Mechanical Verification of Refactorings

Nik Sultana
University of Kent
nik.sultana@yahoo.com

Simon Thompson
University of Kent
s.j.thompson@kent.ac.uk

Abstract

In this paper we describe the formal verification of refactorings for untyped and typed lambda-calculi. This verification is performed in the proof assistant Isabelle/HOL.

Refactorings are program transformations applied to improve the design of source code. Well-structured source code is easier and cheaper to maintain, and this motivates the use of refactoring. These transformations have been implemented as programmer tools and, as with other metaprogramming tools, it is desirable that implementations of refactorings are correct. For a refactoring to be correct the refactored program must be identical in behaviour to the original program.

Since refactorings are source-to-source transformations, concrete program information matters: for example, names (of variables, procedures, etc) and program layout should also be preserved by refactoring. This is a particular characteristic of refactorings since general program transformations operate over machine representations of programs, rather than readable source code.

The paper describes the formalisation adopted, and the alternatives explored. It also reflects on some of the difficulties of performing such formalisations, the interaction between refactoring and phases such as type-checking and parsing, and the generation of correct implementations from mechanised proofs.

Categories and Subject Descriptors D.2.3 [*Software Engineering*]: Coding Tools and Techniques; F.3.1 [*Logics and Meanings of Programs*]: Specifying and Verifying and Reasoning about Programs; D.2.4 [*Software Engineering*]: Software/Program Verification; D.2.13 [*Software Engineering*]: Reusable Software

General Terms Theory, Verification

Keywords Refactoring, Isabelle/HOL

1. Introduction

Refactorings are program transformations applied to improve the design of source code. This is important since well-structured source code is easier and cheaper to maintain (Griswold 1991).

These transformations were initially carried out manually at great expense: this is a repetitive and error-prone task. In the last decade these transformations have been implemented as programs called refactoring engines. These have been collected into refactoring tools (Li and Thompson 2008) and integrated with IDEs. As with all metaprogramming software, it is desirable that refactoring engines are correct. Advances in proof assistants have made possible the certification of software by verifying the software in a proof assistant: here we use Isabelle/HOL. This approach is tractable since it is modular: rather than verifying the whole tool it could be verified in separate pieces and then combined.

The structural changes effected by refactorings must not change the behaviour of the refactored program. Moreover, since refactorings transform source code it is important that the refactored code retains the original source code's characteristics: names (of variables, procedures, etc), comments, and layout. Preserving concrete program information is very relevant to source-to-source program transformations since it keeps the source code easily "recognisable" to its author.

This paper summarises results described in (Sultana 2007) and its contributions are: (i) the first verification carried out on refactorings using an LCF-style proof assistant, (ii) a number of case-studies on the verification of refactorings in this manner to demonstrate the method. Within these contributions we explore both theoretical and practical aspects – for instance, how the problem is stated in terms of standard λ-calculus notions, how the verification process may be made more tractable using a modular approach to separate the refactoring stage from other parts of the process, and so on. These are discussed further at the end of the paper together with a discussion of limits of our approach and suggestions for future work.

The remainder of the paper is organised as follows: the refactoring process will be described next and some of the choices made during verification will be discussed. The verified refactorings are presented in §3 and related research is outlined in §4. The paper then concludes in §5 with a reflection on this work and suggests directions for future research.

2. Background

Refactorings are organised into a simple dichotomy according to their complexity and the kind of expressions they are primarily concerned with. A refactoring is said to be *elementary* if it cannot be decomposed into simpler refactorings, and it said to be *compound* if it can be decomposed into simpler refactorings applied in sequence.

If a refactoring targets the structure of program elements (definitions, expressions etc.) then it is said to be a *structural* refactoring. A refactoring is also classified according to the level of syntax it transforms, for example *module-level* refactorings transform programs at the level of modules rather than expressions. If a refactoring targets types of expressions then it is called a *type-based* refactoring. Within type-based refactorings one finds *data-oriented* refactorings: these target specifically the types modelling particular data.

PEPM'08, January 7–8, 2008, San Francisco, California, USA.
Copyright © 2008 ACM 978-1-59593-977-7/08/0001. . . $5.00

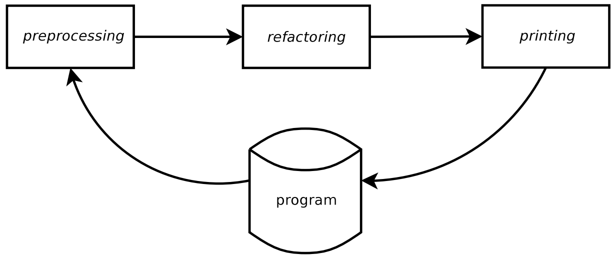


Figure 1. Automated refactoring process

2.1 Stages in refactoring

Li (2006, see Chapter 4) describes refactoring as being made up of three stages. This is illustrated in Figure 1. The preprocessing stage involves producing representations of the program that are suitable for transformation – this stage involves lexing, parsing, and possibly further processing to generate a representation of programs that is more rich than their Abstract Syntax Tree (AST), if required.

The second stage involves the actual refactoring. Applying a refactoring involves two steps: checking the refactoring's preconditions and transforming the program if the preconditions are satisfied by the program.

The last stage involves printing the program representation into the representation we usually manipulate – a list of characters. For some programming languages, such as Erlang, it suffices to pretty-print the program since there is a widely-accepted and adhered-to layout for programs (Li et al. 2006, §3.1). For other languages, such as Haskell, further processing is required to ensure that the printed refactored program mimics the layout of the original program since the language does not enforce a particular layout.

2.2 Preserving program appearance

Since the layout of Haskell programs can be idiosyncratic, transformation tools need to take this into account by restoring the original program's appearance in the transformed program. For Haskell programs one could choose between explicit delimitation using braces and using a so-called *offside* structure: the delimitation of code is inferred from the code's indentation. This is described in the Haskell Report (Jones et al. 2003, §9.3).

