Challenge Proposal: Verification of Refactorings *

Max Schäfer    Torbjörn Ekman    Oege de Moor
Programming Tools Group, University of Oxford, UK
{max.schaefer, torbjorn.ekman, oege.de.moor}@comlab.ox.ac.uk

Abstract
Automated refactoring tools are an essential part of a software developer’s toolbox. They are most useful for gradually improving large existing code bases and it is essential that they work reliably, since even a simple refactoring may affect many different parts of a program, and the programmer should not have to inspect every individual change to ensure that the transformation went as expected. Even extensively tested industrial-strength refactoring engines, however, are fraught with many bugs that lead to incorrect, non-behaviour preserving transformations. We argue that software refactoring tools are a prime candidate for mechanical verification, offering significant challenges but also the prospect of tangible benefits for real-world software development.

Categories and Subject Descriptors    D.3.4 [Programming Languages]: Processors

General Terms    Languages

Keywords    Refactoring, mechanical verification, proof assistants

1. Introduction
Software refactoring is the process of gradually improving the structure of existing software by behaviour-preserving transformations. Such individual transformations can be very complex, but the perhaps most widely used ones are simple operations like renaming variables or splitting up functions into smaller components to reduce their complexity; such operations can be automated and are available in most modern development environments.

Unfortunately, even such seemingly straightforward operations as renaming are very hard to implement correctly for modern programming languages. Especially object-oriented languages have complicated lookup rules that can interact in unexpected ways and make it very hard to effect a name change without introducing unwanted name capture.

We believe formalising and verifying refactoring tools is an interesting and novel challenge for language-based program verification. A refactoring tool has to carry out static semantic analysis and behaviour preserving transformations on a rich source language. This differs from traditional compilation, where the source language is transformed into a simplified intermediate representation on which most analyses are performed. In this paper, we describe some of the challenges arising from this setting, and survey and evaluate possible solutions.

Refactorings can certainly benefit from mechanised verification: Whereas it is easy to formulate and manually verify refactorings for small toy languages, more realistic languages with their complex syntax and semantics defeat pen-and-paper verification. This is especially critical when languages evolve: newly introduced constructs may affect existing refactorings in unexpected ways, and only careful formalisation and proof can help in discovering and addressing all corner cases.

The challenge, thus, is to implement a usable tool, or suite of tools, for automated software refactoring, that can perform non-trivial refactorings on programs written in a rich object language. It should be formally proved that the tool preserves certain properties of its input programs; full semantics preservation is desirable, but, as we will show, sometimes conflicts with usability considerations for realistic programming languages. Finally, the tool and its proof of correctness should be easily adaptable to work on an extended object language.

Refactoring tools have been implemented for many languages, but it seems unrealistic to attempt a direct verification of such off-the-shelf implementations, since a formal semantics of the object language is often lacking and the implementations are highly complex programs.

We propose to attack the problem by modularisation along two axes: The implementation of a refactoring should be split up into smaller components to be verified independently. As we shall show, the individual components often have more specific correctness properties that do not depend on the existence of a complete formal semantics of the object language. Likewise, the object language should not be viewed as a monolithic entity, but rather as a succession of extensions to a core language, with the refactoring first specified and verified for that core language and then modularly extended to encompass more and more features of the full language.

In the rest of the paper, we will flesh out this general proposal: Section 2 shows some of the challenges of correctly refactoring realistic programming languages, and discusses how a refactoring engine can be modularly decomposed on the concrete example of the Rename refactoring. Section 3 discusses some possible implementation languages and formalisms that support the desired modularisation, while Section 4 investigates the question of appropriate proof and verification techniques. These sections also briefly summarise our ongoing work on formalising and verifying refactorings in the theorem prover Coq and discuss the design choices we have made. We conclude in Section 5.

2. Software Refactorings
One of the simplest, yet most widely used refactorings is the Rename refactoring that allows the programmer to consistently change the name of a program entity, such as a variable or type, throughout the program without introducing unwanted name capture that could change program semantics. As a simple example, consider the Java program in Fig. 1.

