Chapter 11

A Certified Refactoring Engine

Nik Sultana[1], Simon Thompson[2]
Category: Research

Abstract: The paper surveys how software tools such as refactoring systems can be validated, and introduces a new mechanism, namely the generation of a refactoring engine for a functional programming language from an Isabelle/HOL theory in which it is verified. This research is a first step in a programme to construct certified programming tools from verified theories. We also provide some empirical evidence of how refactoring can be of significant benefit in reshaping automatically-generated program code for use in larger systems.

11.1 INTRODUCTION

Refactoring is the process of improving the structure of program code and it has been argued [9] that refactoring is crucial to enable code reuse by facilitating its adaptation and comprehension, and therefore lessens the cost of development.

There is a spectrum of interpretations of what constitutes a refactoring, but in this paper we limit our attention to behaviour-preserving program transformations. A refactoring *engine* is a computer implementation of a set of refactorings, and various methods for checking the *correctness* of refactoring engines – that is, that they preserve the behaviour of arbitrary programs – have been studied. These methods include testing the refactored code [7, Chapter 4], testing the refactoring engines [15, 4], as well as formally proving properties about their specifications [3, 6, 8, 11, 22]. In some of the works cited in the last category the specifications used were themselves executable, and therefore could also be interpreted as

[1]Mathematical Institute, University of Munich, Germany;
nik.sultana@yahoo.com
[2]Computing Lab, University of Kent, United Kingdom;
s.j.thompson@kent.ac.uk

implementations of the refactoring engines.
In this work we use HOL as a specification language; it is not entirely executable so initial work on this project consisted in surveying the juncture between executable programs and HOL in the literature. Of particular interest is the juncture between Haskell and HOL, since we aspire to eventually extend the Haskell Refactorer HaRe in this way. The approaches can be classified into two:

- The first possibility of integrating an executable definition of a refactoring and Isabelle/HOL involved translating the former into a definition embedded in the latter. This is the approach taken, for example, in the verification of an L4 microkernel [5] and in the system Hets [25].
- The second option was to generate Haskell code from the Isabelle theory in which the refactoring is verified. This could be done using the Hets system since it supports the generation of Haskell code from HasCASL specifications [16], but we instead used a code generation framework that relied on an executable subset of HOL.

In previous work [22] a number of correctness theorems about refactorings were described. This built on previous work [14] to formalise refactorings over functional programs. It explored using untyped and typed λ-calculi and developed fully-formal proofs that were checked using Isabelle/HOL [17].

The work described in this paper uses a new code generation framework for Isabelle and extends previous work to produce Haskell code for one of the refactorings studied. Software produced in this manner – that is, generated from a machine-checked theory proving its correctness – is said to be *certified*; other examples using this approach are described in §11.5.

This paper is a study of the steps involved in the process of generating a refactoring engine from its verification in an Isabelle/HOL theory and will briefly summarise the work it builds on. We also argue that refactoring can be profitably applied to automatically-generated code with reference to particular examples. The contributions of this paper are:

- An extension of the verification of refactorings to produce verified code;
- A discussion of patterns of refactoring that are applicable to the verified code itself;
- A discussion of the interaction between the refactoring engine and the typechecker, and on the integration of this code with a refactoring tool.

Although we restrict our focus to refactorings, these steps may be followed to produce various other kinds of certified software.

The rest of the paper is organised as follows. The next section describes the approach we use to obtain Haskell programs from an Isabelle theory. Previous work on the verification of refactorings is summarised in §11.3, and in §11.4 we describe the extension of that work to obtain a verified refactoring. Related work to produce certified programmer tools is described in §11.5 and the paper concludes with a discussion.

11.2 ISABELLE

The Isabelle proof assistant is designed to facilitate the embedding of object logics in which to reason. It provides a metalogic consisting of constructive higher-order logic [19]; the work we describe is formalised in HOL [17], a classical higher-order logic embedded in Isabelle.

11.2.1 Program generation for Isabelle

Specifications are generally considered more perspicuous than programs due to their lack of operational details; these details may be considered a distraction at a high level of abstraction. Having written and validated a specification, one might wish to animate it for various reasons: for example, one could "test" the specification, or else generate the implementation directly from its specification.

Since specifications are often grounded in a logic, the relationship between logic and programs has been exploited not only for studying the latter using the former, but also for yielding programs from proofs of their properties.

In this work we use a framework [10] for generating program code from Isabelle theories. This framework relies on restricting definitions to an executable subset and exploits equational theorems to yield defining equations for functions, which are processed to eventually yield code. It also yields definitions for algebraic types and type classes, and can be instructed to target different languages; if the target language does not support type classes natively then a dictionary translation is used.