During manual refactoring the preservation of layout and comments is straightforward, but automating this preservation can be challenging. Li (2006, §2.4) describes the automatic preservation of program appearance for refactored Haskell programs. Her approach uses two basic program representations: the token stream and an AST annotated with type and scope information. These two representations are kept consistent (Li 2006, §4.2.3) since transformations are effected on both: the AST is transformed to effect changes to the program, and the token stream is also modified to ensure that program layout rules are adhered to following the AST's transformation. Comments are also preserved – and moved together with code deemed related – using information in the token stream and heuristics used to associate comments to code.

Besides program layout and comments, names (of variables, definitions, etc) are features that should be preserved too. Names are typically chosen with care in order to improve the program's readability. Name information can be obtained from the AST. In the work described in this paper we focus solely on the main (second) stage in the refactoring process. Within this stage we concentrate on the preservation of name information together with program behaviour. From this point onwards whenever a reference is made to *refactoring* we intend this second stage.

2.3 Correctness property

A refactoring is composed of a collection of preconditions and a program transformation. When a refactoring is applied to a program, the transformation is effected only if all the preconditions are satisfied by the program. Otherwise the program is returned unchanged. A refactoring with conjoined preconditions represented by the effective predicate Q, and effecting program transformation T, behaves thus:

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

Let $\simeq$ denote a behavioural equivalence over programs. Then in order to verify the refactoring (establishing that it is behaviour-preserving for arbitrary programs) one must prove that:

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

Apart from p, refactorings are usually parametrised by other values required by transformation T and which might also be consumed by Q. Let us assume that the parameters have already been provided and that the refactoring is a curried function – so at this stage we only see the last formal parameter: the program itself. Together with the program, the parameter values are inputs to the refactoring and the values themselves might influence whether the preconditions are satisfied. For example, the *rename a variable* refactoring is additionally parametrised by two variable names: the name to change and the name to change it to. These parameters are also provided to the refactoring's preconditions since they include provisions to ensure that name-clash does not occur as a result of transformation.

2.4 Models of refactoring

As previously explained, if the preconditions of a refactoring are not satisfied then the program is not transformed. In implementations of refactorings, if the preconditions are not satisfied then the user may be prompted to provide different parameters to the refactoring and offered the choice to abandon the refactoring. Let us call this the *interactive* model.

A different approach would involve endowing the refactorings with more automation such that they can autonomously change parts of the program in order to satisfy the preconditions. The user is later informed of these changes and might need to effect further corrective changes. For example, in the event of a name-clash the refactoring might perform renamings such that the transformation would still preserve program behaviour. By contrast, this model involves *compensating* for preconditions that are not satisfied.

These two models have analogues in the λ-calculus; for example, with regards to names a transformation can be defined in a *non-renaming* or in a *renaming* manner. These lead to interactive and compensating refactoring definitions respectively. We opt for the interactive approach in the research described in this paper. The two transformation definitions will be described further in the next section and the effect each has on the complexity of proofs will be discussed.

The interactive approach is illustrated by means of a transition diagram in Figure 2.

2.5 Transformation operations

Transformations might simply replace an (sub)expression with another, or else propagate changes in expressions by using *substitution*. Substitution is the canonical transformation operation for classical λ-calculi – other expositions of λ-calculi may use different canonical operations. For example when using nominal techniques (Urban and Tasson 2005) *swapping* is the canonical operation.

In order to facilitate reasoning about programs, programs are usually identified 'up to renaming of bound variables'. Moreover,

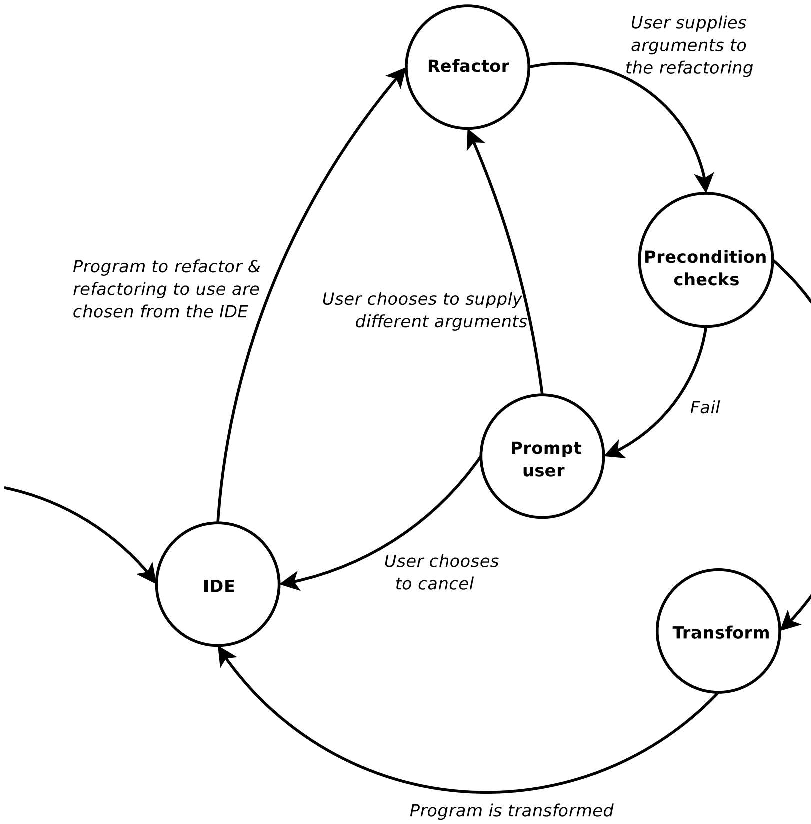

Refactor
User supplies arguments to the refactoring
Precondition checks
Program to refactor & refactoring to use are chosen from the IDE
User chooses to supply different arguments
Fail
Prompt user
Pass
IDE
User chooses to cancel
Transform
Program is transformed

Figure 2. Interactive refactoring

the substitution operations used avoid variable capture by renaming bound variables automatically on demand.

When reasoning formally this creates a contention between informal practise and the complexity of formal proofs. A *renaming* substitution operation might introduce fresh names, implying that equations between expressions in which substitution takes place must be proved modulo renaming of bound variables. In order to simplify this process it is convenient to anonymise syntax, as in de Bruijn indices and levels (de Bruijn 1972), since this avoids any explicit concrete renaming. As illustrated by Berghofer and Urban (2007), a theorem's formulation in a system encoded using first-order abstract syntax and using a renaming substitution operation would appear as follows (note that the universal closure of x and M is left implicit) :

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

Using anonymous syntax or name-carrying syntax where the set of variable names is restricted, as in the approach described by Berghofer and Urban, this result would be formulated thus:

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

Berghofer and Urban call the *mechanical* proof of the first theorem a *tour de force* due to the combination of explicit α-equivalence with the renaming substitution operation because of the latter's provision of new names. The second theorem is proved comparatively easily by straightforward induction on the structure of expressions.

