: t1 \to t2} \quad \text{TABS}$$

$$\frac{G \rhd M :: t1 \to t2 \qquad G \rhd N :: t1}{G \rhd M \circ N :: t2} \quad \text{TAPP}$$

$$\frac{G,\ x{:}t \ \triangleright\ M \ ::\ t}{G \ \triangleright\ fix\ x{:}t{\cdot}M \ ::\ t} \quad \text{TFIX}$$

$$G \ \triangleright\ zero \ ::\ \gamma Nt \quad \text{TZERO}$$

$$\frac{G \ \triangleright\ M \ ::\ \gamma Nt}{G \ \triangleright\ succ\ M \ ::\ \gamma Nt} \quad \text{TSUCC}$$

$$\frac{G \ \triangleright\ M \ ::\ \gamma Nt}{G \ \triangleright\ pred\ M \ ::\ \gamma Nt} \quad \text{TPRED}$$

$$\frac{G \ \triangleright\ L \ ::\ \gamma Nt \qquad G \ \triangleright\ M \ ::\ T \qquad G \ \triangleright\ N \ ::\ T}{G \ \triangleright\ ifz\ L\ M\ N \ ::\ T} \quad \text{TIFZ}$$

$$G \ \triangleright\ unity \ ::\ Unit \quad \text{TUNIT}$$

$$\frac{G \ \triangleright\ M \ ::\ T}{G \ \triangleright\ inL_{T+T'}\ M \ ::\ T{+}T'} \quad \text{TLEFT}$$

$$\frac{G \ \triangleright\ M \ ::\ T'}{G \ \triangleright\ inR_{T+T'}\ M \ ::\ T{+}T'} \quad \text{TRIGHT}$$

$$\frac{G \ \triangleright\ L \ ::\ T{+}T' \qquad G,\ x{:}T \ \triangleright\ M \ ::\ S \qquad G,\ y{:}T' \ \triangleright\ N \ ::\ S}{G \ \triangleright\ \langle M{\Leftarrow}x\rangle\ L\ \langle y{\Rightarrow}N\rangle \ ::\ S} \quad \text{TCASE}$$

6.2.3 Language extensions

We define *let* and *letrec* expressions as readable abbreviations of expressions in the language defined above. This creates two "levels" of the language: the core language and the definitional extension. This approach is an improvement over that taken in the previous chapter to define the language used.

Definition 6.2.4 $let\ x{:}t{:=}N\ in\ M \stackrel{def}{=} \lambda x{:}t{\cdot}M \circ N$

Definition 6.2.5 $letrec\ x{:}t{:=}N\ in\ M \stackrel{def}{=} \lambda x{:}t{\cdot}M \circ fix\ x{:}t{\cdot}N$

6.2.4 Predicates and operations

In this section the principal predicates and operations are defined as in the previous language. These will be used in the definition of the typed equational logic in the next section and in subsequent formulations and proofs.

Definition 6.2.6 *Free variables*

$$
\begin{array}{ll}
FV\ \nabla i & = \{i\} \\
FV\ \lambda i{:}t{\cdot}e & = FV\ e - \{i\} \\
FV\ (e1 \circ e2) & = FV\ e1 \cup FV\ e2 \\
FV\ fix\ i{:}t{\cdot}e & = FV\ e - \{i\} \\
FV\ zero & = \emptyset \\
FV\ (succ\ M) & = FV\ M \\
FV\ (pred\ M) & = FV\ M \\
FV\ (ifz\ L\ M\ N) & = FV\ L \cup FV\ M \cup FV\ N \\
FV\ unity & = \emptyset \\
FV\ inL_t\ M & = FV\ M \\
FV\ inR_t\ M & = FV\ M \\
FV\ \langle M{\Leftarrow}x\rangle\ L\ \langle y{\Rightarrow}N\rangle & = FV\ L \cup (FV\ M - \{x\}) \cup (FV\ N - \{y\})
\end{array}
$$

Definition 6.2.7 *Bound variables*

$$
\begin{array}{ll}
BV\ \nabla i & = \emptyset \\
BV\ \lambda i{:}t{\cdot}e & = BV\ e \cup \{i\} \\
BV\ (e1 \circ e2) & = BV\ e1 \cup BV\ e2 \\
BV\ fix\ i{:}t{\cdot}e & = BV\ e \cup \{i\} \\
BV\ zero & = \emptyset \\
BV\ (succ\ M) & = BV\ M \\
BV\ (pred\ M) & = BV\ M \\
BV\ (ifz\ L\ M\ N) & = BV\ L \cup BV\ M \cup BV\ N \\
BV\ unity & = \emptyset \\
BV\ inL_t\ M & = BV\ M \\
BV\ inR_t\ M & = BV\ M \\
BV\ \langle M{\Leftarrow}x\rangle\ L\ \langle y{\Rightarrow}N\rangle & = BV\ L \cup (BV\ M \cup \{x\}) \cup (BV\ N \cup \{y\})
\end{array}
$$

The following predicate is defined as in the previous formalisation.

Definition 6.2.8 *Captures predicate*

$$Captures\ m\ n \stackrel{def}{=} \exists\, v{\in}FV\ n.\ v \in BV\ m$$

As in the previous chapter, the substitution operation used here does not prevent variable capture.

Definition 6.2.9 *Substitution*

$$
\begin{array}{ll}
\nabla i[M/x] & = \text{if } x = i \text{ then } M \text{ else } \nabla i \\
\lambda i{:}t{\cdot}e[M/x] & = \text{if } x = i \text{ then } \lambda i{:}t{\cdot}e \text{ else } \lambda i{:}t{\cdot}e[M/x] \\
(e1 \circ e2)[M/x] & = e1[M/x] \circ e2[M/x] \\
\text{fix } i{:}t{\cdot}e[M/x] & = \text{if } x = i \text{ then fix } i{:}t{\cdot}e \text{ else fix } i{:}t{\cdot}e[M/x] \\
\text{zero}[M/x] & = \text{zero} \\
\text{succ } N[M/x] & = \text{succ } (N[M/x]) \\
\text{pred } N[M/x] & = \text{pred } (N[M/x]) \\
\text{ifz } L\ N\ N'[M/x] & = \text{ifz } (L[M/x])\ (N[M/x])\ (N'[M/x]) \\
\text{unity}[M/x] & = \text{unity} \\
(\text{inL}_t\ N)[M/x] & = \text{inL}_t\ (N[M/x]) \\
(\text{inR}_t\ N)[M/x] & = \text{inR}_t\ (N[M/x]) \\
\langle M \Leftarrow x \rangle\ L\ \langle y \Rightarrow N \rangle[K/i] & = \text{if } i = x \wedge i = y \text{ then } \langle M \Leftarrow x \rangle\ L[K/i]\ \langle y \Rightarrow N \rangle \\
 & \quad \text{else if } i = x \wedge i \neq y \text{ then } \langle M \Leftarrow x \rangle\ L[K/i]\ \langle y \Rightarrow N[K/i] \rangle \\
 & \quad \text{else if } i \neq x \wedge i = y \text{ then } \langle M[K/i] \Leftarrow x \rangle\ L[K/i]\ \langle y \Rightarrow N \rangle \\
 & \quad \text{else } \langle M[K/i] \Leftarrow x \rangle\ L[K/i]\ \langle y \Rightarrow N[K/i] \rangle
\end{array}
$$

Definition 6.2.10 *A variable x is fresh in M, written $x \sharp M$, if x does not appear in M.*

$$
\begin{array}{ll}
n \sharp \nabla x & = n \neq x \\
n \sharp \lambda x{:}t{\cdot}N & = n \neq x \wedge n \sharp N \\
n \sharp N \circ N' & = n \sharp N \wedge n \sharp N' \\
n \sharp \text{fix } x{:}t{\cdot}N & = n \neq x \wedge n \sharp N \\
n \sharp \text{zero} & = \text{True} \\
n \sharp \text{succ } M & = n \sharp M \\
n \sharp \text{pred } M & = n \sharp M \\
n \sharp \text{ifz } L\ M\ N & = n \sharp L \wedge n \sharp M \wedge n \sharp N \\
n \sharp \text{unity} & = \text{True} \\
n \sharp \text{inL}_t\ N & = n \sharp N \\
n \sharp \text{inR}_t\ N & = n \sharp N \\
n \sharp \langle M \Leftarrow x \rangle\ L\ \langle y \Rightarrow N \rangle & = n \sharp L \wedge n \sharp M \wedge n \neq x \wedge n \sharp N \wedge n \neq y
\end{array}
$$

Definition 6.2.11 *Subexpressions*

$$
\begin{array}{lll}
M \subseteq_\Lambda \nabla x & = & M = \nabla x \\
M \subseteq_\Lambda \lambda x{:}t{\cdot}N & = & M = \lambda x{:}t{\cdot}N \vee M \subseteq_\Lambda N \\
M \subseteq_\Lambda N \circ N' & = & M = N \circ N' \vee M \subseteq_\Lambda N \vee M \subseteq_\Lambda N' \\
M \subseteq_\Lambda \mathit{fix}\ x{:}t{\cdot}N & = & M = \mathit{fix}\ x{:}t{\cdot}N \vee M \subseteq_\Lambda N \\
M \subseteq_\Lambda \mathit{zero} & = & M = \mathit{zero} \\
M \subseteq_\Lambda \mathit{succ}\ N & = & M = \mathit{succ}\ N \vee M \subseteq_\Lambda N \\
M \subseteq_\Lambda \mathit{pred}\ N & = & M = \mathit{pred}\ N \vee M \subseteq_\Lambda N \\
M \subseteq_\Lambda \mathit{ifz}\ L\ N\ N' & = & M = \mathit{ifz}\ L\ N\ N' \vee M \subseteq_\Lambda L \vee M \subseteq_\Lambda N \vee M \subseteq_\Lambda N' \\
M \subseteq_\Lambda \mathit{unity} & = & M = \mathit{unity} \\
M \subseteq_\Lambda \mathit{inL}_t\ N & = & M = \mathit{inL}_t\ N \vee M \subseteq_\Lambda N \\
M \subseteq_\Lambda \mathit{inR}_t\ N & = & M = \mathit{inR}_t\ N \vee M \subseteq_\Lambda N \\
M \subseteq_\Lambda \langle N1 \Leftarrow x \rangle\ L\ \langle y \Rightarrow N2 \rangle & = & M = \langle N1 \Leftarrow x \rangle\ L\ \langle y \Rightarrow N2 \rangle \vee M \subseteq_\Lambda N1 \vee M \subseteq_\Lambda L \\
 & & \vee\ M \subseteq_\Lambda N2
\end{array}
$$

6.2.5 Logic

The logic has only (typed) equations as assertions, written $\Gamma \vdash M \simeq N :: T$. The rules of the logic are organised into categories to improve readability. The structural rules serve to facilitate the handling of typed equations. The λ, equivalence and compatibility rules have also been used in the previous chapter. The last two categories provide the rules of the PCF-like language extended with sum types.