11.3 VERIFYING A REFACTORING

In this section we summarise the results on the verification of refactorings which first appeared in [22]. A refactoring consists of two parts: a program transformation T, and a collection of side-conditions (or pre-conditions) for the transformation in question to be meaning preserving; in other words, the transformation is only performed if the side-conditions are satisfied. The preconditions are conjoined to form the formula Q.

Note that in general a program transformation will be parametrised by a number of other arguments, such as an old and new name for an object; where appropriate in what follows we will suppress these other arguments by $\vec{x}$. The metavariable p ranges over programs.

Definition 11.1. *For a particular Q and T, and modulo $\vec{x}$, the* behaviour *of a refactoring is described by the following function* $\lambda p.\ \mathsf{if}\ (Q\,p)\ \mathsf{then}\ (T\,p)\ \mathsf{else}\ p$

The symbol $\simeq$ will be used to denote a behavioural equivalence over programs.

Definition 11.2. *A refactoring is* correct *iff it is behaviour-preserving, that is it satisfies the following formula* $\forall p.\ (Q\,p) \longrightarrow (T\,p) \simeq p$

In [22] this formulation was used to describe the verification of a number of refactorings in Isabelle/HOL. The principal challenge in carrying out such a verification consists of embedding the semantics of the programming language over which the refactoring is defined.

11.3.1 Program syntax and metalinguistic definitions

The initial investigation of the problem is carried out using a small language: namely PCF [21] extended with sum and unit types; we call this PCF^{+1}.

In what follows we use the following notational conventions: M, N, L range over terms, σ, τ range over types, and Γ ranges over typing contexts – formalised as finite maps. The notation $\Gamma, x:\tau$ abbreviates $(\Gamma_{|\mathrm{Dom}(\Gamma)-x})[x \mapsto \tau]$ – that is, adding a typing to a context will involve first restricting the context then carry out the extension. Symbol empty will denote the empty context.

Definition 11.3. *The* terms *of* PCF^{+1} *are inductively defined by the following grammar:*

$$\begin{array}{rcllll} M & ::= & x & | & \lambda x^{\tau}.M \\ & | & M \cdot N & | & \mathit{fix}\, x^{\tau}.M \\ & | & \mathit{unity} & | & \mathit{zero} \\ & | & \mathit{succ}\, M & | & \mathit{pred}\, M \\ & | & \mathit{ifz}\, L\, M\, N & | & \mathit{inL}_{\tau}\, M \\ & | & \mathit{inR}_{\tau}\, M & | & \langle M \Leftarrow x \rangle L \langle y \Rightarrow N \rangle \end{array}$$

As per convention, *programs* are closed terms, i.e. terms with no free variables.

Definition 11.4. *The* types *of* PCF^{+1} *are inductively defined by the following grammar:*

$$\begin{array}{rcllll} \tau & ::= & \mathit{Nat} & | & \sigma \to \sigma' \\ & | & \mathit{Unit} & | & \sigma + \sigma' \end{array}$$

The notation $\Gamma \rhd M :: \tau$ asserts that term M is typed τ in Γ. Thus $\rhd M :: \tau$ asserts that M has type τ in the empty context, implying that M is a closed term.

A multi-sorted equational logic will be used to reason about programs in PCF^{+1}, in the style of [23, 24]. This will be the vehicle for proving refactorings correct for this language. The equivalence between terms M and N – both typed τ in Γ – will be expressed using $\Gamma \vdash M \simeq N :: \tau$. The relation $\simeq$ is a behavioural equivalence over terms, induced by the equational rules for PCF^{+1}. The language is defined together with the usual metalinguistic definitions:

- FV maps terms to sets containing their free variables.
- BV is analogous but concerns bound variables.

The formulation of some metalinguistic definitions is nonstandard with respect to the usual mathematical practice. When reasoning about programs in the abstract it is convenient to identify them up to renaming – however, names are of central importance in refactoring since the source-code returned to the programmer must be *recognisable*, and the default is that names need to be preserved. To this end

concrete names are used, and substitution is not defined to automatically rename variables – a *naïve* definition of substitution is used. As a result, the theory for this language is defined using a partial β-rule conditional upon non-capture – this condition is formulated in the following definition.