Anonymous syntax is criticised because of poor readability, but in the case of refactoring this encoding is particularly unsuitable since names *do* matter, therefore they should not be abstracted away. Moreover, the model of refactoring we will use, described in the previous section, does not use a renaming substitution operation. The substitution operation used allows variable-capture and does not rename variables, so theorems are formulated as equations rather than identity modulo α-renaming.

In order to compensate for using this substitution operation the β rule will be made partial: it has non-capture as a side-condition. Variable capture can be defined strongly as in the Barendregt Variable Convention (BVC): reasoning is constrained to a subset of Λ where free variables of an arbitrary expression and bound variables of another arbitrary expression do not overlap (Barendregt 1984, §2.1.13). A weaker definition, sometimes expressed as a predicate called *Traps*, involves checking that capture does not occur when the operand is placed in occurrences of the variable in the operator. Either of these ensure that free variables do not become bound during β-reduction, thus keeping the theory consistent.

The BVC is formalised in Definition 3.2, *Traps* is defined in Definition 5.1, and §5.1.2 discusses this further.

2.6 Proof development

In order to verify the refactorings the programming languages for which refactoring was studied were embedded in the system Isabelle/HOL (Nipkow et al. 2002). Isabelle provides a metalogic within which logics can be embedded. It also provides services, for instance unification, which are inherited by object logics. Other services can be instantiated, such as extraction of proof terms. We used the logic HOL, a higher-order logic as first used in the system HOL (Gordon and Melham 1993).

Isabelle is an LCF-style proof assistant. That is, it is implemented as a library for a programming language (ML) that serves as its metalanguage. Within this metalanguage is implemented an abstract datatype of theorems and proofs are terms inhabiting that type. Proof checking is done by means of type checking – this is decidable for the type system used by ML. One of the guiding principles in Isabelle's design is its reliance on a small trusted kernel to ensure sound inference: this is called the *de Bruijn principle*.

Formal verification through theorem proving is an expensive process, but offers attractive advantages beyond the assurance it provides. The proofs may be rendered more intelligible by using a declarative style – in Isabelle this is called Isar (Wenzel 2002) – thus resembling the proofs used in mathematics. Isabelle also provides tools for marking-up formal developments for LaTeX output and extracting programs from proofs.

3. Verification case-studies

This section presents the verification of two refactorings; this work is described in full in (Sultana 2007).

The first refactoring is a *compound* refactoring. It will be specified and verified for the λ-calculus extended with recursive definitions. The second refactoring is a *type-based* refactoring defined over PCF (Plotkin 1977) extended with unit and sum types. The first language is inspired by the work in (Li 2006, see Chapter 7) on the formal verification of the *Generalise a definition* refactoring. Subsequently we studied refactorings in the second language in order to appreciate the effect of the complexity of the language and its features – including types.

Since the methods used to verify these refactorings are very similar the general approach will be outlined first then a description of the specific results will follow.

3.1 Method

Both programming languages are encoded as first-order abstract syntax with concrete variable names. The lambda calculus extended with recursive definitions has two syntactic categories: expressions and definitions. These grammatical categories are mutually dependent – this will affect definitions and proofs for this language. The second language has a single grammatical category defining expressions in the language.

In what follows we use M as a metavariable ranging over expressions, D ranges over definitions, and v over variables. Metavariables may appear primed or indexed.

Various meta-linguistic functions are defined. Several of these are standard – including the function FV that returns the set of free variables and the function BV that returns the set of bound variables in an expression. The first language has another function over expressions called DV. It returns the set of variables bound by recursive definitions. The substitution operation is defined to allow name-capture, as explained in §2.5.

Two important predicates over expressions are defined. The first predicate, *Fresh*, indicates that a variable is fresh relative to an expression. That is, the variable does not appear free or bound in the expression. A weaker definition may also be used: that the variable does not appear free in the expression. However the second definition allows shadowing to take place. The definition of *Fresh* used in the first language is given next; the definition used in the second language is similar but omits the last conjunct.

DEFINITION 3.1.

$$\mathit{Fresh}\ v\ M \overset{\mathit{def}}{=} (v \notin FV\ M) \wedge (v \notin BV\ M) \wedge (v \notin DV\ M)$$

The infix notation $v \sharp M$ will be used instead of $\mathit{Fresh}\ v\ M$. The second predicate, *Captures*, indicates whether substituting an expression into another expression will lead to previously-free variables become bound. The definition of *Captures* used in the first language is given next. The definition used in the second language is similar but omits the last disjunct.

DEFINITION 3.2.

$$\mathit{Captures}\ M\ N \overset{\mathit{def}}{=} \exists v \in FV\ N.\ (v \in BV\ M) \vee (v \in DV\ M)$$

The language semantics are defined as an equational logic over programs. This logic is embedded in HOL using an inductive relation definition. The rules of the logic induce an intensional behavioural equivalence based on $\beta\eta$-equivalence. It has already been suggested that the β-rules of both logics – each logic corresponding to one of the programming languages – are "partial" since they are conditional on $\neg$*Captures*. The predicate *Fresh* plays a rôle in the definition of the logics too: it is used in the side-condition for rule α.

Since the first language has two syntactic categories, definitions must be given for each category. For example, *Fresh* must be defined for both expressions and definitions. Functions and predicates defined for definitions will have their names suffixed with "d" – for instance, *Freshd* is the analogue of *Fresh* over definitions.

Refactorings are specified by universal HOL implications whose antecedents are the preconditions and whose consequents are formulae in the equational logic. Typically, correctness proofs are performed by induction on the structure of expressions. However in the case of the first language proofs are done by simultaneous induction. This is due to the mutually-dependent nature of grammatical categories in the language,

Verifying the type-based refactoring also required proving several lemmas about the type-system, e.g. that substitution preserves well-typing. Such results were proved by induction on the structure of the derivations of type judgements.