Structural rules

$$\frac{G \vdash M \simeq N :: T \qquad G\ x = \mathit{None}}{\mathit{extend}\ G\ x\ S \vdash M \simeq N :: T}\ \text{ADD}$$

$$\frac{G \vdash M \simeq N :: T \qquad x \notin \mathit{FV}\ M \cup \mathit{FV}\ N}{\mathit{contract}\ G\ x \vdash M \simeq N :: T}\ \text{DROP}$$

λ-rules

$$\frac{z \sharp e}{G \vdash \lambda x{:}t{\cdot}e \simeq \lambda z{:}t{\cdot}e[\nabla z/x] :: T}\ \text{ALPHA}$$

$$\frac{G, x{:}t \rhd e :: T \qquad G \rhd f :: t \qquad \neg\ \mathit{Captures}\ e\ f}{G \vdash \lambda x{:}t{\cdot}e \circ f \simeq e[f/x] :: T}\ \text{BETA}$$

$$\frac{G \rhd M :: S \to T \qquad x \notin \mathit{FV}\ M}{G \vdash \lambda x{:}S{\cdot}M \circ \nabla x \simeq M :: S \to T}\ \text{ETA}$$

Equivalence rules

$$\frac{G \rhd M :: T}{G \vdash M \simeq M :: T}\ \text{REFL} \qquad \frac{G \vdash M \simeq N :: T}{G \vdash N \simeq M :: T}\ \text{SYMM}$$

$$\frac{G \vdash M \simeq M' :: T \qquad G \vdash M' \simeq N :: T}{G \vdash M \simeq N :: T}\ \text{TRAN}$$

Compatibility rules

$$\frac{G \vdash M \simeq M' :: S \to T \qquad G \vdash N \simeq N' :: S}{G \vdash M \circ N \simeq M' \circ N' :: T}\ \text{compApp}$$

$$\frac{G, x{:}S \vdash M \simeq N :: T}{G \vdash \lambda x{:}S{\cdot}M \simeq \lambda x{:}S{\cdot}N :: S \to T}\ \text{compAbs}$$

$$\frac{G \rhd \nabla x :: S \qquad G \rhd N :: T \qquad G \vdash M \simeq M' :: S \qquad \neg\, \mathit{Captures}\ N\ M \qquad \neg\, \mathit{Captures}\ N\ M'}{G \vdash N[M/x] \simeq N[M'/x] :: T}\ \text{substCong}$$

PCF-theory rules

$$\frac{G, x{:}T \rhd M :: T \qquad \neg\, \mathit{Captures}\ M\ M}{G \vdash \mathit{fix}\ x{:}T{\cdot}M \simeq M[\mathit{fix}\ x{:}T{\cdot}M/x] :: T}\ \text{FP}$$

$$G \vdash \mathit{pred}\ \mathit{zero} \simeq \mathit{zero} :: \gamma Nt\ \text{predZero}$$

$$G \vdash \mathit{pred}\ (\mathit{succ}\ M) \simeq M :: \gamma Nt\ \text{predSucc}$$

$$\frac{G \rhd M :: T \qquad G \rhd N :: T}{G \vdash \mathit{ifz}\ \mathit{zero}\ M\ N \simeq M :: T}\ \text{IFZ}$$

$$\frac{G \rhd M :: T \qquad G \rhd N :: T \qquad G \rhd L :: \gamma Nt}{G \vdash \mathit{ifz}\ (\mathit{succ}\ L)\ M\ N \simeq N :: T}\ \text{IFZ2}$$

$$\frac{G \vdash M \simeq N :: \gamma Nt}{G \vdash \mathit{pred}\ M \simeq \mathit{pred}\ N :: \gamma Nt}\ \text{compPred}$$

$$\frac{G \vdash M \simeq N :: \gamma Nt}{G \vdash \mathit{succ}\ M \simeq \mathit{succ}\ N :: \gamma Nt}\ \text{compSucc}$$

$$\frac{G, x{:}T \vdash M \simeq N :: T}{G \vdash \mathit{fix}\ x{:}T{\cdot}M \simeq \mathit{fix}\ x{:}T{\cdot}N :: T}\ \text{compFP}$$

$$\frac{G \vdash L \simeq L' :: \gamma Nt \qquad G \vdash N \simeq N' :: T \qquad G \vdash M \simeq M' :: T}{G \vdash \mathit{ifz}\ L\ M\ N \simeq \mathit{ifz}\ L'\ M'\ N' :: T}\ \text{compIf}$$

Coproduct rules

$$\frac{G \rhd L :: T \qquad G,\ x{:}T \rhd M :: S \qquad G,\ y{:}T' \rhd N :: S \qquad \neg\ \mathit{Captures}\ M\ L}{G \vdash \langle M \Leftarrow x\rangle\ \mathit{inL}_{T+T'}\ L\ \langle y \Rightarrow N\rangle \simeq M[L/x] :: S}\ \text{LEFT}$$

$$\frac{G \rhd L :: T' \qquad G,\ x{:}T \rhd M :: S \qquad G,\ y{:}T' \rhd N :: S \qquad \neg\ \mathit{Captures}\ N\ L}{G \vdash \langle M \Leftarrow x\rangle\ \mathit{inR}_{T+T'}\ L\ \langle y \Rightarrow N\rangle \simeq N[L/y] :: S}\ \text{RIGHT}$$

$$\frac{G \rhd M :: S{+}T}{G \vdash M \simeq \langle \mathit{inL}_{S+T}\ \nabla x \Leftarrow x\rangle\ M\ \langle y \Rightarrow \mathit{inR}_{S+T}\ \nabla y\rangle :: S{+}T}\ \text{CASEETA}$$

The rules of the logic suggest that even verifying trivial refactorings for this language will require type-related results to be proved first, since the formulation of the refactorings will have side-conditions, inherited from the logic, asserting well-typing of the original program. The preconditions are therefore stronger than those of the untyped language, and the practical significance is that the meaning of programs is preserved only if it exists (and it can only exist if programs are well-typed). The supporting type-related results are discussed next.

6.2.6 Type-related lemmata

Most of the work in verifying the "enlarge the definition type" refactoring actually went into proving metatheorems about the type system. We describe these next.

Lemma 6.2.12 (Inversion)

(i) $\Gamma \rhd \nabla x :: t \Longrightarrow \Gamma\ x = \mathit{Some}\ t$

(ii) $\Gamma \rhd \lambda x{:}t1{\cdot}M :: T \Longrightarrow \exists\, t2.\ T = t1 \to t2 \wedge \Gamma,\ x{:}t1 \rhd M :: t2$

(iii) $\Gamma \rhd M \circ N :: t2 \Longrightarrow \exists\, t1.\ \Gamma \rhd M :: t1 \to t2 \wedge \Gamma \rhd N :: t1$

(iv) $\Gamma \rhd \mathit{fix}\ x{:}T{\cdot}M :: S \Longrightarrow S = T \wedge \Gamma,\ x{:}T \rhd M :: T$

(v) $\Gamma \rhd \mathit{unity} :: T \Longrightarrow T = \mathit{Unit}$

(vi) $\Gamma \rhd \mathit{zero} :: T \Longrightarrow T = \gamma Nt$

(vii) $\Gamma \rhd \mathit{succ}\ M :: T \Longrightarrow T = \gamma Nt \wedge \Gamma \rhd M :: T$

(viii) $\Gamma \rhd \mathit{pred}\ M :: T \Longrightarrow T = \gamma Nt \wedge \Gamma \rhd M :: T$

(ix) $\Gamma \rhd \mathit{ifz}\ L\ M\ N :: T \Longrightarrow \Gamma \rhd L :: \gamma Nt \wedge \Gamma \rhd M :: T \wedge \Gamma \rhd N :: T$

(x) $\Gamma \rhd \mathit{inL}_{T+T'}\ M :: S \Longrightarrow S = T{+}T' \wedge \Gamma \rhd M :: T$

(xi) $\Gamma \rhd \mathit{inR}_{T+T'}\ M :: S \Longrightarrow S = T{+}T' \wedge \Gamma \rhd M :: T'$

(xii) $G \rhd \langle M \Leftarrow x \rangle\ L\ \langle y \Rightarrow N \rangle :: S \Longrightarrow \exists T\ T'.\ G \rhd L :: T + T' \wedge G, x{:}T \rhd M :: S \wedge G, y{:}T' \rhd N :: S$

Proof sketch Routine by case analysis on the structure of type judgements.

∎

Using this lemma we prove that if a pre-term (expression) is typable then it has a unique type. This is not required for the refactoring, but serves to check the definitions and increases our confidence in the formalisation.

Lemma 6.2.13 (Uniqueness)

$$\frac{G \rhd N :: S \qquad G \rhd N :: T}{S = T}$$

Proof sketch Induction on the structure of the derivation of type judgements and using the inversion lemma.

∎

The principal result required before verifying the refactoring is the *substitution lemma*. Prior to proving this, we need the following lemmata. *Weakening* and *strengthening* involve broadening or restricting the typing context in a judgement while preserving an expression's type.

Lemma 6.2.14

(i) (Weakening): $\dfrac{G \rhd M :: T \qquad x \notin FV\ M}{extend\ G\ x\ t \rhd M :: T}$

(ii) (Strengthening): $\dfrac{G \rhd L :: T \qquad x \notin FV\ L}{contract\ G\ x \rhd L :: T}$

Proof sketch Induction on the structure of the derivation of type judgements, using the typing rules and rewriting using the definitions of *extend* and *contract*. Furthermore, the rule of extensional equality, defined in HOL, is used in order to prove equality between typing contexts.

∎

In the course of building these proofs we experimented with different initial formulations. *Weakening'* is a second weakening lemma that differs from the first by focusing on the contents of the context rather than the preterm's free variables. This lemma uses the predicate *compatible1* defined over a context, a variable, and its type. This predicate is true when extending the context with that variable typing is indeed a compatible extension (i.e. does not override an existing and differing typing of that variable in the context). It is defined formally next.

Definition 6.2.15 $compatible1\ \Gamma\ v\ t \stackrel{def}{=} \Gamma\ v = None \vee \Gamma\ v = Some\ t$

Lemma 6.2.16 (Weakening')

$$\frac{G \rhd M :: T \qquad \mathit{compatible1}\ G\ x\ t}{\mathit{extend}\ G\ x\ t \rhd M :: T}$$

Proof sketch As for previous proof, except that the definitions of *compatible1* is also used for rewriting.

∎

In the case where the substitution operation does not modify an expression we use the following lemma:

Lemma 6.2.17 (idNoSubst)

$$x \notin FV\ N \wedge G \rhd N :: T \longrightarrow G \rhd N[L/x] :: T$$

Proof sketch First proving that if $x \notin FV\ N$ then $N[L/x] = N$ by structural induction on N, then using this equation on the type judgement and using the assumption $G \rhd N :: T$.

∎

Using the foundation provided by the lemmata described so far, we can now prove the substitution lemma. This asserts that the substitution operation is type-sound.

Lemma 6.2.18 (Substitution lemma)

$$\frac{G \rhd N :: S \qquad G \rhd \nabla x :: T \qquad \neg\ \mathit{Captures}\ N\ L \qquad G,\ x{:}T' \rhd L :: T}{G,\ x{:}T' \rhd N[L/x] :: S}$$

Proof sketch We split the goal by case analysis on $x \notin FV N$, and use Lemma 6.2.17 for the first case.

The second case uses induction on the structure of the derivation of type judgements and appealing to lemmata described previously. It is straightforward for most cases. The case when N is a term of type $T + T'$ requires a larger proof since it splits into three subgoals (see rule TCASE), and two of these proof obligations split further to eliminate cases when the locally-bound variables are equal to x or not (see the definition of the substitution operation, Definition 6.2.9). More concretely, consider the term $\langle M' \Leftarrow x' \rangle L' \langle y' \Rightarrow N' \rangle$. Then by the assumption obtained through the first case split in the proof, x must be free in at least one of M', L' or N'. Also, x might be equal to the bound variables x' or y' (or both).

As a short aside, the only other cases in the proof involving bound variables occur when N is either an abstraction or a fixpoint expression. In both these cases the bound variable could not have been x since it would contradict the assumption that x is free in the expression – recall this assumption is due to the first case split.

Back to the case where N is an expression of sum type. We need to analyse the cases when x might be equal to either of the locally-bound variables. In either

situation, by the rule TCASE the context $\Gamma, x : T'$ is extended with $z : t$ (where z is either x' or y' and t is the respective type). Then if $x = z$ then the context is $\Gamma, x : T', x : t$ and we prove that this is equal to $\Gamma, x : t$ (using the extensional equality rule in HOL) to show that this case leads to a type judgement having the correct type.

In the case x and z are distinct then we must show that the term L (to be substituted for x) is of type T in this extended context. Since we have assumed that $(\neg Captures(\langle M' \Leftarrow x' \rangle L' \langle y' \Rightarrow N' \rangle)\ L)$ then we know that x' and y' cannot be free in L. The new context is the old context that typed L as T (i.e. $\Gamma, x : T'$) extended with a typing for z. We have established that z is not free in L and $z \neq x$. Then the new context is a weakened version of the original $\Gamma, x : T'$, so the type of L must be T.