Definition 11.5. $\mathsf{Captures}\ M\ N \stackrel{def}{=} \exists v \in \mathsf{FV}\, N.\ (v \in \mathsf{BV}\, M)$

In order to avoid confusion we will distinguish the language levels using typefaces: terms in the language induced by Definition 11.3 will be shown in *italics*, the `monospace` typeface will be reserved for executable (meta)definitions and sans serif will be used for other (meta)definitions.

The predicate $\mathsf{Captures}\ M\ N$ was defined to formalise *M is free for all free variables in N*. In effect, Definition 11.5 formalises the Barendregt Variable convention [1, §2.1.13]. This predicate is used here for consistency with earlier work, but it is too imprecise for practical programming, and an improved, more precise, version was described in previous work [22, §5.1.2]. We conclude with two definitions by way of 'syntactic sugar'.

Definition 11.6.

$$\begin{aligned} \mathit{let}\ x^{\tau} := N\ \mathit{in}\ M &\stackrel{def}{=} (\lambda x^{\tau}.M) \cdot N \\ \mathit{letrec}\ x^{\tau} := N\ \mathit{in}\ M &\stackrel{def}{=} (\lambda x^{\tau}.M) \cdot (\mathit{fix}\, x^{\tau}.N) \end{aligned}$$

11.3.2 Metalinguistic results

We now turn to a specific refactoring – "enlarge definition type". This refactoring expands the type of a definition into the coproduct of its original type (on the left) and some other type. Its behaviour is illustrated below using Haskell-like pseudocode fragments. Note that the refactoring replaces *every* occurrence of `(f x)` with `(either f L x)`.

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

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

The symbol ⇆ in the above snippet is intended to suggest the bidirectionality of refactoring: if two programs are indeed equivalent than both transforming one to the other and the inverse are behaviour-preserving.

We have modified the formulation of correctness of this refactoring as expressed previously [22] by weakening its assumptions on the typing context. The correctness of this refactoring is given by Theorem 11.7; note that all the variables are implicitly universally quantified. The theorem's formulation instantiates the general form from Definition 11.2.

Theorem 11.7. *(Enlarge definition type)*

$$\Gamma \vdash let\ x^{\tau} := M\ in N \simeq let\ x^{\tau+\tau'} := (inL_{\tau+\tau'}\, M)\ in N[\langle x' \Leftarrow x' \rangle x \langle y \Rightarrow L \rangle / x] :: \sigma$$

The above equation holds provided that these side-conditions are satisfied:

1. *Well-typing:*
   (a) $\Gamma, x : \tau \rhd N :: \sigma$
   (b) $\Gamma, x : \tau, y : \tau' \rhd L :: \tau$
   (c) $\Gamma \rhd M :: \tau$
2. *Non-capture:*
   (a) $\neg \mathsf{Captures}\, N\, \langle x' \Leftarrow x' \rangle x \langle y \Rightarrow L \rangle$
   (b) $\neg \mathsf{Captures}\, N\, M$
   (c) $\neg \mathsf{Captures}\, L\, M$
3. *Non-occurrence:*
   (a) $x' \notin \mathsf{FV}\, M$
   (b) $y \notin \mathsf{FV}\, M$
   (c) $x \notin \mathsf{FV}\, L$

Proving this theorem relied on the "substitution lemma" that asserts that substitution preserves typing:

Lemma 11.8. Substitution lemma

$$\Gamma \rhd N :: \sigma \wedge \Gamma \rhd x :: \tau \wedge \neg \mathsf{Captures}\, N\, L \wedge \Gamma, x : \tau' \rhd L :: \tau \longrightarrow \Gamma, x : \tau' \rhd N[L/x] :: \sigma$$

This concludes the summary of relevant previous results.

11.4 GENERATING THE REFACTORING ENGINE

This section describes how the previous result was extended in order to generate correct code implementing the refactoring described in the previous section. The formulation of the refactoring's correctness showed how the refactoring behaved, but it was not an effective definition – and it relied on other non-effective definitions. It was necessary to derive a program implementing this behaviour; our approach consisted of complementing logical definitions – for instance, the predicate Captures (Definition 11.5) – with their algorithmic refinements and proving them to be equivalent. The refactoring was then defined effectively using these algorithmic definitions and proved correct – the proof appealed to Theorem 11.7. An improved version of the algorithm was subsequently written and proved correct. The following sections elaborate on each step of the process.