3.2 Elementary refactorings

Using this refactoring one could extract a definition from a program and replace occurrences of the expression it defines by the definition's name. The set of expressions in the language for which this refactoring is studied is the least set induced by the following grammar:

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

Definitions are formed from the following grammar:

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

Definitions are well-formed if and only if two parallel definitions do not define the same name.

Note that the two grammars are mutually dependent. This requires definitions to be given in "pairs" in order to be defined over both expressions and definitions. For instance, *BVd* is a function over definitions returning the set of λ-bound variables in a definition. Predicate *Freshd* is defined in the same way as *Fresh* but uses *BVd* instead of *BV*. The substitution operation we use in this language is described in Definition 3.4 – note that the operation is overloaded in order to use the same notation for substitution over expressions and substitution over definitions. The metavariables i, x will range over variables and N will range over expressions in what follows. The term $M[N/x]$ expresses the substitution of N for x in M.

The predicate *DVTop* that appears in the following definition is a restriction of *DV* to the definitions affecting the body of an expression. For instance, assume that $x \neq y$ and let L abbreviate

LEMMA 3.3. Demote a definition

$\neg \mathit{Captures}\ (\mathit{letrec}\ f \mathop{:=} \mathit{letrec}\ h \mathop{:=} N\ \mathit{in}\ M\ \mathit{in}\ L)\ N\ \wedge\ h \neq f\ \wedge\ \neg \mathit{Captures}\ L\, M\ \wedge$
$\neg \mathit{Captures}\ N\, M\ \wedge\ \neg \mathit{Captures}\ L\, h\ \wedge\ \neg \mathit{Captures}\ N\, f\ \wedge\ \neg \mathit{Captures}\ \mathit{fix}\ f$
$\longrightarrow \mathit{letrec}\ h \mathop{:=} N\ \mathit{in}\ \mathit{letrec}\ f \mathop{:=} M\ \mathit{in}\ L \simeq \mathit{letrec}\ h \mathop{:=} N\ \mathit{in}\ \mathit{letrec}\ f \mathop{:=} (\mathit{letrec}\ h \mathop{:=} N\ \mathit{in}\ M)\ \mathit{in}\ L$

the expression:

$$\mathit{letrec}\ x := (\mathit{letrec}\ y := N\ \mathit{in}\ N')\ \mathit{in}\ M$$

Then $DV\ L$ is $\{x, y\}$ and $DVTop\ L$ is $\{x\}$. The operation *DVTopd* is the analogue of *DVTop* defined over definitions instead of expressions.

DEFINITION 3.4. *Variable-capturing substitution*

$$\begin{array}{lcl}
\varepsilon[M/x] & \overset{\mathit{def}}{=} & \varepsilon \\
(y := N)[M/x] & \overset{\mathit{def}}{=} & \textsf{if } x = y \textsf{ then } y := N \\
 & & \textsf{else } y := (N[M/x]) \\
(D_1 \parallel D_2)[M/x] & \overset{\mathit{def}}{=} & \textsf{if } x \in DVTopd\ (D_1 \parallel D_2) \\
 & & \textsf{then } (D_1 \parallel D_2) \\
 & & \textsf{else } (D_1[M/x] \parallel D_2[M/x]) \\
i[M/x] & \overset{\mathit{def}}{=} & \textsf{if } x = i \textsf{ then } M \textsf{ else } i \\
(\lambda i.N)[M/x] & \overset{\mathit{def}}{=} & \textsf{if } x = i \textsf{ then } \lambda i.N \\
 & & \textsf{else } \lambda i.(N[M/x]) \\
(N \cdot N')[M/x] & \overset{\mathit{def}}{=} & (N[M/x]) \cdot (N'[M/x]) \\
(\mathit{letrec}\ D\ \mathit{in}\ N)[M/x] & \overset{\mathit{def}}{=} & \textsf{if } x \in DVTopd\ (\mathit{letrec}\ D\ \mathit{in}\ N) \\
 & & \textsf{then } (\mathit{letrec}\ D\ \mathit{in}\ N) \\
 & & \textsf{else } \mathit{letrec}\ (D[M/x])\ \mathit{in}\ (N[M/x])
\end{array}$$

The sole judgement of equational logic is defined as an inductive relation $\simeq\ \subseteq \Lambda \times \Lambda$. The rules of the logic consist of the rules for theory $\lambda\eta$ together with rules making $\simeq$ an equivalence relation compatible with respect to application and abstraction (the latter rule is often called *weak extensionality*, or ξ), and rules for converting *letrec* expressions into expressions in the pure (*letrec*-free) λ-calculus. Expressions only have meaning if they can be converted into the pure calculus.

The verification of "extract a definition" proceeds by first verifying the elementary refactorings that constitute it and then composing these results. The refactorings will be described next using pseudocode fragments together with informal descriptions of each refactoring's preconditions and the transformation it effects. This will be followed by a formal specification of the refactoring.

Add/drop a redundant definition

```
                              ⇆
  f := ...                          h := ...
                                    f := ...
```

The pseudocode fragment above illustrates the change effected by this refactoring – original code is displayed in the left pane and the refactored code is shown in the right pane.

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.

LEMMA 3.5. Add/drop a redundant definition

$$h \notin FV\ L \wedge \neg \mathit{Captures}\ L\, N \longrightarrow L \simeq \mathit{letrec}\ h \mathop{:=} N\ \mathit{in}\ L$$

The first formal precondition requires that no variable capture results from the user's choice of h. The second conjunct in the precondition might appear superfluous – particularly since it would not appear in an η-rule for *letrec*. However recall that *letrec*-expressions are not attributed any meaning unless they can be translated into *letrec*-free expressions in the pure calculus – the rules for this translation are provided as part of the logic. Once translated into pure λ-expressions the computation would proceed by β-reduction. As explained in §2.5 the β-rule used in this system is partial and the reduction is conditional upon the satisfaction of $\neg \mathit{Captures}\ L\, N$. Many similar preconditions will appear in the next refactorings for the same reason.