∎

Proving the Substitution Lemma required the most effort, much more than verifying the refactoring.

6.3 Refactoring

6.3.1 Enlarging the definition type

This refactoring enlarges the type of a definition into a sum of types as described in §4.2.

Since we use a logic of typed equations, preserving the meaning of a program involves at least preserving its type. In symbols, modulo side-conditions the refactoring changes the expression:

$$let\ x : T := M\ in\ N$$

into the following:

$$let\ x : (T + T') := inL_{T+T'}\ M\ in\ N[\langle \nabla x' \Leftarrow x' \rangle \nabla x \langle \nabla y' \Rightarrow L \rangle / x]$$

where x', y' are fresh variables and L is a newly-introduced expression. Some of the side-conditions will constrain the values of these variables – for example, L must be well-typed in a given context.

The statement of the theorem is fairly large, and the proof reflects this. To improve manageability the proof is split to isolate deductions in small steps. These serve as supporting lemmata needed in the overall proof. One such result merely instantiates the Substitution Lemma with values needed for verifying this refactoring:

```
G ▷ N :: S ∧
G ▷ ∇x :: T ∧ G, x:T+T', y:T' ▷ L :: T ∧ ¬ Captures N ⟨∇x'⇐x'⟩ ∇x ⟨y⇒L⟩
⟶
G, x:T+T' ▷ N[⟨∇x'⇐x'⟩ ∇x ⟨y⇒L⟩/x] :: S
```

The side-conditions of the refactoring are:

- $G \rhd N :: S$, $G \rhd \nabla x :: T$, $G \rhd M :: T$ and $G, y{:}T' \rhd L :: T$ express the requirement that the original program is well-typed and that the newly-introduced expression is of the right type.
- $\neg$ *Captures* $N\ \langle \nabla x' \Leftarrow x' \rangle\ \nabla x\ \langle y \Rightarrow L \rangle$, $\neg$ *Captures* $N\ M$ and $\neg$ *Captures* $L\ M$ are required to ensure β-reduction can take place.
- $x' \notin FV\ M$ and $y \notin FV\ M$ are constraints on the new variables x' and y introduced by the refactoring. They do not constrain the expression M since its presence precedes the application of the refactoring.
- $x \notin FV\ L$ enables the following result:

  $$x \notin FV\ L \longrightarrow$$
  $$N[\langle \nabla x' \Leftarrow x' \rangle\ \nabla x\ \langle y \Rightarrow L \rangle / x][inL_{T+T'}\ M/x] = N[\langle \nabla x' \Leftarrow x' \rangle\ inL_{T+T'}\ M\ \langle y \Rightarrow L \rangle / x]$$

  Without this assumption, the RHS would have been:

  $$N[\langle \nabla x' \Leftarrow x' \rangle\ \nabla x\ \langle y \Rightarrow L[inL_{T+T'}\ M/x] \rangle / x]$$

  This assumption is also used to derive an intermediate result needed in the refactoring and serves to simplify the specification of the refactoring.

The last side-condition suggests that the refactoring could have been stated more generally by not constraining L. The refactoring is formalised as:

Theorem 6.3.1 (Enlarge the type of a definition)

$$\begin{array}{l} G \rhd N :: S \ \wedge \\ G \rhd \nabla x :: T \ \wedge \\ G, y{:}T' \rhd L :: T \ \wedge \\ \neg\ \textit{Captures}\ N\ \langle \nabla x' \Leftarrow x' \rangle\ \nabla x\ \langle y \Rightarrow L \rangle \ \wedge \\ G \rhd M :: T \ \wedge \\ \neg\ \textit{Captures}\ N\ M \wedge \neg\ \textit{Captures}\ L\ M \wedge x' \notin FV\ M \wedge y \notin FV\ M \wedge x \notin FV\ L \longrightarrow \\ G \vdash \textit{let}\ x{:}T := M\ \textit{in}\ N \simeq \textit{let}\ x{:}T+T' := inL_{T+T'}\ M\ \textit{in}\ N[\langle \nabla x' \Leftarrow x' \rangle\ \nabla x\ \langle y \Rightarrow L \rangle / x] :: S \end{array}$$

Proof sketch First unfold the *let* to work at the level of the core language. Use BETA on the LHS and RHS, and form a new equation from these two results using rule TRAN. At this point the equation will be:

$$N[M/x] \simeq (N[\langle \nabla x' \Leftarrow x' \rangle \nabla x \langle y \Rightarrow L \rangle / x])[inL_{T+T'}\ M/x]$$

Then show that the RHS is equal to $N[\langle \nabla x' \Leftarrow x' \rangle\ inL_{T+T'}\ M\ \langle y \Rightarrow L \rangle / x]$ (this step was anticipated in the explanation of side-conditions previously). Then using the rule SUBSTCONG results in a straightforward subproof. The resulting type judgement is proved by showing that the typing context is extensionally equal to the that used in the preconditions. This proof is also interspersed with steps to prove the preservation of typing, including use of the instantiation of the Substitution Lemma described earlier and the typing rules TLEFT and TVAR.

∎

6.4 Discussion

6.4.1 Weakening the preconditions

The predicate *Traps* is a weaker alternative to *Captures* and provides us with weaker preconditions. Unlike the definition of *Captures*, we give an executable definition for *Traps*. As previously explained, such a definition style makes it easier to use this definition in proofs since the term rewriting system can introduce the clauses of the definition automatically. We have seen this for the definitions *Fresh* and *Fresh'* in the previous chapter. However, unlike those two definitions, *Traps* and *Captures* are not logically equivalent. We will prove this result below.

Definition 6.4.1 *Traps – a "finer-grained" version of Captures*

```
Traps ∇i N x                = False
Traps λi:t·N M x            = if i ∈ FV M ∧ x ≠ i then x ∈ FV N
                              else Traps N M x
Traps (N1 ∘ N2) M x         = Traps N1 M x ∨ Traps N2 M x
Traps fix i:t·N M x         = if i ∈ FV M ∧ x ≠ i then x ∈ FV N
                              else Traps N M x
Traps zero M x              = False
Traps (succ N) M x          = Traps N M x
Traps (pred N) M x          = Traps N M x
Traps (ifz N1 N2 N3) M x    = Traps N1 M x ∨ Traps N2 M x ∨ Traps N3 M x
Traps unity M x             = False
Traps inL_t N M x           = Traps N M x
Traps inR_t N M x           = Traps N M x
Traps ⟨N1⇐y⟩ L ⟨z⇒N2⟩ M x  = y ≠ x ∧ y ∈ FV M ∧ x ∈ FV N1 ∨ z ≠ x ∧
                              z ∈ FV M ∧ x ∈ FV N2 ∨ Traps L M x
```

Relationship between Traps and Captures

Traps implies *Captures*, but not vice versa. Formally:

Lemma 6.4.2

(i) $\forall N\ M.\ (\exists x.\ \mathit{Traps}\ N\ M\ x) \longrightarrow \mathit{Captures}\ N\ M$

(ii) $\neg\ (\forall M\ N.\ \mathit{Captures}\ N\ M \longrightarrow (\exists x.\ \mathit{Traps}\ N\ M\ x))$

Proof sketch (i) is proved by induction on structure of N, unfolding *Captures*, and then reasoning set-theoretically. (ii) is proved by producing a counterexample to refute the formula's negation: instantiating N using $\nabla x \circ \lambda x{:}t{\cdot}\nabla x$ and M using ∇x.

∎

Having weaker side-conditions is beneficial since more programs may be transformed by the refactoring.

6.4.2 Conclusion

Compared to the previous chapter, working in a typed language brought a new dimension to the formulation of refactoring: for the behaviour of programs to be equal, their types must also be equal. Programs that are not well-typed are not considered to have a meaning, and therefore there is no meaning to be preserved. The refactoring rejects such programs since it requires its inputs to be well-typed. The formulation of refactorings in the previous language only regarded variable non-capture, but here well-typing becomes equally important.
Even though the refactoring studied in this chapter is simple it allowed us to make some observations:

- Proofs related to refactoring are simpler than those in the previous language since no simultaneous induction is needed. Recall that the mutual dependence between syntactic categories in the previous language made it necessary to use simultaneous induction.

- As in the previous formalisation, we have avoided post-properties in preconditions to make the refactoring computationally economical.

- There might be a more general formulation of the refactoring, that is, a formulation which effects the same transformation but guarded by weaker preconditions. One possible candidate involves using *Traps* instead of *Captures* in side-conditions.

- Since the language is typed, assumptions concerning well-typing have appeared in the refactoring's preconditions. This draws our attention to the interaction between refactoring tools and other development tools (viz. the parser, or more specifically, the type checker). A different approach to this verification would have involved embedding a type-checking algorithm for this language, verifying it wrt the static semantics and expressing the refactoring's side-conditions concerning well-typing with invocations to this algorithm.

- In terms of definition style, this chapter has various improvements over the previous one. Structuring the language into a core language and a layer of syntactic sugaring improved the presentation of the language while keeping the proof system simpler. We did not have to do simultaneous induction for proving results about this language; however this would still be necessary in languages having multiple syntactic categories – e.g. intra-module and module levels. We also benefited from using executable definitions for predicates, rather than starting from abstract definitions.

- More than half of all the work to verify this refactoring involved proving type-theoretic groundwork to arrive at the Substitution Lemma. Such extensive prior groundwork inhibits exploration. For example, changing the substitution operation slightly would have required redoing parts of the Substitution Lemma: this is easy for cases such as *zero*, but the *case of* clause

is far more challenging. The accumulation of a corpus of mechanised results would hasten the early phase of development, but perhaps further automated support is needed to adapt previous formalisations for other contexts of use.

Possible improvements to this work include:

metatheory implementation : using more machine-friendly techniques (e.g. '*locally-nameless*') to implement the language. This will be discussed in the next section.

metatheoretical definitions : using weaker predicates in preconditions – such as using *Traps* rather than *Captures*.

6.4.3 More economical proof development

Low effort techniques for mechanising results on programming languages are also valuable to other source-to-source activities apart from refactorings. For example, a source-to-source translator for different language versions could be verified using these techniques. These transformations share the importance of keeping the transformed code recognisable, partly by preserving names. Names are usually chosen by programmers and must be handled very carefully by the machine. Changing the names of variables might be distracting to programmers. Other kinds of metaprograms, such as compilers, do not have this characteristic and in their verification programs differing only in the names of bound variables can be identified and represented as the pure binding graphs.

For this reason a *name-carrying* embedding of the language syntax is usually ideal when studying refactoring. On the other hand, *anonymous* syntax lends itself better to automation since the names are abstracted away and only the pure binding graph is retained.

When implementing a refactoring the syntax can be anonymised before transforming the program, but after transformation the variables must be named again. The computer could generate names from scratch but since the choice of names in programs can matter greatly it would be preferable to attempt to adapt names from the original program. However we cannot use the original names if variable capture or name-clash is detected. This would invalidate the whole refactoring process, wasting the resources expended transforming and post-checking the program. It would have been computationally cheaper to leave names in the program and check for clashes before having done any processing.

Not every name-carrying embedding might be suitable; techniques used to study terms in the abstract might not be suitable to verify refactorings since these operate on programs. The Barendregt Variable Convention, described in the Preliminaries chapter, is too strong an assumption for programs. In our formalisation we emulate this Convention using the *Captures* predicate but the weaker alternative, *Traps*, is a more realistic predicate to be used instead.

Nonetheless, it might be useful to have an anonymous encoding of the programming language. As we have seen earlier, verifying the refactoring in the typed language involved a considerable amount of work directed at type-theoretic groundwork. Using an anonymous approach would be a partial and "lightweight" alternative to a full verification: the effort saved reasoning about name-issues could be invested in ensuring type soundness.

Part III

Chapter 7

Related work

This chapter contains a review of other work on the verification of refactorings. We focus on the refactoring of program source code: refactoring has been studied formally on more abstract descriptions of programs too (cf. Van Der Straeten et al. 2007).