Assume we want to rename the parameter a to x to better document its use as an initial value of the like named field. Since the renamed parameter shadows the field x, references to the field

\begin{figure}[h] 
\centering
\begin{lstlisting}[language=Java]
public class Example {
  // A
  public void doSomething() {
    // references to the field x are incorrect
    x = 10;
    // x is now available
    x = 20;
  }
}
\end{lstlisting}
\caption{Java program with a shadowing field.}
\end{figure}

\texttt{http://dx.doi.org/10.1145/1481848.1481859}
class Point {
  int x, y;
  public Point(int a, int b) {
    x = a;
    y = b;
  }
}

Figure 1. A simple Java class

class Point {
  int x, y;
  public Point(int x, int b) {
    this.x = x;
    y = b;
  }
}

Figure 2. A simple Java class, refactored

class A {
  void m() {
    final int x = 42;
    new Object() {
      int y = 23;
      { System.out.println(x); }
    }
  }
}

Figure 3. A tricky example

class A {
  public static void main(String[] args) {
    System.out.println(A.class.getName());
  }
}

Figure 4. A tricky example, wrongly refactored

class A {
  void m() {
    final int x = 42;
    new Object() {
      int y = 23;
      { System.out.println(y); }
    }
  }
}

Figure 5. Example of reflection

need to be qualified by this, so the refactored program should look like the one in Fig. 2.

More precisely, the refactoring engine needs to perform the following steps:

1. ensure that the renaming results in a correct Java program: we could not rename a to b, since a local variable of that name already exists
2. update the declaration of a to reflect the name change, and update every reference to a to refer to x instead
3. qualify every reference to a variable that is shadowed by the renaming

Roughly, these activities fall into the two categories of analysis and transformation: Analyses are needed to determine viability of the renaming and to find all references to the variable being renamed, but also to find all variables that will be shadowed, and to determine how (and if) they can be qualified to avoid name capture. The transformations perform the necessary changes on the basis of the information collected by the analyses.

As we have argued before, the analysis part is much harder than the transformation itself, and a frequent source of bugs even in industrial strength refactoring tools. For example, when renaming a local variable x to y, it seems safe to simply replace every simple name x in the variable’s scope by y in step 2 above. After all, Java does not allow lexical nesting of identically named local variables, so if the checks in step 1 passed, we can assume that no local variable y is declared in x’s scope that could capture a reference to the renamed variable.

Unfortunately, this heuristic fails to take local classes into account: In Fig. 2 we see an example of a local variable x with a field declared in its scope. Invoking method m on an object of dynamic type A will print the value of x, i.e. 42, to the console.

Although there are no immediate syntactic objections to renaming x to y that could be caught during the checks in step 1 (nesting fields and local variables of the same name is allowed in Java), the renaming cannot be performed, since the reference to x inside the anonymous class would be captured, and references to local variables cannot be qualified. Yet, many well-known IDEs fail to check for this possibility, and proceed with the refactoring, yielding the program in Fig. 3 where calling m now prints 23.

While extensive testing will certainly catch many bugs like this and eventually ensure that the refactoring engine behaves correctly on most programs it is likely to encounter, in a language the size of Java there are bound to be many corner cases that easily escape the attention of even the most conscientious analysis implementor. Only formal verification could give us complete assurance that we have covered every eventuality.

But proving correctness of a refactoring for a language like Java is a formidable undertaking. The most obvious correctness condition would be semantics preservation: If performing a refactoring (such as Rename) on a program P yields a program P′, the two programs P and P′ should be semantically equivalent. It is not clear, however, whether this is a viable criterion at all.

For one, there is no formal semantics of Java. Although the Java Language Specification contains a careful informal description of all aspects of the language, formalising this description would be a gargantuan task of doubtful value, since the resulting specification is likely to contain many mistakes.

On the other hand, there are some features of the Java language that make guaranteed behaviour preservation for refactorings unrealistic: Consider the trivial program in Fig. 5 where the main method makes use of the Java reflection capabilities to obtain a string representation of class A’s name, and print it to the console.

What should a refactoring engine do when asked to rename A to B? Just changing the name of the class would yield a program printing A instead of A, i.e. the transformation would not be behaviour preserving. On the other hand, it would certainly not be desirable to change the print statement to System.out.println("A");

Although this change does preserve behaviour here, it is much too intrusive, and in general it is clearly undecidable where and how such changes need to be performed to preserve semantic correctness. The situation is even worse with more dynamic languages, let alone scripting languages.
We thus propose to abandon the correctness criterion of behaviour preservation. While useful as a guideline to the implementor, it is not a realistic or desirable criterion for verification, except for small languages with a well-understood and formalised semantics.

Finding an equally general criterion as a substitute is part of our challenge. So far, all we can offer are more pragmatic correctness standards for individual refactorings. In the case of Renaming, for example, an obvious requirement is that the refactoring preserves a program’s binding structure; after the renaming, all names should still refer to the same declaration as before. This criterion is not sufficient to preserve behaviour in the presence of reflection, but it has the advantage of only depending on a small part of the language semantics (viz. name scoping), and seems like a fairly sensible compromise.