11.4.1 Effective refinements to logical definitions

The formulation of Theorem 11.7 indicates the shape of programs over which the refactoring's transformation is defined – for the rest of the programs it behaves like the identity function, as can be seen from Definition 11.1. Despite the computational hints provided in Theorem 11.7 it cannot be executed directly or yield a program. A program implementing the behaviour specified by this theorem would behave as follows:

- It must first recognise the program's shape for which the transformation is defined. This is achieved by pattern matching on the input program.
- The propositions appearing before the implication in Theorem 11.7 need to be checked in some order, and therefore
- An algorithm for each proposition needs to be invoked.
- The program transformation itself needs to be implemented. This is straightforward since the transformation merely rearranges the arguments given to the transformation around a new form of expression. The transformation constructs the expression on the right hand side of the consequent in Theorem 11.7. A refactoring is a *meta*-program and therefore can use the substitution operation – this operation is *implicit* and not part of the actual program.

11.4.2 Changing logical definitions into effective ones

Inspecting the side-conditions of the "enlarge definition type" refactoring – that is, the antecedents in Theorem 11.7 – reveals that they can be classified into the following three categories:

- *Well-typing checks*, for instance $\Gamma, x : \tau \rhd N :: \sigma$
- *(non)Capture checks*, for instance $\neg \mathsf{Captures}\, N\, M$
- *Free occurrence checks*, for instance $x' \notin \mathsf{FV}\, M$

Each of these predicates must be refined into an effective characteristic function. For the first category of checks this involves implementing a type-reconstruction algorithm for PCF^{+1} and proving it to be correct relative to the static semantics. Handling the second category of checks is easier since the predicate Captures is simpler and therefore its algorithm is easier to verify. Nothing needs to be done for the third category since the code generation framework can yield code for of FV and the set-(non)membership test it depends on thanks to internal preprocessing instructions the framework uses for HOL theories.

The algorithmic equivalent to Captures (Definition 11.5) is examined next. Logical notation is used for the effective Boolean operations here.

Definition 11.9. *The algorithm* `CapturesEff` *is defined thus:*

$$
\begin{array}{lcl}
\texttt{CapturesEff} & :: & \mathsf{Terms} \to \mathsf{Terms} \to \mathsf{bool} \\
\texttt{CapturesEff}\, x\, M' & = & \mathsf{False} \\
\texttt{CapturesEff}\, (\lambda x^{\tau}.M)\, M' & = & ((x \in \mathsf{FV} M') \vee (\texttt{CapturesEff}\, M\, M')) \\
\texttt{CapturesEff}\, (M_1 \cdot M_2)\, M' & = & ((\texttt{CapturesEff}\, M_1\, M') \vee \\
 & & (\texttt{CapturesEff}\, M_2\, M'))
\end{array}
$$

The other clauses are defined by structural induction in the obvious way.

Lemma 11.10. *The executable definition* `CapturesEff` *is equivalent to the logical predicate* Captures*:* $\forall M\,N.\ \mathsf{Captures}\ M\ N \longleftrightarrow \mathtt{CapturesEff}\ M\ N$

Proof sketch Induction on M. □

We can now replace every occurrence of $\mathsf{Captures}\,M\,N$ with $\mathtt{CapturesEff}\,M\,N$ – such as in Theorem 11.7. This would not lead to any notable benefit however, since even if all the side-condition checks were defined effectively we still would not be able to produce the code of the full refactoring. We will proceed with the original plan: using Theorem 11.7 as a blueprint and building an effective definition for the refactoring according to it.

The next step involves providing an algorithmic equivalent to the relation asserting that a term is well-typed. Given a term in the language induced by Definitions 11.3 and 11.4 and a typing context, the algorithm is to decide whether the term is typable – i.e., we require an algorithm solving the *type reconstruction* problem. The signature for this definition is given next; the actual implementation is omitted here.

Definition 11.11. Type reconstruction

$$\mathtt{typeInfer} :: \mathsf{Terms} \to \mathsf{Contexts} \rightharpoonup \mathsf{Types}$$

In terms of Haskell types, `typeInfer` returns values of type `Maybe Type` . Having defined the type reconstruction algorithm it is proved to be correct next.

Lemma 11.12. Correct Type Reconstruction

$$\forall \Gamma\,M\,\tau.\ (\Gamma \rhd M :: \tau) \longleftrightarrow (\mathtt{typeInfer}\,M\,\Gamma = \mathsf{Just}\ \tau)$$

Proof sketch ($\Rightarrow$) Straightforward induction on the derivation. ($\Leftarrow$) Induction on M; the typing rule is used in the proof of each case. In non-trivial cases the proof involves case analysis on `typeInfer` and using "inversion lemmata" concerning `typeInfer`.