The early work on refactorings will be outlined next, then more recent formal approaches to studying behaviour-preservation will be described in detail. Mens & Tourwé (2004) provide a broader survey of work done on refactoring program source code.

Griswold
Griswold's thesis is based on the observation that restructuring programs improves their maintainability, so he studies transformations that "can change the appearance or speed of a program without affecting its input/output behaviour" (Griswold 1991, p. 23). He also draws from an observation that maintenance is the most expensive phase in a program's lifetime. Griswold also gives a detailed account of related work – including transformational/derivational programming, and program optimisation. While he does not use the term "refactoring" explicitly the concept is very similar – the only difference being that refactoring is usually concerned solely with the appearance of a program and not with its speed. The behaviour of the transformations studied by Griswold is identical to that of refactorings: these transformations are "guaranteed to either succeed and produce a new program with the same meaning as the initial program, or else to fail and leave the program unchanged" (Griswold 1991, p. 23). Griswold does not assume program transformations to be behaviour-preserving and considers restructurings to be a specific class of program transformations.

Griswold studies refactoring for the language Scheme, an eagerly-evaluated, dynamically-typed, and statically-scoped language. He uses a graph-based formalism to reason about behaviour-preservation: a Program Dependence Graph (PDG) is a graph containing data-flow and control-flow information. The PDG is described in (Griswold 1991, §4) and contrasted with using the AST: the PDG facilitates reasoning about the effects of transformations on dependencies of transformed expressions. The extra information contained in the PDG (when compared

to the AST) is useful to reason about behaviour-preservation. However, the PDG is not a precise representation of the program since it lacks other information obtainable from the AST; the transformed PDG alone might not yield correct programs.

Griswold suggests using the two representations to complement each other and defines an equation (which he calls *globalisation*) describing a necessary correspondence between transformations defined over the AST and transformations defined over the PDG. This realises his intention to transform the AST while using the PDG to reason about behaviour-preservation.

Opdyke

(Opdyke 1992, §4) studied refactoring in the context of the object-oriented paradigm and described seven properties that need to be preserved in order to ensure that a refactoring is correct. The first six properties address well-formedness constraint on programs: a refactored program must be well-formed too. Some of these properties can easily be checked syntactically – e.g., "distinct class names" – and the last property, "semantically equivalent references and operations", is refined into a representation for programs and predicates that can be used in a refactoring to help guarantee behaviour-preservation.

Refactorings can be behaviour-preserving if they act within restrictions that would violate this property. Opdyke contrasts strong and weak restrictions, arguing that while it is easier to determine violation of a strong restriction, it is weaker restrictions that allow for more useful refactorings. For instance, weaker restrictions allow for more sophisticated transformations on the program. The problem is then to ensure that the refactoring is indeed behaviour-preserving for arbitrary programs given these weaker restrictions. Opdyke provides a representation for programs that goes beyond an AST: it is a graph that contains information such as the class hierarchy apart from the parsed program. Opdyke provides this structure to serve as input to a refactoring. Such rich representations are not uncommon for refactoring purposes – Eloff (2002), Griswold (1991), Mens et al. (2005) used similar representations. Opdyke also defines several functions for use in expressing preconditions for refactorings.

In (Opdyke 1992, §5) he describes "low-level refactorings" (elementary refactorings). He uses the seven properties to argue for the correctness of these refactorings. Then in (Opdyke 1992, §6-8) he describes "high-level refactorings" (composite refactorings); these are argued to be correct by the composition of correct elementary refactorings.

Roberts

Roberts (1999) focuses on a practical aspect of refactoring: producing fast and reliable refactoring tools. He studies refactorings for Smalltalk, a dynamically-typed object-oriented language with reflection facility. His contributions are also related to checking for behaviour-preservation. Roberts complements preconditions with *postconditions*: properties that are satisfied by programs after having undergone

a specific refactoring. Postconditions are used when composing refactorings; this was described further in §2.2.2.

Dependencies between refactorings are a related idea; Roberts argues that certain refactorings are carried out to enable other refactorings to be performed – e.g. renaming a variable before extracting a method. He also argues for the usefulness of undoing refactorings out-of-order. That is, changes would not only be undone strictly in reverse order, as is usual. Dependencies between refactorings also serve to support this feature: in order to undo a refactoring one must calculate other refactorings that depend on it and undo them too if their preconditions are no longer satisfied if the first refactoring were undone.

Roberts also contributed the idea of performing refactoring partly or fully during a program's runtime. He calls this *dynamic refactoring* and differentiates it into the following (Roberts 1999, §5.5):

offline dynamic refactoring involves only the analysis phase of refactoring being done at runtime.

online dynamic refactoring means both analysis and transformation phases of refactoring are done at runtime. Roberts had also described this approach in an earlier paper (Roberts et al. 1997) on the development of a refactoring tool for Smalltalk.

7.1 Li

In her PhD thesis Li (2006) describes the design and implementation of HaRe, a refactoring tool for Haskell. She describes a catalogue of Haskell refactorings and discusses how the features of different programming languages give rise to different refactorings for each programming language. Despite this there are refactorings which persist across languages, and others which require some modification when specified for new languages.

HaRe was developed as a prototype but intended for general use. Its authors addressed concerns such as usability (such as ease of installation and use), efficiency (keeping waiting time low), reliability and extensibility (through an API) in order to make the tool generally appealing. HaRe is implemented in Haskell and has been integrated with Emacs and Vi (which the authors found to be the IDEs most commonly used by Haskell programmers after conducting a survey). HaRe relies on specialised components developed elsewhere: it uses the Programatica compiler front-end to parse full Haskell'98, and uses Strafunski for traversing and transforming the abstract syntax tree (AST). Strafunski's API functions are used to perform the analysis and transformation needed by refactorings.

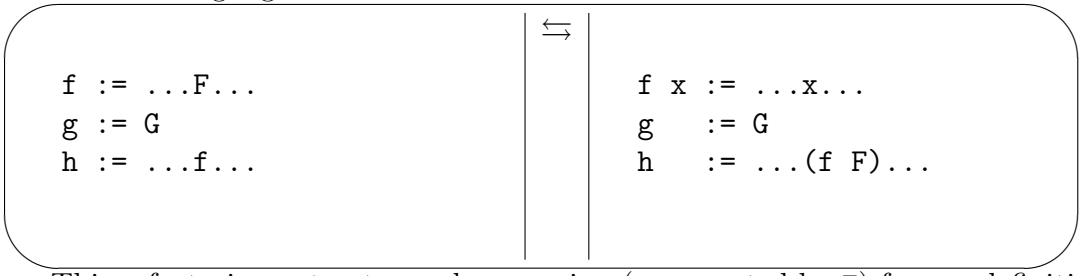
Refactoring tools usually operate on some abstract representation of the program, such as the AST, but simply pretty-printing the refactored program from the AST might be too naïve an approach since layout information about the original program is lost. It is more useful for programmers if the refactoring tool also preserves the layout of their programs since it keeps the program recognisable

by them; Haskell does not enforce a specific program layout style and therefore program layouts are often idiosyncratic. To address this concern, HaRe uses the token-stream provided by Programatica in order to preserve the appearance of program code, such as comments, indentation, etc. The importance of this is emphasised in (Li 2006, §2.4) and the algorithm used is found in (Li 2006, §C).

HaRe also offers an API of its own. This provides a higher-level interface to the APIs of Programatica and Strafunski, and facilitates the task of defining more Haskell refactorings or program transformations for other developers. The API is documented in (Li 2006, §F). HaRe's API has been used in a short internship to partially implement *warm fusion* as an exercise to study the use of HaRe's API to implement program transformations that are more general than refactorings. This is documented in Nguyen-Viet (2004). Fusion algorithms are a family of algorithms used for program *deforestation* – eliminating intermediate structures that arise in the *lazy* evaluation of a program by transforming the program statically. Fusion algorithms serve to optimise programs since they eliminate many unnecessary expensive memory allocations and operations. However the transformed programs are usually less comprehensible; fusion algorithms are not refactorings but are more general (yet behaviour-preserving) program transformations.

The formal specification and verification of refactorings is described in (Li 2006, Chapter 7); this is of direct interest to our work. In this chapter Li describes λ_M – a call-by-name, untyped λ-calculus extended with letrec and mimicking Haskell's module system. She uses λ_M to reason about module-aware Haskell refactorings. Li starts by describing λ_{Letrec}, which is λ_M without the module system. λ_{Letrec} is adapted from a theory for $\lambda\circ_{name}$, developed by Ariola & Blom (1997).

Using λ_{Letrec} and λ_M Li specifies and verifies the refactorings "Generalise a definition" and "Move a definition from one module to another". In (Li 2006, §7.3) she specifies "Generalise a definition", which we illustrate as follows using a Haskell-like language:



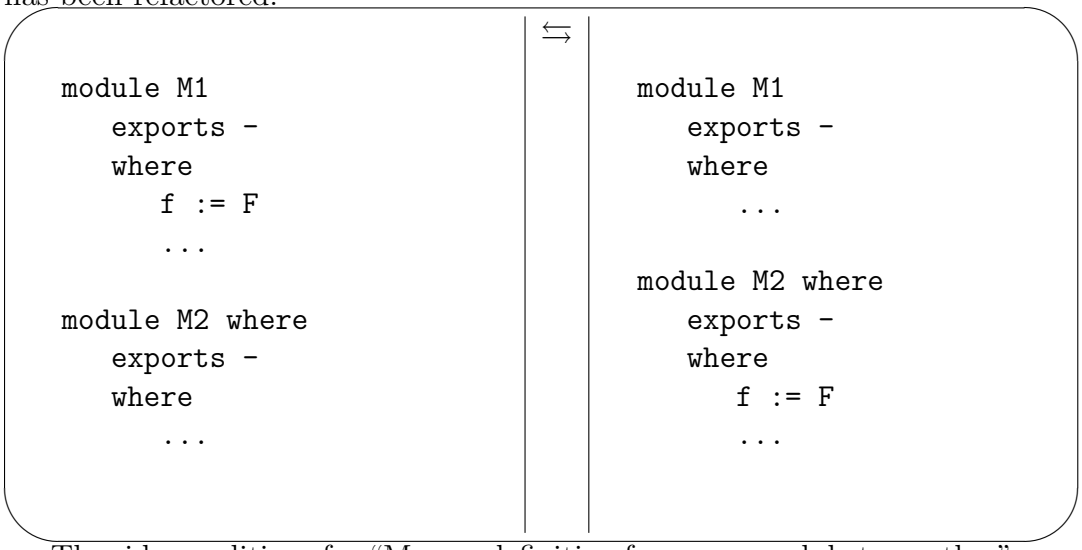
This refactoring extracts a subexpression (represented by `F`) from a definition (`f`, in the above illustration) and replaces the extracted subexpression with a fresh variable (`x`) which is abstracted from the original definition (i.e. `f := ...` becomes `f x := ...`). Finally, each calling site of the original definition is updated into the application of the defined variable to the extracted subexpression (in `h`'s definition, `...f...` becomes `...(f F)...`).

Li also outlines various other ways of interpreting the refactoring (she also discusses this "design space problem" in (Li 2006, §2.8)). In her approach she uses a *renaming* substitution, therefore the method is that of compensation rather than interaction (recall these two models have been described in §2.2.1). Renaming

is indicated by priming terms, and new variable names (e.g. `x` in the above illustration) are not explicitly indicated in side-conditions but are assumed to be fresh.

Since Li's work is centred on the development of a refactoring tool for Haskell, granularity is determined by the implementation: Li observes that "generalise a definition" can be decomposed into four refactorings but treats this refactoring as elementary, since the refactoring is implemented as an elementary refactoring for efficiency reasons.

Using the language λ_M she proceeds to study the refactoring "Move a definition from one module to another". The specification of the refactoring immediately brings to the fore many subtle issues concerning side-condition checks, such as avoiding name-clashes in modules and avoiding to introduce recursive modules. These arise because of the complexity of the relations between modules and the resulting extended scopes of definitions contained inside. Module-aware refactorings must preserve the dependencies of definitions – they must ensure that expressions dependant on a definition will still have the definition in their scope, and the definition itself will still be able to access the same definitions after the program has been refactored.