Demote a definition

```
                           ⇆
  h := H                         h := H
   in let f := F                  in let f := F
    in ...                         in let h := H
                                    in ...
```

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. The elementary refactorings being described here will ultimately serve to accommodate the compound refactoring being verified, and thus may seem awkward for individual application.

Compared to the refactoring described previously, this refactoring is more sophisticated and has many more preconditions. This refactoring's correctness is formulated in Lemma 3.3. Note that *fix* is not part of the grammar of the language being used here. It stands for an expression in the language that behaves like a fixpoint combinator, therefore we need to ensure that the choice of variables in this expression will not lead to capture when this expression is evaluated – thus the precondition $\neg \mathit{Captures}\ \mathit{fix}\ f$. In the language described in the next section, *fix* is a primitive notion and a precondition such as this will not be necessary since the unwinding of *fix* is done using a rule in the logic rather than by β-reducing the expression denoted by *fix*.

The precondition $h \neq f$ is needed since if $h = f$ then M and N cannot be arbitrary expressions. Finally, preconditions such as $\neg \mathit{Captures}\ L\, M$ are required to ensure that both the original and refactored programs can be translated into β-redexes – recall that this was explained for the precondition of the previous refactoring. Moreover, preconditions such as $\neg \mathit{Captures}\ L\, h$ and $\neg \mathit{Captures}\ N\, f$ are needed to ensure that the refactoring will preserve non-recursion when h and f define non-recursive definitions. These preconditions also serve to preserve recursion on the same definition. Definitions are used by simply replacing occurrences of their defining variables with the expressions they define; these preconditions ensure that the expression being replaced is not different in meaning.

Declare/inline a definition

```
f := ...N...         ⇆     f := let g := N
                            in ...g...
```

This refactoring produces a local definition from a subexpression. When applied in the opposite direction this refactoring inlines a definition in all its calling sites – that is, in all free occurrences of the variable with which it is bound. As explained for the previous refactoring, there may be different ways of specifying a particular refactoring. For instance, the definition of this particular refactoring contains a toplevel definition f which might seem unecessary. The refactoring was specified in this manner due to its rôle in the compound refactoring it features in, as will be seen in §3.3.

Some new definitions appear in the specification of this refactoring. The substitution operation $M[g\!:\!N]$ substitutes variables for expressions, $Rec\;(g\!:=\!N)$ is true whenever $g\!:=\!N$ is recursive (i.e. g is free in N), and $N \subseteq_{\Lambda} M$ is true when N is a subexpression of M.

LEMMA 3.6. Declare/Inline a definition
$\neg Rec\;(g\!:=\!N) \wedge (g \sharp (f\!:=\!M)) \wedge (N \subseteq_{\Lambda} M) \wedge$
$\neg Capturesd\;(f\!:=\!M)\,N \wedge f \notin DVTop\;N \longrightarrow$
$letrec\; f\!:=\!M\;\;in\;\;L \simeq letrec\; f\!:=\!letrec\; g\!:=\!N\;\;in\;\;M[g\!:\!N]\;\;in\;\;L$

The precondition $\neg Capturesd\;(f\!:=\!M)\,N$ ensures that the side-condition of the β-rule is satisfied. Recall that *Capturesd* is the analogue of the predicate *Captures* defined over definitions rather than expressions. As in the previous refactoring, $f \notin DVTop\,N$ ensures that non-recursion is preserved, or if $f := M$ is recursive that the recursion on the same definition will be preserved.

The proviso that $g \sharp (f := M)$ stipulates that the name chosen for the new definition is fresh. The expression in the new definition was formerly a subexpression of the main expression, by precondition $N \subseteq_{\Lambda} M$.

Recall that definitions are used by simply inlining them in place of their defining variables. The precondition $\neg Rec\;(g\!:=\!N)$ requires the definition to be non-recursive since when this definition is removed (in the right-to-left direction) recursion would no longer be made on the same definition of g and thus the meaning of the expression would have been changed.

3.3 "Extract a definition"

```
f := ...N...         ⇆     g := N
                           f := ...g...
```

"Extract a definition" is a non-trivial, compound refactoring which we define by composing the previous three refactorings using the transitivity rule. The compound refactoring involves the following steps:

1. $letrec\; f := M\;\;in\;\;L$ is the original expression, and is changed to
2. $letrec\; f\;:=\;letrec\; g\;:=\;N\;\;in\;\;M[g\;:\;N]\;\;in\;\;L$ by "declare a definition", then to
3. $letrec\; g\!:=\!N\;\;in\;\;letrec\; f\!:=\!letrec\; g\!:=\!N\;\;in\;\;M[g\!:\!N]\;\;in\;\;L$ using "add a redundant definition", and finally to
4. $letrec\; g\!:=\!N\;\;in\;\;letrec\; f\!:=\!M[g\!:\!N]\;\;in\;\;L$ by using "demote a definition".

As any compound refactoring, this refactoring inherits the preconditions of its constituent refactorings. It is not always obvious which refactorings the preconditions originate from since the preconditions might need to be adapted to optimise the refactoring. Moreover, further adaptation of the preconditions may be necessary in order to "interface" between the constituent refactorings – i.e. proving that the output of a refactoring in a compound always satisfies the precondition of a successive refactoring. Roberts (1999) calls these *postconditions*; they serve to lessen the number of potentially wasteful checks made on programs after they have been transformed.

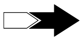
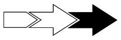
For example, when the compound refactoring arrived at the last step – that is, "lift or demote a definition" – the following had to be checked for satisfaction:

$$\begin{array}{l}\neg Captures\;(letrec\; f\!:=\!letrec\; g\!:=\!N\;\;in\;\;M[g\!:\!N]\;\;in\;\;L)\;N \wedge \\ g \neq f \wedge \neg Captures\;L\;\;(M[g\!:\!N]) \wedge \\ \neg Captures\;N\;(M[g\!:\!N]) \wedge \neg Captures\;L\;g \wedge \\ \neg Captures\;N\;f \wedge \neg Captures\;fix\;f\end{array}$$

These are the preconditions of the "lift/demote a definition" refactoring instantiated to the refactored program produced so far in the "extract a definition" refactoring. Note that the first, third and fourth conjuncts of this formula are propositions concerning a (intermediate) transformed program. As the program is transformed by each constituent refactoring in turn, the preconditions of each successive refactoring need to be satisfied by the refactored program.