An example inversion lemma, concerning application, is given below:

$$\forall \tau\,\Gamma.\exists\sigma.(\mathtt{typeInfer}\,(M_1 \cdot M_2)\,\Gamma = \mathsf{Just}\,\tau) \longrightarrow$$
$$(\mathtt{typeInfer}\,M_1\,\Gamma = \mathsf{Just}\,(\sigma \to \tau) \wedge \mathtt{typeInfer}\,M_2\,\Gamma = (\mathsf{Just}\,\sigma))$$

These lemmata are analogues to the inversion lemmata for the typing relation, but instead concern the type reconstruction algorithm. These lemmata are proved by case analysis on each occurrence of `typeInfer` in the consequent. □

Since the algorithms have been proved to be equivalent to the specifications we can use them interchangeably in the specification of theorems. More profitably, we could employ the algorithmic definitions in building the refactoring; this will be described next.

Definition 11.13. *An algorithm implementing the* enlarge definition type *refactoring:*

$$
\begin{array}{lcl}
\mathtt{R} & :: & \mathsf{Term} \to \mathsf{Var} \to \mathsf{Var} \to \mathsf{Type} \to \mathsf{Term} \to \mathsf{Term} \\
\mathtt{R}\ p@(\mathit{let}\ x^{\tau} := M\ \mathit{in} N) & \stackrel{def}{=} & \lambda x'\, y\, T'\, L. \\
 & & \quad \mathsf{case}\ \ (\mathtt{typeInfer}\, L\, \mathsf{empty}, x:\tau, y:\tau')\ \mathsf{of} \\
 & & \qquad \mathsf{Nothing} \Rightarrow p \\
 & & \qquad \mathsf{Just}\ \tau'' \Rightarrow \\
 & & \qquad\ \ \mathsf{if}\ \mathsf{not}(Q\,\mathsf{and}\,(\tau = \tau''\,)) \\
 & & \qquad\ \ \mathsf{then}\ p \\
 & & \qquad\ \ \mathsf{else}\ p' \\
 & & \quad \mathsf{where} \\
 & & \quad \mathtt{Q} \stackrel{def}{=} \mathtt{CapturesEff}N(\langle x' \Leftarrow x' \rangle x \langle y \Rightarrow L \rangle)) \\
 & & \qquad \mathsf{and}\ \ (\neg \mathtt{CapturesEff}\, N\, M) \\
 & & \qquad \mathsf{and}\ \ (\neg \mathtt{CapturesEff}\, L\, M) \\
 & & \qquad \mathsf{and}\ \ (\mathtt{notIn}\, x'\, (\mathtt{FV}\, M)) \\
 & & \qquad \mathsf{and}\ \ (\mathtt{notIn}\, y\, (\mathtt{FV}\, M)) \\
 & & \qquad \mathsf{and}\ \ (\mathtt{notIn}\, x\, (\mathtt{FV}\, L)) \\
 & & \quad p' \stackrel{def}{=} \mathit{let}\ x^{\tau+\tau'} := (\mathit{in} L_{\tau+\tau'}\, M)\, \mathit{in} N[\langle x' \Leftarrow x' \rangle x \langle y \Rightarrow L \rangle / x] \\
\mathtt{R}\ p & \stackrel{def}{=} & p
\end{array}
$$

11.4.3 Obtaining an algorithm for the refactoring

The algorithm was defined by reading-off the intended behaviour from Theorem 11.7 and filling in the practical details. The algorithm is shown in Definition 11.13 – note that this does not show the Haskell code produced, but a stylised simplification: for instance, in the interest of clarity we perform pattern matching on the abbreviation rather than on the core terms of PCF^{+1}. Note that the side-conditions are elided under the local definition `Q`, and that the last clause of the definition specifies that `R` behaves like the identity function when control "falls through" because of an unsuccessful match in the previous line.

Apart from a program the algorithm is parametrised by two variables, a type and a term: as can be seen from the right hand side of the equation in the antecedent of Theorem 11.7 these parameters are used to build the transformed program.

In line with previous usage, the symbol $\stackrel{\text{def}}{=}$ is used to convey a definition and the symbol $=$ will represent the equality test carried out in the metalanguage. In a Haskell implementation these would be represented by $=$ and $==$ respectively.

11.4.4 Verifying the algorithm

The refactoring given in Definition 11.13 is *correct* iff it preserves the behaviour of arbitrary programs. This statement must indeed be weakened to the following: a refactoring is correct iff it preserves the behaviour of arbitrary *well-typed*

programs. This formulation is sensible since maltyped programs are considered to be meaningless, and is necessary since assuming the program to be well-typed discharges the related preconditions in the refactoring – see Theorem 11.7. This will be discussed further below.