The individual analyses and transformations, in turn, have their own correctness criteria, which are usually quite simple and work together to guarantee correctness of the whole refactoring. As an example, consider the analysis needed in step 2 that finds all references to a given variable. Its correctness can be formalised with respect to a formalisation of Java name lookup, which is still a substantial undertaking, but much more manageable than formalising the whole language.

Another example is an auxiliary analysis in step 3 that we called access computation in [21]. Its aim is, for a given position $p$ in the program, to compute a (possibly qualified) name $n$ under which some declaration $d$ can be accessed. In the introductory example, access computation should determine that, after renaming the parameter, field $x$ can be accessed from the first line of the constructor as this.$x$, whereas field $y$ can simply be accessed as $y$. Again, this analysis has a very simple correctness criterion relative to lookup: Whenever access computation determines that $d$ can be accessed from $p$ as $n$, looking up $n$ from $p$ should really yield $d$, i.e. access computation should be a right inverse to lookup.

We thus see that a refactoring can usually be understood as being composed of several simpler analysis and transformation steps. From an implementation point of view, this decomposition is useful since many components can be reused in other refactorings (for example, almost all refactorings that move or duplicate code need to do a certain amount of name analysis), but most importantly decomposition aids verification, as the individual components have simple correctness criteria. It is thus important to find an implementation formalism that cleanly supports this modularisation and allows us to express analyses and transformations at a high level of abstraction that is suitable for verification.

3. Implementation Language

To implement analyses, attribute grammars [15] are, in our experience, a good choice. They hide the syntax tree traversal essential to analyses behind the simple concepts of synthesised and inherited attributes, which are functions on syntax tree nodes that are defined in terms of attributes on the child nodes and the sibling nodes, respectively. Two further concepts are important to increase the expressivity of this basic idea.

Reference attributes [10] are attributes that return nodes of the syntax tree itself as their result, on which further attributes can then be evaluated. The paradigmatic use case for this extension is name lookup: To look up a qualified name $a.b$ in Java, we need to look up $b$ as a member of the type of $a$; hence it makes sense to have a reference attribute $type$ that returns a reference to the node corresponding to the type of an expression, and an attribute $member$ that looks up a name on the returned node.

Circular attributes [7] are circularly defined attributes that can nevertheless be meaningfully evaluated by performing a fixed point iteration over a lattice. Such attributes are a powerful tool for succinctly expressing control and data flow analyses, which are essential for code moving refactorings: In order to extract part of a function into a new function, for example, we need to compute which local variables are live in the block to be extracted, and if any of them are updated, in order to determine which variables have to be passed to or returned from the extracted function.

The combined formalism of Circular Reference Attribute Grammars [18], known as CRAGs, form the basis of the JastAdd system, which has been used to implement the JastAddJ compiler [4], a full featured, standards compliant Java compiler, on top of which we have implemented a refactoring engine [21]. The success of this compiler provides a major argument in favour of using CRAGs to implement analyses, since it shows that despite their conceptual simplicity they have the power needed to tackle a real-world language.

One crucial feature of the JastAdd language is that both grammar descriptions and attribute definitions are open and can be modulated. In the case of JastAdd, there are some core modules implementing a compiler for Java 1.4, to which one can add a group of extension modules that define the new syntactic extensions introduced in Java 1.5 and provide additional attribute definitions to cover the new cases. With an eye towards verification, it would make sense to decompose the language even further, maybe starting from a core calculus as simple as MJ [1] or Jinja [13].

Another good candidate for implementing refactorings is JunGL [25], a multi-paradigm domain specific language for implementing refactorings. It provides an interesting mix of functional and logical features, in particular the concept of path queries, which are a concise and declarative way of expressing analyses. JunGL does not currently have any dedicated support for extensible specifications, though.

While attribute grammars and path queries excel at expressing analyses, they provide no explicit support for expressing transformations. For this part of the implementation, it might be interesting to turn to a rewriting based approach as championed, for example, in Maude [19] and Stratego [23]. The Strafunski system [22] shows that strategic rewriting can be embedded into a strongly typed functional programming language, which is also true of attribute grammars; it thus seems possible to combine both formalisms to take advantage of their complementary strengths. Strafunski also supports generic programming which would be helpful to support modular language extension.

Inspired by our previous positive experience with CRAGs, we chose them as the basis of our experiments with verifying analyses for refactorings in a theorem prover [20]. Attribute grammars translate quite naturally to functional programs with syntax trees represented by algebraic datatypes and attributes by functions. To model reference attributes, a concept of node identity is needed, for which Huet’s zipper [17] is well-suited. Circular attributes are more challenging to express in a total language like Coq, and we ultimately decided to explicitly encode the underlying fixed point computation.