Proving additional lemmas about the implication of preconditions pertaining to constituent refactorings from the preconditions of the compound refactoring supports the construction of compound refactorings. This is because it guarantees that if the compound preconditions are satisfied then the compound refactoring can be effected in its entirety. This also has economic significance: if a compound refactoring is aborted in an intermediate stage because of failed preconditions then the computing resources expended checking and transforming until that point would have been wasted.

To improve the specification of the compound refactorings we have proved such additional lemmas, and the compound refactoring process can be illustrated as follows:

1. The compound refactoring's preconditions are checked and found to be satisfied.
2.  The first constituent refactoring's transformation is effected. The satisfaction of its preconditions usually follows the compound refactoring's preconditions directly, but small adaptations may be done – for instance, where the compound's preconditions are in a different form to accommodate the preconditions of several constituent refactorings.
3.  The second constituent refactoring's transformation is effected, its preconditions having been guaranteed to be satisfied once the compound's preconditions have been satisfied.

The process continues; satisfaction of the compound's preconditions guarantees that preconditions of all constituent refactorings will be satisfied.

The main result is stated formally as follows:

THEOREM 3.7. Extract a definition

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

3.4 Type-based refactoring

The same conventions and metavariables used in the precious section will be used there. The additional metavariable T will range over types. The grammar of the language is the following:

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

The clause $fix\ x : T.M$ binds x in M and is unfolded recursively to solve the fixpoint equation $x = M\,x$. The symbol *zero* is a constant of the type of natural numbers, and *succ* and *pred* are unary functions in that type. We use *unity* to denote the only value inhabiting the unit type. An *ifz*-expression is a ternary function and evaluates to either its second or third arguments depending on whether its first argument is *zero*. The last clause in the grammar 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 in order to check our definitions. The grammar of types is defined next.

$$\begin{array}{rcl}
T & ::= & Nat \\
& | & T \rightarrow T' \\
& | & Unit \\
& | & T + T'
\end{array}$$

As one can expect, type *Nat* is the type of natural numbers, *Unit* is the unit type, $T \rightarrow T'$ is the function space and $T + T'$ forms coproducts.

A *typing context* is formalised as a finite map from variables to types. Let Γ be a metavariable ranging over typing contexts. We will use “$\Gamma, x : t$” to denote the extension of the typing context Γ with a type for x of t. Type judgements are triples expressed using the syntax $\Gamma \rhd M :: T$. The equational semantics for this language are expressed using a logic of typed equations; the notation $\Gamma \vdash M \simeq N :: T$ is used.

Recursive definitions were part of the “core” language in the approach described in §3.2. Definitions and recursive definitions are not part of the present language but are “syntactic sugaring” and are defined next in terms of the core language.

DEFINITION 3.8.

$$let\ x : T := N\ in\ M \overset{def}{=} (\lambda x : T.M) \cdot N$$

DEFINITION 3.9.

$$letrec\ x : T := N\ in\ M \overset{def}{=} (\lambda x : T.M) \cdot (fix\ x : T.N)$$

In order to verify the refactoring one must first prove type-related lemmas, such as inversion, weakening, strengthening and the substitution lemma. The substitution lemma asserts that the substitution operation is type-sound. Indeed, proving the substitution lemma required much more effort than the correctness proof for the refactoring. This lemma is stated next. The metavariable S will range over types.

LEMMA 3.10. (Substitution lemma)

$$\begin{array}{l}
\Gamma \rhd N :: S \ \wedge \\
\Gamma \rhd x :: T \ \wedge \\
\neg Captures\ N\ L \ \wedge \\
\Gamma, x : T' \rhd L :: T \\
\longrightarrow \ \Gamma, x : T' \rhd N[L/x] :: S
\end{array}$$

3.5 “Enlarge definition type”

```
                          ⇆
   x :: T                       x :: Either T T’
   x := ...                     x := Left ...

   ...(f x)...                  ...(either f L x)...
```

“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.

The refactoring is specified formally in Theorem 3.11. The preconditions $\Gamma \rhd N :: S$, $\Gamma \rhd x :: T$, $\Gamma \rhd M :: T$ and $\Gamma, y : T' \rhd L :: T$ express the requirement that the original program is well-typed and that the newly-introduced expression L is of the right type.

The precondition of the β-rule is satisfied by requiring that $\neg Captures\ N\ \langle x' \Leftarrow x' \rangle x \langle y \Rightarrow L \rangle$, $\neg Captures\ N\ M$ and $\neg Captures\ L\ M$.

The constraints $x' \notin FV\ M$ and $y \notin FV\ M$ are placed on the new variables x' and y. The constraint $x \notin FV\ L$ is placed on the newly-introduced expression L. These constraints help keep the specification of the refactoring simple since, for example, if we do not assume $x \notin FV\ L$ then the refactored program would have been transformed to:

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

4. Related work

The work described in this paper was inspired by the formal specification and verification of refactorings described by Li (2006, see Chapter 7). In that chapter of her doctoral dissertation Li studies *generalise a definition*, a non-trivial structural refactoring, and *move a definition from one module to another*, a module-level refactoring. The first refactoring is studied for the language λ_{Letrec} – this is an adaptation of $\lambda\circ_{name}$ described by Ariola and Blom (1997). In order to study the module-level refactoring Li extends λ_{Letrec} with notions inspired from Haskell’s module system.

Other related work includes the formal, but not mechanised, work by Cornélio (2004) and Ettinger (2007) for similar languages resembling a fragment of Java. The mechanisation of part

THEOREM 3.11. Enlarge definition type

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

of Cornélio's work is described by Junior et al. (2007). The approach used by Cornélio is followed closely in the mechanisation: this involves first proving laws (equations between programs) in the refinement calculus under study, then defining refactorings in terms of these laws. The refactorings would be behaviour-preserving by their construction.

The first mechanised verification of refactorings was described by Garrido and Meseguer (2006), where they use Maude to specify and verify refactorings for Java. They build on previous work in which the semantics of Java were formalised in Maude. Compared to the work described in this paper, Java is clearly a more practical object-language to address. The work described here studies fragments of functional languages embedded in an LCF-style proof assistant – the checked proofs have higher assurance due to the latter. In this work we were more concerned with studying the method rather than aiming for a more complex object language. These fragments may be extended to study more realistic languages, or alternatively the method might be adapted to study other languages. There already exist mechanisations of practical languages in LCF-style systems with similar logics – for instance C (Norrish 1998) in the system HOL – that might be adapted for this purpose.