In the algorithm given in Definition 11.13 we invoke the type checker on argument L and later confirm that L is indeed well-typed and has the expected type. Note however that this formulation assumes L to be a closed term – this assumption will be weakened in the improvement of the algorithm presented below. Since L, M and N shared the same typing context, this leads us to consider the argument to the refactoring – i.e., part of a program – as a closed term: a program. This is a strong assumption, and renders the refactoring inapplicable to open subterms of programs having the right shape. Weakening this formulation requires redefining the refactoring algorithm, so we postpone the generalisation and first seek to verify Definition 11.13 using the appropriate formulation.

During the main proof we will need to use the following lemma; it simply asserts that any well-typed program is behaviourally-equivalent to itself.

Lemma 11.14.

$$\forall p\,\tau.\ (\texttt{typeInfer}\ p\ \mathsf{empty} = \mathsf{Just}\ \tau) \longrightarrow \mathsf{empty} \vdash p \simeq p :: \tau$$

Proof sketch Case analysis on "`typeInfer` p empty" followed by appealing to the reflexivity of $\simeq$ and Theorem 11.12. □

We now proceed to formulating the correctness of the algorithm. For clarity we elide the arguments x, y, τ', L into $\vec{x}$ in the formulation of the algorithm's correctness given below.

Theorem 11.15. *Correctness of the refactoring* for closed terms.

$$\forall p\,\tau.\ (\texttt{typeInfer}\ p\ \mathsf{empty} = \mathsf{Just}\ \tau) \longrightarrow \forall\ \vec{x}\ .\ \mathsf{empty} \vdash\ (\mathtt{R}\ p\ \vec{x}) \simeq p :: \tau$$

Proof sketch We seek to show that for any well-typed input, `R` returns a program equivalent to the original program – and thus in any case the theorem should rest on Lemma 11.14 if `R` behaves in this manner.

We expand `R` to obtain its conditional checks (in the local definition `Q` in `R`) as preconditions and transform the consequent into that of Theorem 11.7.

We then show that the assumptions obtained by expanding `R` imply the antecedents in Theorem 11.7: this is straightforward by appealing to the logical equivalence of Captures and `CapturesEff`, and the membership tests are equivalent.

Finally we must show that the type-related preconditions of Theorem 11.7 are also implied. This involves showing that if p is well-typed then M, N and L are also typed as expected. The inversion lemmata of the type system of PCF^{+1} are used to transform the goals and then Lemma 11.12 is used to show that the type-related preconditions of Theorem 11.7 are implied by the result returned by the type reconstruction algorithm. □

The result we have just proved makes a strong assumption about L – that it is a closed term – but it has served to highlight the general approach we should take when verifying such an algorithm. Equally importantly, it explained more clearly the interconnection between a type-checker and the refactoring tool in the setting of a typed language.

11.4.5 Generalising the algorithm and correctness proof

The previous section revealed that Definition 11.13 is not sufficiently general: it can only work on closed sub-programs. In order to generalise this result, and thus render the algorithm applicable over arbitrary subprograms of well-typed programs, we need to pass typing information to the algorithm as a parameter. The new definition differs from Definition 11.13 only in the addition of a typing context as a formal parameter and the replacement of empty in the case..of using this context.

11.4.6 Code generation

Once the algorithm was proved to be correct, the code generation framework by Haftmann and Nipkow [10] – described in §11.2.1 – was used to generate Haskell code from the algorithm's definition in Isabelle/HOL.

Four Haskell modules were generated by the framework – corresponding to the Isabelle theories from which the code was generated. Apart from the code most closely associated with the synthesised refactoring engine, other code was generated on which the implementation depended – for instance code related to natural numbers, sets and HOL itself. In total 313 lines of Haskell code were generated, and could be immediately compiled under GHC 6.4.1.

11.4.7 Improving generated code

When the generated Haskell code was studied it was unsurprising that the generated code was very similar to the definitions in the theories it was generated from; but the redundancy and illegibility of some parts of the code immediately suggested the opportunity to refactor the generated code.

In refactoring jargon, patterns of code such as these are suggestively called "bad smells" – or opportunities for refactoring. This suggested the value of applying refactoring to code generated in this manner. Some examples are outlined next:

Removing dead code This code might be redundant local definitions within definitions or else replacing dummy variables in code produced with anonymous placeholders (_ in Haskell) if they are supported by the target language.