The side-conditions for "Move a definition from one module to another" ensure that the move will not disrupt the program's well-formedness. For example, if the definition to be moved already exists in the target module, the definition in the target module must have been imported from the source module if the refactoring is to succeed.

The behaviour of the transformation is conditional: it splits into four cases depending on whether the definition is being exported by the module, whether the definition has dependencies within the module, and whether the source and target modules are related (i.e. the latter is imported, directly or through some other module, by the former). A commentary follows the specification to explain it, suggesting the non-trivial nature of this refactoring. The argument for behaviour-preservation addresses two points: the refactoring *always* produces a well-formed

program, and the resulting program is behaviourally-equivalent since the refactoring does not change the definitions themselves and the definitions' dependencies are preserved.

The approach taken for the second refactoring seems more interactive than compensatory since the process is stopped whenever a side-condition fails and the user is prompted for guidance. This suggests that the approach is compensatory at the lowest level (viz. for α-convertibility), and that this is due to explanatory convenience rather than style.

Our work seeks to build on Li's to study the refactoring of functional programs. Like Li we prefer the interactive style of implementing refactorings. Compared to her approach, we do not decompose refactorings based on efficiency of implementation but on their relative elementarity. From this perspective, "generalise a definition" is considered to be a compound refactoring since it can be broken down into more elementary refactorings.

7.2 Bannwart

Bannwart (2006) studies refactoring in a Java-like imperative class-based statically-typed language for which he provides an evaluation semantics. This work is also summarised in Bannwart & Müller (2006).

Bannwart seeks to develop a general and scalable method for (i) proving refactorings to be behaviour-preserving, and (ii) applying them correctly. He is concerned with the difficulty of implementing non-trivial refactorings that are guaranteed to be behaviour-preserving. This method may also involve generating assertions to ascertain behaviour-preservation dynamically. He claims that this overcomes the shortcomings of regression testing since tests are ultimately incomplete. Moreover, Bannwart argues that having assertions in the code is a positive side-effect of the method since it improves program documentation and renders the program amenable to assertion-based verification tools. The assertions form a specification of the program, and this motivated the subtitle "refactoring with specifications" in his thesis.

In order to ensure correct application of refactorings, Bannwart suggests the following steps:

1. Establish "essential applicability conditions", i.e. conditions that ensure that the class of well-formed programs is closed under the refactoring.

2. Determine "correctness conditions" (which are usually called *side-conditions*). Conditions may be added to a program as assertions; these might be checked at runtime or else checked statically by other tools.

   Bannwart differentiates correctness conditions as:

   - properties that can be checked statically (they call them "a priori checks" in Bannwart & Müller (2006), and "preconditions" in Bannwart (2006))

- those that can be checked dynamically (called "a posteriori checks" in Bannwart & Müller (2006), and confusingly called "postconditions" in Bannwart (2006))

3. The refactoring is proved to be behaviour-preserving using the above conditions. These conditions must be sufficient to guarantee equivalence. Bannwart's notion of equivalence is "external equivalence": this is true iff two programs have the same sequence of I/O interactions with an environment, thus rendering them indistinguishable by the environment.

Before proceeding it would be helpful to outline Bannwart's formalisation. Programs operate by manipulating the state s which is a triple $(vars, heap, ext)$. Within the state, *vars* maps from variables to values, *heap* is a mapping from object identifiers to objects and *ext* is the accumulated trace of a program's I/O interactions. Thus the state maintains a partial memory of past states through *ext*. Let Γ be a program, then the program code for Γ is denoted by $code_\Gamma$. The state transition relation is a subset of the set of triples of form $t \times s \times s$, where t is the set of statements (recall that the language is imperative) and s is the set of states. The notation used is $\Gamma \vdash s \xrightarrow{t} s'$ for the transformation of state s into s' by statement t (found in the code of program Γ).

Having outlined the notation we now turn to expressing the external equivalence property. Two programs are externally equivalent iff their cumulative interactions with the outside world are identical. Let *ini* be the initial state (where all variables map to initial values, and the heap and *ext* are empty). Let s and s' be states, and *s.ext* be the projection of *ext* from state s (likewise for *s'.ext*). Programs Γ and Γ' are externally equivalent iff, adapting Bannwart's notation slightly:

$$(\Gamma \vdash ini \xrightarrow{code_\Gamma} s) \Longrightarrow \exists s'.(\Gamma' \vdash ini \xrightarrow{code_{\Gamma'}} s') \wedge (s.ext \ = \ s'.ext)$$

That is, after running the whole code in Γ' we end up with the same I/O interaction sequence as after having run program Γ. Note how this equivalence abstracts away other state details: although the intermediate states between the initial and final states might have been different, it is only the trace of I/O interaction that matters. Thus "external equivalence", as the name suggests, abstracts away internal resource usage or access patterns. In the case of there not being final states s and s' (i.e. the program is non-terminating), then finite approximations of infinite traces are considered.

In order to render provability of this property tractable, Bannwart suggests proving a stronger property such that the original property can be implied from it. This involves constructing a relation between states of the two programs, which he denotes using a prefix β_R (where R stands for a particular refactoring) that serves as a simulation relation between the original and refactored programs. The transformation operation of a refactoring is denoted by μ_R and Bannwart specifies refactorings in terms of the conditions mentioned earlier (e.g., correctness

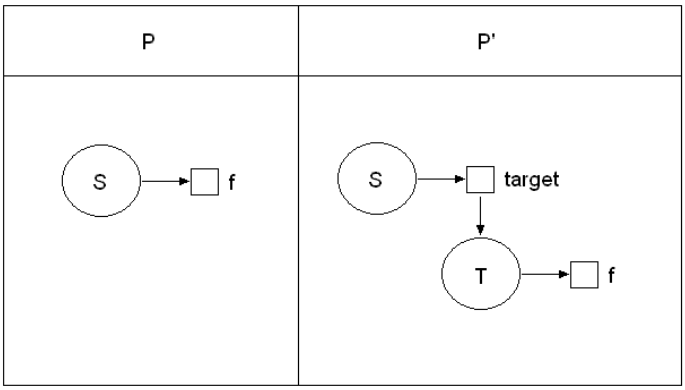


Figure 3: MoveField refactoring

conditions) and this transformation. Proof of external equivalence between a program and its refactoring is done by induction on the derivation of $\Gamma \vdash s \xrightarrow{t} s'$.

Only the correctness proof of "Rename a Method" is presented in detail in Bannwart (2006); it would have been useful to see detailed accounts for other refactorings to appreciate the tractability of the method.

Bannwart & Müller (2006) study the refactoring "Move Field". This refactoring moves a field declared in one class (let us denote it by S) to a different class (T) and modifying all accesses to that field to reflect this change. As specified in Bannwart & Müller (2006), the refactoring works by (i) moving the field, (ii) redirecting access to this field via another field (let us denote it by `target`) in S that points to T, and (iii) updating statements containing references to the moved field and making them point to the new address (i.e. via `target` in S). All other things being equal, and for the original and transformed program being denoted by P and P' respectively, the change brought about in the source object is illustrated in Figure 3. Note that for any object o of type S, the value of $o.f$ in the original program is accessed in the refactored program by reading $o.target.f$.

Note that moving a field between classes progressively builds a trail of references from the source class. This accumulation of references forms a thread from the original class containing the field to subsequent classes that the field was moved to. If the field is moved several times then the program's readability will deteriorate rapidly: comprehending the program would require understanding the sequence of deflections needed to access moved fields.

This accumulation can be appreciated by the example of moving field `f` to T from S after it had been moved to S from T. As illustrated in Figure 4, the definition of this refactoring results in a reference to `f` being refracted once again from T to S: for any object o of type S, $o.f$ in the original program is accessed via $o.target.target.f$ after the second move. This refactoring seems to be more useful in the backward direction since it simplifies the relationships between classes.

In (Bannwart 2006, §3.2.4) he mentions that refactorings need not necessarily be "symmetric" (i.e. reversible) since according to his definition it is sufficient for

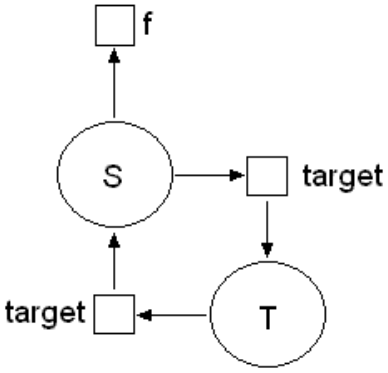


Figure 4: Using MoveField to move again to original class

a refactored program to simulate the original program, but the converse need not be true. The lack of symmetry in the relation β_R between original and refactored programs makes it impossible to induce an equivalence. The (external) equivalence however is defined as an identity relation defined over I/O traces for the original and refactored programs, and is therefore an equivalence relation. Bannwart discusses the reversal of refactorings in (Bannwart 2006, §3.3.2). Bannwart also discusses data-oriented refactorings in his thesis, and provides examples of how to implement refactorings as plugins for Visual Studio and Eclipse development environments in (Bannwart 2006, §6). Refactoring the assertions themselves is suggested as future work. The interaction between refactoring programs and assertions is discussed by Goldstein et al. (2006) and will be described in §7.6.

7.3 Mens

Mens et al. (2005) describe a method to formalise and prove refactorings to be behaviour-preserving. This method involves representing programs as graphs and refactorings as graph transformations. They describe a variety of behaviours which need to be preserved, each of which contribute to behaviour-preservation. Each of these behaviours is a property of the graph representing the original program which must be preserved in the refactored graph.

The authors argue that representing programs as an AST is not good enough since it contains both too much information (i.e. the representation is specific to a particular programming language) and too little information (e.g. control-flow information is instantly available, but other kinds of information is requires analysis to obtain). This argument was also made by Griswold (1991), his work was described at the beginning of this chapter. The authors claim that behaviour-preservation is not well-defined since behaviour is associated with run-time but refactoring tools can only manipulate source code – this point of view neglects the possibility of dynamic refactoring, put forward by Roberts (1999) and described at the beginning of this chapter. To overcome this gap the authors suggest using

a more appropriate representation for programs rather than the AST.

The representation suggested by the authors provides easier access to information (e.g. dynamic calls to a method) and can be used to define refactorings "independently of the programming language". Language genericity is challenging: truly generic refactorings might be too trivial and too few to be significantly useful since some concepts might exist in one programming language but not in another (e.g. subtyping). However the authors do not have full language genericity in mind since they are solely concerned with object-oriented languages. Recall that refactoring originated as a technique associated with object-oriented frameworks but has since been adapted for various other paradigms. The formalism suggested by Mens et al. (2005) is suited to represent languages having features such as classes, subtyping, and dynamic binding.

They study two refactorings: "Encapsulate Field" is used to encapsulate a class' public attribute by making it private and accessible through setter and getter methods. "PullUp Method" involves moving a method to a class higher in the class hierarchy. Three forms of behaviour-preservation are studied:

access preservation means that variables accessible from a method in the original program are still accessible from that method in the refactored program,

call preservation means that calls in methods in the original program are also made in the associated methods in the refactored program, and

update preservation means that variables modifiable by methods in the original program are still modifiable (though possibly through some intermediate means) by corresponding methods in the refactored program.

Mens et al. call their technique "lightweight" because it is concerned with preservation of specific types of behaviour rather than full behaviour-preservation.

The technique is limited to preserving the behaviour of meaningful programs through the specification of well-formedness criteria using *graph schemas* (also called *type graph*): graphs are well-formed if they are instances of graph schemes. For negative elimination they use *forbidden subgraphs*: graphs are malformed if they contain a forbidden subgraph. These serve as acceptance and elimination criteria constraining the set of well-formed programs.