Garrido and Meseguer (2006) state that their goal is to derive tools from executable specifications in Maude. They also plan to render their method more appealing through language genericity. In previous work, Garrido also studied the formalisation of refactorings for C's preprocessor language in Maude.

The tools Maude and CafeOBJ are related: both are algebraic specification languages and refactorings are defined as operations in an algebra. The behaviour of refactorings is described using equations in the algebra's theory. The refactorings can then be executed by performing rewriting using their equations. Garrido and Meseguer (2006) and Junior et al. (2007) seem to have similar goals albeit using different tools.

There is a wealth of other work concerned with the verification of program transformations other than refactorings. Minamide and Okuma (2003) verify the transformations of programs into Continuation-Passing Style (CPS). These transformations are useful for making the control flow explicit in declarative programs. Glesner et al. (2007) verify various optimisations on non-terminating programs. These programs are modelled as streams of states and the authors use coinductive reasoning to prove that the optimisations are behaviour-preserving. Leroy (2006) uses Coq to verify a compiler back-end translating Cminor (an intermediate language resembling C) to PowerPC assembly code. A front-end is bolted on Leroy's work by Blazy et al. (2006): they verify the front-end of a compiler translating a subset of the C programming language into Cminor. A certified compiler is then composed from the code extracted from either proof.

5. 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.

5.1 Reflections

In this section we reflect on the experience of proving these refactorings correct in Isabelle.

5.1.1 Isabelle usage

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. Rather than mechanising a 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 can be used to automate this.

The size of the Isabelle development described in this paper is around 5000 lines. Two theory files – containing the formal development – of roughly equal size were produced, one for each language studied. An Isabelle theory consists of definitions, lemmas and proofs, however it may also include, as it did in this case, additional explanatory material to improve the presentation when rendering these theories into formal documents.

Mechanisation entails finding sensible ways of encoding a theory: a naïve encoding of the informal methods, although correct, may prove too limiting or inefficient in formal practice. This process involves experimenting with techniques that are both faithful to the theory and also possess some desired practical property in terms of formal development. The variety of techniques is particularly manifest in the embedding of languages. Some of these techniques, such as de Bruijn indices, have been mentioned earlier.

As in all fully-formal work, verifying non-trivial refactorings required first discharging several smaller lemmas in order to dampen the complexity of proving the overall result. In the account provided here these results were concealed in order to convey a high-level view of the development. The amount of results unrelated to refactoring were particularly appreciable when verifying the refactoring described in §3.5: more than half of all the work needed 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.

5.1.2 Weaker preconditions

In §2.5 we briefly described the benefit of weak preconditions. Weak preconditions render refactorings more generally-applicable since they allow the transformation to be effected on a greater number of programs. Thus using weaker preconditions improves the specification of refactorings.

For instance, the predicate *Captures* used above can be replaced with a stronger alternative *Traps*, defined in Definition 5.1. It can be proved that *Traps* implies *Captures* but not vice versa.

Note that the definition of *Traps* is recursive – such a definition is amenable to automation using the proof assistant's term rewriting engine: this may facilitate developing the formal proof.

5.1.3 Economy

We sought to make the terms checked by preconditions as small as possible in order to optimise the definitions of refactorings. Trying to split up checks on large terms into several checks on smaller terms can help since it might spare some unnecessary computation.

When specifying the preconditions, we focused on checking the original program rather than the transformed version. This is beneficial since if checks on the latter fail then the effort spent transforming the program would have been wasted, as explained in §3.3. This often required further lemmata in order to prove properties about the transformed program using properties of the original program.

5.1.4 Language encoding

Low effort techniques for mechanising results on programming languages are valuable for verifying source-to-source transformations – for instance verifying a source-to-source translator for different versions of a language. These transformations share the characteristic 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 requirement; in the verification of such transformations, programs differing only in the names of bound variables can be identified and represented as 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 use names from the original program. However the original names cannot be used if variable capture or name-clash is detected. This would invalidate the whole refactoring process and waste 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 §2.5, is too strong an assumption for programs. In our formalisation we emulate this Convention using the *Captures* predicate but the weaker alternative, *Traps*, would have been a better choice.

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.

5.2 Future work

Further work can be done on a number of fronts. Some possibilities will be elaborated next.

5.2.1 Larger languages and refactorings

One direction for future work involves studying refactorings that cannot be verified using the method used here – for instance when the original and refactored programs are extensionally, but not intensionally, equivalent.

The work described in this paper focused on functional programs. Future work could also address refactorings in other language paradigms. This could complement other work done (Garrido and Meseguer 2006; Junior et al. 2007) to study the mechanisation of refactorings' correctness proofs. Mechanising the semantics of realistic programming languages can be challenging: languages may lack formal semantics, and mechanising formal semantics may require additional work to study the best means of embedding them in the proof assistant's logic.

Another possible route for research involves studying larger refactorings. It has been suggested that formalised large refactorings may appear more complex and contain conjunctions of implications in the consequent rather than just equations. For instance, the *Move a definition from one module to another* refactoring verified by Li (2006, see Chapter 7) has this kind of behaviour.

5.2.2 Mechanised catalogue of refactorings

One could also explore the design space further, as described by Li (2006, § 2.8), and gradually build a useful catalogue of mechanised refactorings. For example, the refactoring described in §3.5 could be specialised to focus on functions to produce the refactoring "Enlarge return type of a function". Rather than refactor functions of type $T \to T'$ into $(T \to T') + S$, this new refactoring would instead change the type to $T \to (T' + S)$. The work described in §3.5 could then be extended to verify this refactoring. Another possibility for future work involves extracting refactoring engines from their correctness proofs.

5.2.3 Verifying other parts of the process

One could also study the refactoring process from start to finish – encompassing the three stages described in §2.1. This might involve mechanising work such as the layout-preservation algorithm described by Li (2006, §2.4). Such a mechanisation would require one to use different representations of programs, starting at the token stream and the AST. This would involve verifying the different tools interacting during this process, including the type-checker (recall that the preconditions of "Enlarge definition type" included several propositions regarding the types of expressions). 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 stages.