Type synonyms Some types reoccur in the program and it is well worth giving them meaningful names to distinguish them. The readability of type signatures could be improved by adding type synonyms to name frequently-used signatures.

Moving code This would facilitate reordering the definitions in the generated code, moving code between existing modules or else into new modules to better reflect the relationships between definitions.

Layout style The generated code uses the coarser style of Haskell programming, but the user could be offered the choice of which style to use through the use of a refactoring tool.

11.4.8 Summary

This section described this paper's contribution and the steps needed to bridge the refactoring's correctness theorem with generating its program code.

Since the generation of code from specifications is not fully automatic, executable versions of logical definitions needed to be written and proved equivalent. In particular, this had to be done for predicates appearing in the refactoring's side-conditions. Another algorithmic definition was then given for the whole refactoring, which in turn called the previously-defined algorithms.

This algorithm was verified using the theorem proved in earlier work, and Haskell code for the refactoring – and the other definitions it relies on – was generated using Isabelle's new code generation framework.

It was then observed that refactoring could be useful for managing automatically-synthesised code, and a number of suggestions were made for refactoring the code we generated.

11.5 RELATED WORK

The intention to build refactoring tools from the verification of refactorings was also expressed in earlier work by Garrido & Meseguer [8] and Junior et al. [11], using the systems Maude and CafeOBJ respectively.

Using interactive theorem provers to build certified programming tools has been attempted for different tools and using different systems. For example Okuma & Minamide [18] use Isabelle/HOL to specify and verify a compiler, of which code is then generated and embedded into a larger system that compiles a small functional language into Java bytecode. A larger development is described by Blazy et al. [2] and Leroy [12], in which a compiler is certified – its frontend compiles a fragment of C into an intermediate language called Cminor, and the backend completes the compilation into PowerPC's assembly language.

11.6 DISCUSSION

Using PCF^{+1} to study the synthesis of refactoring engines in this manner was perhaps a good starting point not only because the complexity of the language does not eclipse refactoring as the scope of the research, but also because when it came to synthesising the refactoring engine the code produced was small and more amenable to analysis than had we used a larger language.

The simplicity of the language may be adequate for initial study but is indeed far removed from realistic programming languages. Addressing a "toy" language in this experiment has helped prepare us better for tackling a larger language – this work has provided insight on the entire span from formalising the programming language to producing a certified refactoring engine for that language.

In the long term this line of research seeks to study the feasibility of producing tools using an approach similar to this – by relying on a dialogue with a theorem prover to ensure correctness of the synthesised tool. At the very least this would be as difficult as formalising a "non-toy" programming language, but the steady activity in this area of research lends hope to a rapid realisation of such developments – in part animated by increasing interest devoted to mechanical theorem proving.

The type-based refactoring engine studied here invokes the type reconstruction program as part of its operation and, as observed earlier, a positive result from the type reconstruction program is necessary before invoking the refactoring engine. It might be fruitful to explore if the correspondence between these tools could be rearranged more optimally, and studying the interaction between programming and refactoring tools in order to explore how they can cooperate better.

This paper extends previous work [22], partly summarised in §11.3, with the results described in §11.4. The extension involved constructing effective equivalents to logical predicates used in earlier specification of refactorings, building an executable specification of a refactoring engine that refined the specification expressed in Theorem 11.7, and proving it to be correct by Theorem 11.15. This was carried out in part to explore the phases of the process, but a second look at this process might instead seek to study to what extent parts of the process could be automated following the validation of a specification. The amount of work needed to extend the Isabelle theory to generate code was only a few days, but this increased the size of the theory file by more than 50%; it would be desirable to study the optimisation of this process to reduce manual proving and keep the size of the formalisation small. This would seek to produce a system that given a formula such as Theorem 11.7 it would greatly hasten arriving at Definition 11.13. This system could perhaps be modified to yield *two* programs – a refactoring in either direction – due to the bidirectionality of refactoring, described in §11.3.2.

Refactoring is usually done on program code written by people, but while studying the synthesis of refactoring engines we were presented with an opportunity to reflect on refactoring code that has been generated by machine. Such code bears the artefacts of the definitions from which it has been generated, and of the generation process itself, but is bare of comments; perhaps future work on code generation could adapt heuristics such as those used by Li [13, p.65] to extract comments deemed associated with definitions.

11.6.1 Future work and Conclusions

Apart from the ideas discussed in the previous section, there are various other proposals for extensions of this work. Central among these proposals is the pro-

duction of usable programming tools using these methods, or extending existing tools with new functionality. This might be realised by integrating a refactoring engine obtained in this way with HaRe [13] – the Haskell Refactorer.