Refactorings are formalised as graph rewrite rules parametrised by identifiers. The identifiers serve to name variables or methods, depending on the refactoring. If the rule matches a subgraph then the subgraph is transformed to the rule's production. The newly generated subgraph is then connected to the containing graph according to rules provided in *embedding mechanisms*: a set of rules that adapt the context (the rest of the program/graph) in order to link to the subgraph correctly. Further constraints on refactored programs, in order to ensure that behaviour is not changed, are expressed as *negative application conditions* (also called *negative preconditions*): if any of these conditions are satisfied then the refactoring does not take place. These conditions, together with well-formedness conditions, form the preconditions for performing a refactoring.

The behaviour being preserved (i.e. a property satisfied by both original and refactored graphs) is expressed using *path expressions* and verifying a refactoring involves proving that there is a correspondence between the original and refactored graphs wrt the path expression. For example, using the notation used in Mens et al. (2005), the path expression $B \xrightarrow[?^*a]{} V$ is satisfied by any path ending with an access (thus the final a in the expression) of the value held in a variable (to be unified with V) starting from some method body (to be unified with B). The alphabets containing a, B and V make up part of the definition of the graph language used in Mens et al. (2005). Note that these path expressions would not work with an AST since it does not express this kind of information directly. For the refactoring to be access-preserving every match of this expression in the original graph must correspond with a path in the refactored graph.

7.4 Garrido

Garrido uses Maude to formally specify and verify refactorings. She studies refactoring C's preprocessor language (Garrido 2005) and Java (Garrido & Meseguer 2006). In both cases the languages' semantics are first formalised in Maude as equational theories. Maude is an algebraic specification language and its specifications might be executable. Specifications are encapsulated in *modules*. Within modules one defines sorts and operations over them, along with relationships with other modules and local variables. In *functional* modules one can also specify a collection of equations to define the algebra's equational theory. The specifications are executable since the equations are used for term rewriting.

Garrido (2005) formalises the syntax and semantics of the language `Cpp` – C's preprocessor – in Maude. This (meta) language is independent of the programming language C, and used by programmers to organise the structure of their programs or tune the compilation according to external parameters, for instance the kind of target platform being compiled to. Prior to compilation the source code is transformed according to Cpp commands ("directives") and Cpp-defined macros are expanded.

In Garrido (2005, §6.4) she describes refactorings on C code, followed by a catalogue of refactorings for Cpp directives. The informal description of each refactoring is accompanied by fragments of Maude specifications. No proofs of correctness are offered however.

In (Garrido & Meseguer 2006) this approach is repeated to formalise Java refactorings and correctness proofs are also provided. The authors build on past work to formalise Java in Maude and seek to address the dearth of work done on formal refactoring. They identify two tasks: (i) formalising the specification of refactorings, thus producing a formalised catalogue, and (ii) proving that refactorings are indeed behaviour-preserving.

They give the following reasons why their approach is appealing:

- refactorings are specified formally

- the refactorings are proved to be behaviour-preserving
- the specifications are executable
- users can add their own refactoring

In this article the authors focus on three refactorings and describe correctness proofs for two of them. The proofs are carried out using the algebra's equational theory to show that refactored programs are equal in behaviour to the original programs. Part of the specification of "Rename Temporary Variable", from (Garrido & Meseguer 2006), is shown in Figure 5. As a brief outline of the syntax: `fmod` and `endfm` enclose a functional module, `pr` indicates which other modules the current module extends, and `var` declares variables. The keywords `op` and `eq` declare operations and equations respectively. The symbol `<-` denotes the application of a refactoring on its RHS to a piece of program code on its LHS. Note how the description of the refactoring in this specification follows the usual pattern:

```
op RenameTemp : Name Name NatList
                           -> JavaBlockRefactoring.

eq B <- RenameTemp(Old, New, L)
 = if precondsRenTempHold(computeSymbolTable(B),
                                Old, New, L)
   then applyRenTemp(B, Old, New, front(L))
   else B fi .
```

The rest of the specification elaborates further on the definitions that this refactoring relies on. Garrido & Meseguer (2006) place their work in the context of a plan to derive programming tools from the specification of the programming language in order to produce correct tools. They emphasise the importance of having generic methods, that is, being able to address several languages.

7.5 Cornélio

Cornélio (2004) studies refactorings in a language called ROOL (for "Refinement Object-Oriented Language") inspired from Java. The language is a sequential class-based language with dynamic binding. The semantics of ROOL are formalised using weakest preconditions. Based on these semantics, Cornélio proves laws, i.e. equations between commands, about the language. This approach is described in detail in (Borba et al. 2004). Since ROOL is a refinement language, commands might be programs or specifications or hybrids of both, and are used within the refinement system to derive programs from specifications in a correct (meaning-preserving) manner.

Cornélio then derives refactorings by appealing to these laws to show that refactorings are "correct by construction". This approach is also used in Cornélio et al.

```
fmod RENAME-VAR-REF is
  pr JAVA-REF. pr BLOCK-REF-HELPERS. pr ST-QUERIES.
  var B:Block. vars Old New: Name. var L:NatList.
  var ST:SymbolTable. var bs:BlockStatements. var N:Nat.

  op RenameTemp : Name Name NatList
                            -> JavaBlockRefactoring.

  eq B <- RenameTemp(Old, New, L)
   = if precondsRenTempHold(computeSymbolTable(B),
                                Old, New, L)
     then applyRenTemp(B, Old, New, front(L))
     else B fi .
  op precondsRenTempHold : SymbolTable Name Name
                                    NatList -> Bool.
  eq precondsRenTempHold(ST, Old, New, L)
   = isDeclarationAt(ST, Old, L) and
       not isNameVisible(ST, New, front(L)) .

  op applyRenTemp : Block Name Name NatList -> Block.
  eq applyRenTemp({ bs }), Old, New, (0 L))
   = { applyRenTemp(bs, Old, New, L) } .
  eq applyRenTemp(bs, Old, New, (N L))
   = replaceSubtree(bs, N,
         applyRenTemp(subterm(bs, N), Old, New, L)).
  eq applyRenTemp(bs, Old, New, nil)
   = replace(Old, New, bs) .

  op replaceSubtree : BlockStatements Nat
               BlockStatements -> BlockStatements.
  op replace : Name Name BlockStatements
                                -> BlockStatements.
```

Figure 5: Part of the specification of "Rename Temporary Variable" refactoring in Maude, from (Garrido & Meseguer 2006)

class C **extends** D
  ads
  meth $m \triangleq pc$ **end**; ops
end
	$=_{cds,c}$	**class** C **extends** D
		  ads
		  ops
		end

provided
($\rightarrow$) $B.m$ does not appear in cds, c nor in ops, for any B such that $B \leq C$.
($\leftarrow$) m is not declared in ops nor in any superclass or subclass of C in cds.

Figure 6: Specification of "Method elimination" refactoring from (Cornélio 2004)

(2005) to demonstrate the use of refactoring in arranging the organisation of software into functional layers that address different concerns – e.g., a layer of purely functional behaviour and various layers of communication, such as GUI, database access, etc.

In ROOL a program consists of a sequence of classes cds and a main command c, and is denoted using $cds \bullet c$.

The equivalence of two programs $cds_1\ cds \bullet c$ and $cds_2\ cds \bullet c$, differing by the definition of the prefixed classes, is expressed as $cds_1 =_{cds,c} cds_2$ (where $=$ denotes semantic equality and $=_{cds,c}$ indicates that the LHS and RHS share a common suffix or *context*). Class-based refactorings are then specified as equations between the original and refactored classes against the equation's context.

Refactorings are specified as equations followed by their side-conditions – for example, Cornélio's definition for the *method elimination* refactoring in ROOL is shown in Figure 6. In the side-conditions, note that $B.m$ is an invocation of method m in class B and $B \leq C$ asserts that B is a subclass of C. Side-conditions that apply when moving from the left to the right of the equation are prefixed by ($\rightarrow$), and ($\leftarrow$) is used for the converse. Side-conditions that apply when refactoring in either direction are prefixed by ($\leftrightarrow$).

In (Borba et al. 2004, §8) the authors mention that some of these ideas have been mechanised using Maude (described in the previous section). More recently, Junior et al. (2007) use CafeOBJ to mechanise ROOL and some of its refactorings. CafeOBJ is a recent member of the OBJ family of wide spectrum languages, i.e. languages encompassing a programming language and an algebraic specification layer. Maude, mentioned earlier, is also related to the OBJ family. The mechanisation follows the same pattern as the formal development: refactorings are composed from the laws of ROOL. The authors give examples of mechanised specifications of refactorings and their correctness proofs. However the development was done under the assumption that the side-conditions were satisfied. That is, the side-conditions were not mechanised and the authors set this as future work.

7.6 Other

Mens & Tourwé (2004) survey research on refactoring from both a theoretical and a tool-based point of view. Some of the principal approaches to formal refactoring have been described in previous sections. We will now briefly outline other work in which formal methods play a central rôle.

Kniesel & Koch (2004) address the difficulty encountered by end-users in defining refactorings they might need. Kniesel & Koch propose a *refactoring editor*[1] to facilitate the construction of refactorings. This editor rests on their formalisation of refactoring composition. They argue that composing refactorings is an important operation but it is hindered by the difficulty of deducing the preconditions for the compound refactoring from its constituents. Inspired by Roberts' idea of "postconditions" (described in §2.2.2), the authors formalise refactorings in a way that facilitates the calculation of preconditions. They do this in a *program-independent* manner in order to avoid having to recalculate the preconditions of the composite refactoring on a per-program basis. Recall that the verification of the "Extract a definition" refactoring described in Chapter 5 involved lemmas showing that the compound preconditions implies the preconditions of intermediate refactoring steps.

Other research related to refactoring at the University of Bonn, where Kniesel & Koch's work originates, include the framework *JTransformer*, a code-querying and transformation framework for Java programs, and *GenTL* – it is similar in purpose but is language generic. It is similar to *JunGL*, described in §2.1.1, developed at the University of Oxford. A different way to improve the extensibility of refactoring tools involves producing an API using which third-parties can develop their own refactorings. This was the approach taken in the Refactoring Browser for Smalltalk (Roberts et al. 1997) and the HaRe tool for Haskell (Li 2006).

Ettinger (2007) studies refactoring for a language similar to that used by Cornélio (2004). Like Cornélio, Ettinger formalises the semantics of this language using weakest preconditions. He focuses on using *slicing* to assist in refactoring. Slicing involves extracting the portions of a program that directly or indirectly influence the value of a particular variable at a specific location in the code. The extracted code is called the *program slice* and the rest of the program is called the *complement*. Slicing can be useful in refactoring when, for example, we would like to encapsulate the code affecting a variable into a new procedure. Ideally, the extracted code would not be duplicated into a new procedure, but removed from the original program so as not to clutter it up. In order to mitigate the complexity of writing slicing algorithms, Ettinger proposes a representation of programs decomposed into an overlaid collection of *slides* (subprograms of the original program) which may contain distinct but related program code. Slides are sequentially composed together and interleaved by a padding of *compensatory code* that serves to make the transformation behaviour-preserving by preserving

[1] their prototype is called ConTraCT, for "Conditional Transformation Composition Tool"

the program's binding graph. Ettinger calls this approach to building slicing algorithms *sliding*. He presents this idea by starting from a naïve approach involving code duplication and develops optimisations from this.

Goldstein et al. (2006) describe the method of jointly refactoring programs and their contracts. Contracts originated in the programming language Eiffel and are specification for parts of a program. They assert invariants, the assumptions under which methods operate and the guarantees they provide to other methods they call. Goldstein et al. study the use of contract information when refactoring – for example, the contract might influence a refactoring's side-conditions – or the transformation of the assertions themselves as an effect of refactoring the code. This is similar in spirit to one of the ideas described by Bannwart (whose thesis was described in §7.2). The authors also describe the development of a tool, Crepe, to partially automate this process. Crepe is an Eclipse plugin that makes calls to Mathematica and uses its theorem-proving functionality to discharge proof goals pertaining to refactoring assertions.