Unfortunately it is not easily possible to automatically translate the existing JastAdd based refactoring implementation into Coq, since JastAdd allows attribute definitions to contain arbitrary Java code. However, almost all of JastAdd and the refactoring engine is written in a declarative style that is easily transcribed into a pure functional programming language by hand.

---

1 This situation has some parallels with Kniesel’s work on static composition of refactorings [14], but whereas their goal is to verify the composition of entire refactorings based on given pre- and postconditions, we are concerned with putting together components to form individual refactorings.
4. Verification and Proof

Once we have decomposed a refactoring implementation both in terms of its functionality and in terms of the object language, we need to find a suitable verification method to prove correctness of the components. If the correctness properties are simple enough, an intriguing possibility would be to use model checking techniques for automatic verification. Such an approach is taken by Estler et al. [6], who verify refactorings on the Z specification language. The scope of their approach, however, seems to be very limited, and it is probably not applicable to general purpose refactoring tools.

Inspiration can be drawn from the success of Rhodium [15], a domain specific language for declaratively specifying dataflow analyses that are verified by an automatic theorem prover without any need for human intervention. However, analyses implemented in Rhodium work on a simplified intermediate representation of programs, and it seems likely that in order to verify source-level analyses interactive theorem proving is a more appropriate choice.

The experience of the CompCert project [17, 2] shows that verification of a compiler for a simple but realistic, C-like language is indeed possible with a theorem prover. It is not clear that their techniques transfer to our setting, since many of the challenges in refactorings are due to the complexities of dealing with a rich source language.

Sultana and Thompson have succeeded in mechanically verifying refactorings using the theorem prover Isabelle/HOL [23]. Their success is heartening, but they are again only dealing with a very simple object language and it is unclear how well their techniques would scale to larger languages, since they do not seem to put much emphasis on modularity.

The experiences of our own verification show that attribute grammars generally lend themselves well to interactive proofs: the evaluation schemata of synthesised and inherited attributes correspond (via Curry-Howard) to induction schemata, and in a prover with an extensible tactic language like Coq the use of domain specific tactics simplifies commonly occurring proof steps and provides some automation. Proofs involving circularly defined attributes are a bit more problematic, since explicit proofs of monotonicity have to be provided; additional automation would be helpful here.

The biggest challenge to verifying major refactorings, however, are extensible proofs. Since our encoding of CRAGs maps attributes to Coq functions, which have to be defined en bloc, there is no way to match JastAdd’s mechanism for gradually extending attribute definitions to deal with new language constructs. The same is true of proof scripts, which likewise are monolithic entities impervious to extensions. Although our experiments show that extending the language requires little additional code and only modest changes to the proof scripts, all these changes still have to be performed in place and are hence not true extensions.

What we would like to see is a three-tiered extension mechanism, in which the object language, the refactoring implementations, and their proofs can all be extended modularly. In particular, the usual approach of introducing new language features in an extended language and then provide a semantics preserving translation back to a core calculus (as done, for example, for inner classes in [13]) would not work here, since the analyses and transformations to be verified have to work on the extended language, not on the core calculus.

We are thus facing the well-known expression problem [25]. Our underlying datatype is the abstract syntax of the object language, on which we write functions (the refactorings and their correctness proofs). Coq as a functional language is, for our purposes, located at the wrong end of the spectrum, where it is easy to extend the functions, but hard to extend the datatype. It would certainly be interesting to survey the literature of solutions to the expression problem to see if they are applicable to our challenge.

Verification of refactorings implemented in other formalisms like JunoGL or a rewriting based approach might require different techniques. The only prior work in this area that we are aware of is Garrido’s work on specifying and verifying Java refactorings in Maude [8]. While impressive for their succinctness, the described refactorings perform very little analysis and are thus not widely applicable. The correctness proofs unfortunately seem not to have been formalised.

5. Conclusion

We have introduced software refactorings as a verification challenge that promises to have a direct impact on practical programming by improving tools used by developers on a daily basis. Formalising and mechanically verifying a realistic refactoring tool for a real-world programming language is a formidable problem that has not been successfully solved yet. We believe that the key to attacking this problem is modularity, not only with respect to the functional decomposition of the refactoring engine itself, but also, even more crucially, with respect to the object language. If the verification of refactorings is to succeed at all, we have to start from a manageable, well-understood core language; if it is to be meaningful, we need a way to modularly extend this core implementation and its verification all the way to a realistic language.

References