Another direction for future work involves addressing a more expressive language – for instance one with ML-style polymorphism [20, §22.7]. This would inch us closer towards a realistic programming language and enable the synthesis of more useful refactoring engines.

We could expand horizontally by broadening the scope of the mechanisation *beyond* refactoring, to include other programming tools and studying their interaction. Alternatively, we could turn our attention to studying other refactorings over this language to test this method further.

Programming tools need to be usable and correct, and this paper describes research into the latter. This work is concerned with the synthesis of a correct refactoring engine through the use of the proof assistant Isabelle. It extends earlier work [22] and elaborates on the process of producing a certified refactoring engine, making use of Isabelle's new code generation framework. In the course of this research we observed that refactoring can be profitably applied to code generated from mechanised theories.

We thank Huiqing Li for valuable discussions on the topic of this paper and the anonymous referees for helpful feedback. The first author acknowledges the support of a Marie Curie EST fellowship, the second author acknowledges the support of the EPSRC for building the HaRe and Wrangler tools. Work on earlier related research was possible thanks to financial support provided to the first author by the Computing Lab at the University of Kent and by the Malta Government Scholarship Scheme through award MGSS/2006/007.

REFERENCES

[1] H. Barendregt. *The Lambda Calculus, its Syntax and Semantics*. North-Holland, 1984.

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

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

[4] B. Daniel, D. Dig, K. Garcia, and D. Marinov. Automated testing of refactoring engines. In *Proceedings of Foundations of Software Engineering (FSE'07)*, Dubrovnik, Croatia, Sep 2007.

[5] K. Elphinstone, G. Klein, and R. Kolanski. Formalising a high-performance microkernel. In R. Leino, editor, *Workshop on Verified Software: Theories, Tools, and Experiments (VSTTE 06)*, Microsoft Research Technical Report MSR-TR-2006-117, pages 1–7, Seattle, USA, Aug. 2006.

[6] R. Ettinger. *Refactoring via Program Slicing and Sliding*. PhD thesis, Oxford University Computing Laboratory, June 2007.

[7] M. Fowler and K. Beck. *Refactoring: Improving the Design of Existing Code*. Addison-Wesley Professional, 1999.

[8] 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.

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

[10] F. Haftmann and T. Nipkow. A code generator framework for Isabelle/HOL. Technical Report 364/07, Department of Computer Science, University of Kaiserslautern, 08 2007.

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

[12] 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.

[13] H. Li. *Refactoring Haskell Programs*. PhD thesis, Computing Laboratory, University of Kent, September 2006.

[14] H. Li and S. Thompson. Formalisation of Haskell Refactorings. In M. van Eekelen and K. Hammond, editors, *Trends in Functional Programming*, September 2005.

[15] H. Li and S. Thompson. Testing Erlang Refactorings with QuickCheck. In *Draft Proceedings of the 19th International Symposium on Implementation and Application of Functional Languages, IFL 2007*, Freiburg, Germany, Sep 2007.

[16] T. Mossakowski, C. Maeder, and K. Luttich. The Heterogeneous Tool Set, HETS. *LECTURE NOTES IN COMPUTER SCIENCE*, 4424:519, 2007.

[17] T. Nipkow, L. C. Paulson, and M. Wenzel. *Isabelle/HOL: A Proof Assistant for Higher-Order Logic*, volume 2283 of *LNCS*. Springer, 2002.

[18] K. Okuma and Y. Minamide. Executing Verified Compiler Specification. *Programming Languages and Systems: First Asian Symposium, APLAS 2003, Beijing, China, November 27-29, 2003: Proceedings*, 2003.

[19] L. C. Paulson. *Isabelle: A Generic Theorem Prover*, volume 828 of *Lecture Notes in Computer Science*. Springer, 1994.

[20] B. Pierce. *Types and Programming Languages*. MIT Press, 2002.

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

[22] N. Sultana and S. Thompson. Mechanical Verification of Refactorings. In *Workshop on Partial Evaluation and Program Manipulation*. ACM SIGPLAN, January 2008.

[23] S. Thompson. Formulating Haskell. Technical Report 29-92*, University of Kent, Computing Laboratory, University of Kent, Canterbury, UK, November 1992.

[24] S. Thompson. A Logic for Miranda, Revisited. *Formal Aspects of Computing*, 7(7), March 1995.

[25] P. Torrini, C. Lueth, C. Maeder, and T. Mossakowski. Translating haskell to isabelle. Number 364/07, 08 2007.