7.7 Program transformation

One must acknowledge that refactorings are comparatively recent examples of behaviour-preservation program transformations. Earlier, "program transformation" actually implied *behaviour-preserving* program transformation. Indeed, Philipps & Rumpe (2001) claim that work on program transformation is the "roots of refactoring". They emphasise the link between refactoring and prior work in program transformation. Li described this connection too in (Li 2006, §8.2).

Program transformations are implicitly behaviour-preserving but address different concerns. Examples of program transformations include program optimisation and derivation. We have already seen some cross-fertilisation within the area: Cornélio's work, described in §7.5, applied ideas from program derivation to refactoring.

Techniques for reasoning about a class of program transformations are often applicable to other program transformations. Having previously described some techniques to formalise and mechanise refactorings, we will now outline work done to mechanise other kinds of program transformations. In our outline we describe work that also uses Isabelle/HOL but indeed various other frameworks and proof assistants have been used to verify program transformations.

Program optimisation is done by most modern compilers to render the target code more efficient in some respect. For example, the optimisations *common subexpression elimination* and *fusion* work in different ways but both minimise the memory used by the compiled program when executed.

Continuation-Passing Style (CPS) transformations are used in compilers to render explicit the control flow in a declarative program. Minamide & Okuma (2003) verify various CPS transformations using Isabelle/HOL.

Glesner et al. (2007) verify different program optimisations (e.g. unreachable code elimination) and concentrate on nonterminating programs. They formalise the semantics of programs as state-transition systems. Nonterminating programs correspond to infinite objects and the authors use coinductive reasoning to prove the transformations to be behaviour-preserving. They use Isabelle/HOL to mechanise their proofs.

BMF ("Bird-Meertens formalism"), also called *squiggol*, is a relational approach to program construction. It is an abstract form of programming involving starting from abstract specifications and deriving programs that satisfy the original specifications. This style of programming is combinator-based and is also called "point-free programming". Programs derived using such a calculus are said to be "correct by construction". This is similar in spirit to the approach taken by Cornélio (see §7.5) to derive refactorings. Glimming (2001) mechanises part of BMF using Isabelle/HOL, building on top of a mechanisation of Category Theory.

Chapter 8

Conclusions

A number of refactorings have been verified mechanically using Isabelle/HOL. The refactorings ranged from simple and elementary to compound structural and type-based refactorings. The mechanisation process also served to reveal the challenges faced when verifying refactorings formally and we have improved the technique used in the second formalisation and suggested further improvements.

Using a proof assistant incurred a startup cost but we have benefited greatly from using Isabelle to mechanise and present our results. Various similar and complementary tools exist to assist in the mechanisation of mathematics, and more are being developed for the purpose of programming language theory. For example, the tool `ott` (Sewell et al. 2007) reads specifications of programming languages and can translate them into various other languages (including LaTeX, Isabelle, Coq, etc) and can check the specification for basic flaws. These developments are very encouraging, and indeed the challenge posed by Aydemir et al. (2005) to determine the facility with which programming language theory can be mechanised was in response to the progress done in tool development. It is hoped that appropriate tool support may facilitate formal development, leading to the widespread development of correct programming tools and resulting in more correct programs for end-users. It is more pressing that metaprogramming tools are correct since they might pass on defects to other programs.

We have not made full use of the tools available. More benefit can be derived by using the proof assistant's program extraction facilities to generate certified implementations of refactorings.

In the next section observations made during Chapters 5 and 6 will be discussed, then directions for future work will be suggested.

8.1 Discussion

8.1.1 Correctness

The correctness of our proofs was checked by machine, so our trust in the proof is delegated to the proof checker. The level of assurance in the checked proofs is

increased if the proof checker's design abides to the de Bruijn principle (described in §3.1.1).

If we trust the proof checker then the validity of our proofs hinges on the definitions used in the formalisation. An omitted side-condition or mistype might make our rules too weak, leading to non-theorems becoming provable. To some extent, checking the definitions can be automated (see the tool `ott` (Sewell et al. 2007), described earlier, or the approach suggested by Cheney & Momigliano (2007) based on bounded model-checking to search for counter-examples).

8.1.2 Economy

By "economy" we refer to both the efficiency of the specified refactorings and to reducing the effort of formal development:

- When compared to Chapter 5, proof effort in Chapter 6 was facilitated by using two layers of language: the first layer (core) contains language primitives and the second layer consisted of a definitional extension ("syntactic sugaring"). During proofs one needs only to unfold the syntactic sugaring into the primitive syntax and reason at the level of the core language.

- Weak side-conditions render refactorings more generally-applicable. This was discussed with an example in §6.4.1.

- Making the terms checked by side-conditions as small as possible may optimise a refactoring's definition. Trying to split up checks on large terms into several checks on smaller terms can help since it might spare some unnecessary computation. This was discussed in §5.4.2.

- When specifying the side-conditions, we focused on checking the original program rather than the transformed version. If checks on the latter fail then the effort spent transforming the program would have been wasted. This was discussed on page 50.

- A popular rule of thumb when using a proof assistant advises to keep definitions simple, since the complexity of definitions affects the complexity of proofs.

- Rather than mechanising the complete system, one could save work by building on a foundation found in a mechanised corpus, if available.

- Apart from establishing a theorem, the formal development can be used to produce the implementation of the refactoring. The proof assistant's program extraction facility might be used to automate this.

8.1.3 Technique

In Chapter 5 we witnessed the effect of reasoning in a language having mutually-dependent syntactic categories. This occurrence is not unusual in languages: consider languages having module and intra-module levels as, for example, can be seen in Li's language λ_M in (Li 2006, §7.6) (described in §7.1).

Recall the two models of refactorings described in §2.2.1: interaction and compensation. Choosing the model is important as a starting point since it affects the definition of the transformation operations. After choosing to use the interactive model we then chose the variable-capturing definition of the substitution operation, whereas a renaming substitution operation would have seemed more natural had we taken the compensating approach. The interactive model requires making all the checks prior to transformation. A variable-capturing substitution operation is used since the checks done earlier would have served to ensure that no capture will take place during substitution.

Had we gone for the compensation model and used a renaming substitution operation the refactoring would have had less side-conditions – for example, one would not need to check for variable capture. However, this model potentially transfers effort to the user: the user might need to perform another refactoring or choose to undo the refactoring, adapt the code, then reapply the refactoring.

As discussed earlier, if a refactoring is defined having strong side-conditions then there are less programs to which the refactoring can be applied. We explored weaker predicate definitions in the mechanised results. The $\neg$*Captures* predicate formalises the Barendregt Variable Convention (BVC), described in the Preliminaries chapter. While this Convention is suitable to facilitate reasoning about programs in the abstract, the BVC is too strong an assumption when reasoning about normal programs since they not written to adhere to this variable-naming principle. An improvement to using *Captures* was discussed in §6.4.1.

8.1.4 Readability

We tried to specify the behaviour-preservation theorems such that they expose the behaviour of the refactoring. That is, just by looking at the theorem one can understand both the side-conditions and the transformation. Specifications such as those in Maude, described in §7.4, are very rich in information but this can degrade readability.

We chose this approach over another approach that was purely directed at verifying the refactorings. Let r be a refactoring defined in the metalogic (since refactorings are total functions they can be encoded in HOL in this manner) and let $\vec{i}$ represent the arguments given to r. The correctness theorems would then all look alike modulo r and $\vec{i}$:

$$\forall p\, \vec{i}.\ (r\, p\, \vec{i}) \simeq p$$

While still establishing correctness, such a formulation would not have exposed the behaviour of the refactoring being verified.

8.2 Future work

There are several directions in which future work can be pursued.

Technical: The technical direction entails exploring different ways of achieving the same thing to derive empirical contrasts. One might compare the difficulty and benefits of using algebraic systems such as Maude or CafeOBJ to using LCF-style proof assistants such as Coq or Isabelle, or using a logical framework such as Twelf.

Another variation involves comparing different language encodings used when verifying refactorings. Some encodings were described in the Preliminaries chapter and in §3.2.

One could also extract certified refactorings from the correctness proofs. This would also entail working to extract efficient implementations of refactorings.

Method: One could use a weaker equivalence to verify refactorings, perhaps by adapting the method used by Glesner et al. (2007) described in §7.7. Through such an approach one could study refactorings effecting more powerful transformations.

Language-related: More work on formal refactoring in larger programming languages needs to be done. Despite that the language used in Chapter 5 is less sophisticated than that in Chapter 6 due to being untyped, it exposed the complexity of reasoning about languages consisting of multiple syntactic categories. Such a multicategorical organisation in realistic languages occurs frequently – for instance, in languages having module systems – so future work could seek to fuse the two styles of languages studied in this dissertation. Another possibility for future work could involve *language-generic* refactorings, e.g. for type-based refactorings in object-oriented languages as suggested by Mens et al. (2005) (described in §7.3).

Tool-related: One could also focus on the interface between refactoring tools and other tools in a programmer's toolchain. For example, in Chapter 6 one could appreciate the interaction between the refactoring and the typechecker. In this refactoring some of the side-conditions involved invoking the type checker on parts of the program and on the new expression to be introduced in the program.

In their work Goldstein et al. (2006) exploit the interaction between a refactoring tool and a theorem prover. Their work is described further in §7.6.

By studying the interface between the tools one can make explicit the obligations of each tool and verify that their combined use is correct.

Refactoring-related: The mechanised refactorings could be used in exploring the design space, described in (Li 2006, § 2.8). That is, experimenting with

different formulations of refactorings; this is useful to study differences between similar formulations and would also serve to gradually build a proper catalogue of mechanised refactorings.

For example, the refactoring described in §4.2.6 and verified in Chapter 6 could be specialised to focus on functions to produce the refactoring "Enlarge return type of a function". Rather than refactor functions of type $\tau \to \tau'$ into $(\tau \to \tau') + \sigma$, this new refactoring would instead change the type to $\tau \to (\tau' + \sigma)$. The work described in Chapter 6 could then be extended to verify this refactoring.

Another possible direction involves mechanising internal quality metrics of programs to capture the usefulness of applying a refactoring. That is, when a user chooses to apply a refactoring the tool might assess the effect of the refactoring on the program structure and inform the user about how the software metrics would be changed by the refactoring.

One could focus on the usability of refactoring tools. This might involve mechanising work such as the layout-preservation algorithm described in (Li 2006, §2.4). Such a mechanisation would require one to bridge the gap to work at a concrete level of programs – at least at the level of tokens. This would provide further assurance to users of the refactoring tool: that not only the correctness of refactoring transformations has been checked, but also that of other related pre/post-processing steps.

The work described in this dissertation has focused on interactive refactorings. The core idea of this view of refactoring is that a refactoring is not empowered to effect any lateral changes – such as rename variables – and if the refactoring attempt fails it may be retried by the user with different parameters. Despite its "pure" nature this view also seems sensible, but it would be fruitful to examine the compensating view closely and especially to study the relationship between interactive and compensating refactorings – for instance, how easily they could be intertransformed, or the appeal of using one approach over the other in certain settings. This could potentially enrich the programmer's toolset by providing them with more configurable refactorings as a result of an interactive-compensating hybrid approach.

Bibliography

A. D. Gordon & T. Melham (1996), Five axioms of alpha-conversion, *in* J. Von Wright, J. Grundy & J. Harrison, eds, 'Ninth international Conference on Theorem Proving in Higher Order Logics TPHOL', Vol. 1125, Springer Verlag, Turku, Finland, pp. 173–190.

Abadi, M., Cardelli, L., Curien, P.-L. & Lévy, J.-J. (1990), Explicit substitutions, *in* 'Conference Record of the Seventeenth Annual ACM Symposium on Principles of Programming Languages, San Francisco, California', ACM, pp. 31–46.

Abramsky, S. (1990), 'The lazy lambda calculus', pp. 65–116.

Ariola, Z. & Blom, S. (1997), *Lambda Calculi Plus Letrec*, Vrije Universiteit, Faculteit der Wiskunde en Informatica.

Aspinall, D. (2000), 'Proof General: A generic tool for proof development', *Tools and Algorithms for the Construction and Analysis of Systems (TACAS)* **1785**, 38–42.