Acknowledgments

Work on this research was possible thanks to financial support provided to the first author by the Computing Laboratory and by the Malta Government Scholarship Scheme through award MGSS/2006/007; the second author acknowledges the support of the EPSRC for building the HaRe and Wrangler tools. We also acknowledge the helpful feedback of the anonymous referees.

DEFINITION 5.1. Traps – *a “finer-grained” alternative to* Captures

$$
\begin{array}{lcl}
\mathit{Traps}\ i\ N\ x & \stackrel{\mathit{def}}{=} & \mathsf{False} \\
\mathit{Traps}\ (\lambda i : t.M)\ N\ x & \stackrel{\mathit{def}}{=} & \mathsf{if}\ i \in FV\ N \wedge x \neq i \\
 & & \mathsf{then}\ x \in FV\ M \\
 & & \mathsf{else}\ \mathit{Traps}\ M\ N\ x \\
\mathit{Traps}\ (M_1 \cdot M_2)\ N\ x & \stackrel{\mathit{def}}{=} & (\mathit{Traps}\ M_1\ N\ x) \vee (\mathit{Traps}\ M_2\ N\ x) \\
\mathit{Traps}\ (\mathit{fix}\ i : t.M)\ N\ x & \stackrel{\mathit{def}}{=} & \mathsf{if}\ i \in FV\ N \wedge x \neq i \\
 & & \mathsf{then}\ x \in FV\ M \\
 & & \mathsf{else}\ \mathit{Traps}\ M\ N\ x \\
\mathit{Traps}\ \mathit{zero}\ N\ x & \stackrel{\mathit{def}}{=} & \mathsf{False} \\
\mathit{Traps}\ (\mathit{succ}\ M)\ N\ x & \stackrel{\mathit{def}}{=} & \mathit{Traps}\ M\ N\ x \\
\mathit{Traps}\ (\mathit{pred}\ M)\ N\ x & \stackrel{\mathit{def}}{=} & \mathit{Traps}\ M\ N\ x \\
\mathit{Traps}\ (\mathit{ifz}\ M_1\ M_2\ M_3)\ N\ x & \stackrel{\mathit{def}}{=} & (\mathit{Traps}\ M_1\ N\ x) \vee \\
 & & (\mathit{Traps}\ M_2\ N\ x) \vee \\
 & & (\mathit{Traps}\ M_3\ N\ x) \\
\mathit{Traps}\ \mathit{unity}\ N\ x & \stackrel{\mathit{def}}{=} & \mathsf{False} \\
\mathit{Traps}\ (\mathit{inL}_T\ M)\ N\ x & \stackrel{\mathit{def}}{=} & \mathit{Traps}\ M\ N\ x \\
\mathit{Traps}\ (\mathit{inR}_T\ M)\ N\ x & \stackrel{\mathit{def}}{=} & \mathit{Traps}\ M\ N\ x \\
\mathit{Traps}\ (\langle M_1 \Leftarrow y \rangle L \langle z \Rightarrow M_2 \rangle)\ N\ x & \stackrel{\mathit{def}}{=} & (y \neq x \wedge y \in FV\ N \wedge x \in FV\ M_1) \vee \\
 & & (z \neq x \wedge z \in FV\ N \wedge x \in FV\ M_2) \vee \\
 & & (\mathit{Traps}\ L\ N\ x)
\end{array}
$$

References

Z.M. Ariola and S. Blom. *Lambda Calculi Plus Letrec*. Vrije Universiteit, Faculteit der Wiskunde en Informatica, 1997.

HP Barendregt. *The Lambda Calculus, its Syntax and Semantics*. North-Holland, 1984.

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

S. Blazy, Z. Dargaye, and X. Leroy. Formal Verification of a C Compiler Front-end. *Symp. on Formal Methods*, pages 460–475, 2006.

M.L. Cornélio. *Refactorings as Formal Refinements*. PhD thesis, Universidade Federal de Pernambuco, 2004.

N.G. de Bruijn. 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, 1972.

Ran Ettinger. *Refactoring via Program Slicing and Sliding*. PhD thesis, Oxford University Computing Laboratory, June 2007.

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

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

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

MJC Gordon and TF Melham. *Introduction to HOL: a theorem proving environment for higher order logic*. Cambridge University Press New York, NY, USA, 1993.

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

S.P. Jones et al. *Haskell 98 language and libraries*. Cambridge University Press, 2003.

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

X. Leroy. Formal certification of a compiler back-end or: programming a compiler with a proof assistant. *ACM SIGPLAN Notices*, 41(1):42–54, 2006.

Huiqing Li. *Refactoring Haskell Programs*. PhD thesis, Computing Laboratory, University of Kent, September 2006.

Huiqing Li and Simon Thompson. Tool support for refactoring functional programs. In *ACM SIGPLAN Workshop on Partial Evaluation and Program Manipulation*, January 2008.

Huiqing Li, Simon Thompson, László Lövei, Zoltán Horváth, Tamás Kozsik, Anikó Víg, and Tamás Nagy. Refactoring erlang programs. In *The Proceedings of 12th International Erlang/OTP User Conference*, Stockholm, Sweden, November 2006. URL `http://www.cs.kent.ac.uk/pubs/2006/2455`.

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

Tobias Nipkow, Lawrence C. Paulson, and Markus Wenzel. *Isabelle/HOL: A Proof Assistant for Higher-Order Logic*, volume 2283 of *LNCS*. Springer, 2002.

Michael Norrish. *C formalised in HOL*. PhD thesis, Computer Laboratory, University of Cambridge, 1998.

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

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

Peter Sewell, Francesco Zappa Nardelli, Scott Owens, Gilles Peskine, Thomas Ridge, Susmit Sarkar, and Rok Strniša. Ott: Effective Tool Support for the Working Semanticist. 2007. To appear.

Nik Sultana. Verification of refactorings in Isabelle/HOL. Master's thesis, University of Kent, 2007.

C. Urban and C. Tasson. Nominal techniques in Isabelle/HOL. *CADE-20*, 3632, 2005.

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