Aydemir, B., Bohannon, A., Fairbairn, M., Foster, J., Pierce, B., Sewell, P., Vytiniotis, D., Washburn, G., Weirich, S. & Zdancewic, S. (2005), 'Mechanized metatheory for the masses: The POPLmark challenge', *Proceedings of the Eighteenth International Conference on Theorem Proving in Higher Order Logics (TPHOLs 2005)*.

Backus, J. (1978), 'Can programming be liberated from the von Neumann style?: a functional style and its algebra of programs', *Communications of the ACM* **21**(8), 613–641.

Bannwart, F. (2006), Changing software correctly, Master's thesis, ETH Zürich.

Bannwart, F. & Müller, P. (2006), Changing programs correctly: Refactoring with specifications, *in* J. Misra, T. Nipkow & E. Sekerinski, eds, 'Formal Methods (FM)', Vol. 4085 of *Lecture Notes in Computer Science*, Springer-Verlag, pp. 492–507.

Barendregt, H. (1981), *The Lambda Calculus, its Syntax and Semantics*, North-Holland.

Barendregt, H. (1997), 'The Impact of the Lambda Calculus on Logic and Computer Science', *Bulletin of Symbolic Logic* **3**(3), 181–215.

Berghofer, S. (2003), Proofs, Programs and Executable Specifications in Higher Order Logic, PhD thesis, Technische Universität München.

Berghofer, S. & Urban, C. (2007), 'A Head-to-Head Comparison of de Bruijn Indices and Names', *Electronic Notes in Theoretical Computer Science* **174**(5), 53–67.

Borba, P., Sampaio, A., Cavalcanti, A. & Cornélio, M. (2004), 'Algebraic Reasoning for Object-Oriented Programming', *Science of Computer Programming*.

Cheney, J. & Momigliano, A. (2007), Mechanized metatheory model-checking, *in* 'PPDP '07: Proceedings of the 9th ACM SIGPLAN international symposium on Principles and practice of declarative programming', ACM Press, New York, NY, USA, pp. 75–86.

Cornélio, M. (2004), Refactorings as Formal Refinements, PhD thesis, Universidade Federal de Pernambuco.

Cornélio, M., Cavalcanti, A. & Sampaio, A. (2005), 'Refactoring Towards a Layered Architecture', *Electronic Notes in Theoretical Computer Science* **130**, 281–300.

Cunha, A. (2005), 'Point-free Program Transformation', *Fundamenta Informaticae* **66**(4), 315–352.

Daniel, B., Dig, D., Garcia, K. & Marinov, D. (2007), Automated testing of refactoring engines, *in* 'Proceedings of Foundations of Software Engineering (FSE'07)', Dubrovnik, Croatia.

Davis, M. (2001), 'The early history of automated deduction', *Handbook of Automated Reasoning* **1**, 3–15.

de Bruijn, N. (1972), 'Lambda-calculus notation with nameless dummies: a tool for automatic formula manipulation with application to the Church-Rosser theorem', *Indag. Math* **34**(5), 381–392.

Drape, S. (2004), Obfuscation of Abstract Data Types, PhD thesis, University of Oxford.

Eloff, J. (2002), 'Software restructuring: implementing a code abstraction transformation', *Proceedings of the 2002 annual research conference of the South African institute of computer scientists and information technologists on Enablement through technology* pp. 83–92.

Ettinger, R. (2007), Refactoring via Program Slicing and Sliding, PhD thesis, Oxford University Computing Laboratory.

Fowler, M. & Beck, K. (1999), *Refactoring: Improving the Design of Existing Code*, Addison-Wesley Professional.

Gabbay, M. J. & Pitts, A. M. (1999), A new approach to abstract syntax involving binders, *in* '14th Annual Symposium on Logic in Computer Science', IEEE Computer Society Press, Washington, DC, USA, pp. 214–224.

Garrido, A. (2005), Program Refactoring in the Presence of Preprocessor Directives, PhD thesis, University of Illinois at Urbana-Champaign.

Garrido, A. & Meseguer, J. (2006), 'Formal Specification and Verification of Java Refactorings', *Proceedings of the Sixth IEEE International Workshop on Source Code Analysis and Manipulation (SCAM'06)-Volume 00* pp. 165–174.

Glesner, S., Leitner, J. & Blech, J. (2007), 'Coinductive Verification of Program Optimizations Using Similarity Relations', *Electronic Notes in Theoretical Computer Science* **176**(3), 61–77.

Glimming, J. (2001), Logic and Automation for Algebra of Programming, Master's thesis, University of Oxford.

Goldstein, M., Feldman, Y. A. & Tyszberowicz, S. (2006), 'Refactoring with Contracts', *agile* **0**, 53–64.

Gordon, A. (1994), *Functional Programming and Input/Output*, Cambridge Univ Pr.

Gordon, M. (2000), 'From LCF to HOL: a short history', *Proof, language and interaction: essays in honour of Robin Milner, Foundations of Computing* pp. 169–185.

Gordon, M. & Melham, T. (1993), *Introduction to HOL: a theorem proving environment for higher order logic*, Cambridge University Press New York, NY, USA.

Grioen, D. & Huisman, M. (1998), 'A comparison of PVS and Isabelle/HOL', *Theorem Proving in Higher Order Logics: 11th International Conference, TPHOLs* pp. 123–142.

Griswold, W. (1991), Program Restructuring as an Aid to Software Maintenance, PhD thesis, University of Washington.

Gunter, C. (1992), *Semantics of Programming Languages: Structures and Techniques*, MIT Press.

Haftmann, F., Klein, G., Nipkow, T. & Schirmer, N. (2005), Latex sugar for isabelle documents, Distributed with the Isabelle system.

Hankin, C. (1994), *Lambda Calculi: A Guide for Computer Scientists*, Clarendon Press.

Hughes, J. (1989), 'Why functional programming matters', *The Computer Journal* **32**(2), 98–107.

Junior, A., Silva, L. & Cornélio, M. (2007), 'Using CafeOBJ to Mechanise Refactoring Proofs and Application', *Electronic Notes in Theoretical Computer Science* **184**, 39–61.

Kniesel, G. & Koch, H. (2004), 'Static composition of refactorings', *Science of Computer Programming* **52**(1-3), 9–51.

Lee, D., Crary, K. & Harper, R. (2007), 'Towards a mechanized metatheory of standard ML', *Proceedings of the 34th annual ACM SIGPLAN-SIGACT symposium on Principles of programming languages* pp. 173–184.

Li, H. (2006), Refactoring Haskell Programs, PhD thesis, Computing Laboratory, University of Kent.

Li, H. & Thompson, S. (2007), Testing Erlang Refactorings with QuickCheck, *in* 'Draft Proceedings of the 19th International Symposium on Implementation and Application of Functional Languages, IFL 2007', Freiburg, Germany.

McBride, C. & McKinna, J. (2004), 'Functional pearl: i am not a number–i am a free variable', *Proceedings of the ACM SIGPLAN workshop on Haskell* pp. 1–9.

McKinna, J. & Pollack, R. (1993), Pure type systems formalized, *in* M. Bezem & J. F. Groote, eds, 'Proceedings 1st Int. Conf. on Typed Lambda Calculi and Applications, TLCA'93, Utrecht, The Netherlands, 16–18 March 1993', Vol. 664, Springer-Verlag, Berlin, pp. 289–305.

Mens, T. & Tourwé, T. (2004), 'A survey of software refactoring', *Software Engineering, IEEE Transactions on* **30**(2), 126–139.

Mens, T., Van Eetvelde, N., Demeyer, S. & Janssens, D. (2005), 'Formalizing refactorings with graph transformations', *Journal on Software Maintenance and Evolution: Research and Practice* **17**(4), 247–276.

Milner, R. (1997), *The Definition of Standard ML*, MIT Press.

Minamide, Y. & Okuma, K. (2003), Verifying CPS transformations in Isabelle/HOL, *in* 'Proceedings of the 2003 ACM SIGPLAN workshop on Mechanized reasoning about languages with variable binding.', ACM Press, pp. 1–8.

Morris, J. H. (1968), Lambda-Calculus Models of Programming Languages, PhD thesis, MIT.

Nguyen-Viet, C. (2004), Transformation in HaRe, Technical report, Computing Laboratory, University of Kent, Canterbury, Kent, UK.

Nipkow, T. (2003), Structured Proofs in Isar/HOL, *in* H. Geuvers & F. Wiedijk, eds, 'Types for Proofs and Programs (TYPES 2002)', Vol. 2646 of *LNCS*, Springer, pp. 259–278.

Nipkow, T., Paulson, L. C. & Wenzel, M. (2002), *Isabelle/HOL: A Proof Assistant for Higher-Order Logic*, Vol. 2283 of *LNCS*, Springer. There is also the book written by LPaulson in 1994.

Opdyke, W. (1992), Refactoring Object-Oriented Frameworks, PhD thesis, University of Illinois.

Paulson, L. (1987), *Logic and Computation: interactive proof with Cambridge LCF*, Cambridge University Press.

Paulson, L. C. (1994), *Isabelle: A Generic Theorem Prover*, Vol. 828 of *Lecture Notes in Computer Science*, Springer.

Pfenning, F. & Elliot, C. (1988), Higher-order abstract syntax, *in* 'PLDI '88: Proceedings of the ACM SIGPLAN 1988 conference on Programming Language design and Implementation', ACM Press, New York, NY, USA, pp. 199–208.

Philipps, J. & Rumpe, B. (2001), Roots of Refactoring, *in* 'Tenth OOPSLA Workshop on Behavioural Semantics', Northeastern University.

Pierce, B. (1997), Foundational calculi for programming languages, *in* A. B. Tucker, ed., 'The Computer Science and Engineering Handbook', CRC Press, Boca Raton, FL.

Pierce, B. (2002), *Types and Programming Languages*, MIT Press.

Pitts, A. M. (1995), Operationally-Based Theories of Program Equivalence, *in* P. Dybjer & A. M. Pitts, eds, 'Semantics and Logics of Computation', Cambridge University Press.

Plotkin, G. (1977), 'LCF considered as a programming language', *Theoretical Computer Science* **5**(3), 223–255.

Reinke, C. (1997), Functions, Frames, and Interactions–completing a λ-calculus-based purely functional language with respect to programming-in-the-large and interactions with runtime environments, PhD thesis, Faculty of Engineering, Christian-Albrechts-University, Kiel.

Roberts, D. (1999), Practical Analysis for Refactoring, PhD thesis, University of Illinois at Urbana-Champaign.

Roberts, D., Brant, J. & Johnson, R. (1997), 'A refactoring tool for Smalltalk', *Theory and Practice of Object Systems* **3**(4), 253–263.

Rudnicki, P. (1992), 'An overview of the Mizar project', *Proceedings of the 1992 Workshop on Types for Proofs and Programs* pp. 311–332.

Scott, D. (1993), 'A type-theoretical alternative to ISWIM, CUCH, OWHY', *Theoretical Computer Science* **121**(1-2), 411–440.

Sewell, P., Zappa Nardelli, F., Owens, S., Peskine, G., Ridge, T., Sarkar, S. & Strniša, R. (2007), Ott: Effective Tool Support for the Working Semanticist. To appear.

Turner, D. (2006), Church's thesis and functional programming, *in* A. Olszewski, ed., 'Church's Thesis after 70 years", Ontos Verlag, Berlin, pp. 518–544.

Urban, C. & Tasson, C. (2005), 'Nominal techniques in Isabelle/HOL', *CADE-20*.

Van Der Straeten, R., Jonckers, V. & Mens, T. (2007), 'A formal approach to model refactoring and model refinement', *Software and Systems Modeling* **6**(2), 139–162.

Wenzel, M. (2002), Isabelle, Isar-a Versatile Environment for Human Readable Formal Proof Documents, PhD thesis, Technische Universität München.

Wiedijk, F. (2006), 'The Seventeen Provers of the World, volume 3600 of Lecture Notes in Computer Science'.