Formal Specification and Verification of Java Refactorings

Alejandra Garrido and José Meseguer
University of Illinois at Urbana-Champaign
201 N. Goodwin Ave., Urbana, IL 61801, USA
{garrido,meseguer}@cs.uiuc.edu

Abstract

There is an extensive literature about refactorings of object-oriented programs, and many refactoring tools for the Java programming language. However, except for a few studies, in practice it is difficult to find precise formal specifications of the preconditions and mechanisms of automated refactorings. Moreover, there is usually no formal proof that a refactoring is correct, i.e., that it preserves the behavior of the program.

We present an equational semantics based approach to Java refactoring. Specifically, we use an executable Java formal semantics in the Maude language to: (i) formally specify three useful Java refactorings; and (ii) give detailed proofs of correctness for two of those refactorings, showing that they are behavior-preserving transformations. Besides the obvious benefits of providing rigorous specifications for refactoring tool builders and rigorous correctness guarantees, our approach has the additional advantage of its executability: our formal refactoring specifications can be used directly to refactor Java programs and yield a provably correct Java refactoring tool.

1. Introduction

Refactorings were defined by Opdyke and Johnson [27] in 1990 as transformations of the source code that make it easier to understand and reuse, while preserving its behavior. The term 'refactoring' differs from program restructuring in that transformations are applied "*not so much to infuse structure into a poorly structured program, but rather to refine the design of an already structured program, and make it easier to reuse*" [26]. In his thesis [26], Opdyke provides a catalog of low-level refactorings for C++ code and precisely describes the preconditions and mechanisms for each one. Roberts does a similar job for Smalltalk code [28]. There have been many publications about refactoring object-oriented programs [15, 30, 16, 25, 23], and numerous Java refactoring tools have been built [1, 2, 3, 4], but there is usually no documentation specifying precisely the preconditions and mechanisms of refactorings, nor there is any proof of correctness.

In order to guarantee the correctness of refactoring tools, two tasks are absolutely essential: (1) refactoring themselves should be formally specified; and (2) each refactoring should be proved correct, i.e., behavior-preserving, with respect to the language's formal semantics. We address tasks (1) and (2) for some Java refactorings in this paper. We have also tackled other Java refactorings in [18] and some refactorings for the C preprocessor in [19]. As further explained in the Related Work section, other researchers have also made contributions in this direction for different languages, including [6, 11, 7, 22, 21, 20]. However, most of that work, with the exception of [6, 11], has concentrated primarily on task (1).

Our approach to tasks (1) and (2) is based, in the case of Java, on an equational executable semantics of sequential Java specified in Maude [9, 10] as part of the JavaFAN project [12]. In fact, the Java semantics in [12] includes also the concurrent features; however, in this work we restrict ourselves to the sequential fragment, which is specified as an equational theory. Our formal specification of several frequently-used Java refactorings (task (1)) extends this equational Java semantics and has the important advantage of being *executable*. This means that the formal definitions of refactorings are at the same time their *implementation*, yielding a Java refactoring tool *for free* from such definitions. It also means that there is no *gap* between specification and implementation, so that any refactoring specification proved correct will indeed operate correctly as specified.

We give also a detailed mathematical proof of correctness (task (2)) for two of those refactorings. That is, we show that they preserve program behavior with respect to the formal Java semantics. Without a formal semantics of the underlying language such mathematical proofs of correctness would be impossible. Our proofs do indeed make essential use of the equational axioms defining the Java semantics and also of the formal refactoring specifications.

Besides the obvious benefits of providing rigorous specifications for refactoring tool builders and rigorous correctness guarantees, plus the already-mentioned benefit of obtaining a provably correct Java refactoring tool for free from the formal specifications, a further important advantage of our approach is its *extensibility*: the algebraic approach we propose makes it straightforward for a user to introduce new user-defined refactorings. Such user-defined refactorings can be specified as algebraic expressions in terms of a basic library of already verified refactorings, and can then be guaranteed to be correct by construction, without any need for additional verification.

Finally, our approach is not restricted to Java or OOP: it can be used in conjunction with any language for which an equational or rewriting logic semantics has been provided. We have, for example, applied the same methodology described in this paper to the C Preprocessor (Cpp), i.e., we have used Maude to specify Cpp refactorings on top of a formal specification of Cpp, which allowed us to prove Cpp refactorings correct based on the semantics of the language [19].

This paper is organized as follows. The next subsection describes other efforts to formalize refactorings. Section 2 gives an overview of the rewriting logic specification for the Java syntax and semantics on which we base our work. Section 3 first describes some generic operations that are used by different refactorings. It then provides details of the specification of '*Push Down Method*', '*Pull Up Field*' and '*Rename Temporary*' and then describes how refactorings can be composed to create new refactorings. Section 4 presents the formal proofs of correctness for '*Push Down Method*' and '*Pull Up Field*' and Section 5 concludes with some remarks and future work.

1.1 Related work

We discuss related work on formal approaches to refactoring and also some related work on program transformation.

Most formal approaches to refactoring focus on task (1), that is, on giving a precise formal specification of refactorings. Work in this direction includes [6, 11, 7, 22, 21, 20]. Several formalisms are used for this purpose. For example, [7, 22] represent programs as graphs and use graph rewriting to specify refactorings. Instead, [21] uses monads and polymorphic functions in Haskell to specify refactorings in a language-generic way. The work in [20, 6, 11] agrees in specifying refactorings as transformation rules with formal predicates for applicability preconditions. In [20] the emphasis is on allowing user-defined refactorings by composing basic ones, whereas in [6, 11] refactorings are viewed as bi-directional transformation rules for which preconditions are specified for use in each direction. Our approach to task (1) has some similarities with [7, 22, 21], since we represent programs as terms and specify refactorings equationally as *conditional term rewrite rules*, which are similar to both graph rewrite rules with applicability constraints and to functional program definitions. What the approaches in [7, 22, 21, 20] lack in relation to our work is a formal semantics of the underlying programming language.

Task (2), in the full sense of proving behavior preservation with respect to the language semantics, is studied much less often, or is even despaired of as in [22], where it is asserted (pg. 253) that "all researchers agree that a full guarantee of preservation of behavior is impossible," although [22] nevertheless shows some preservation of static properties. We of course beg to disagree with such pessimism. In fact, the work in [6, 11] has shown how, for a simplified Java-like sequential language, refactorings can be formally proved correct with respect to an axiomatic, weakest precondition (WP) semantics. Our work is indeed in the same spirit as [6, 11], but addresses Java itself, and uses a different semantics (algebraic, instead of axiomatic) with two important advantages: (1) *executabilty*, since for us both the Java semantics and the refactoring specifications are executable and yield Java tools; and (2) *extensibility*, since our language definitions are modular and will support reasoning about refactorings in a more general context such as multithreaded programs, whereas extensibility of a WP semantics with new features such as exceptions or concurrency would be a nontrivial and perhaps problematic task.

We can also mention some related work on program transformation. Visser [33] surveys several approaches and extensions to term rewriting, like different strategies for tree parsing, tree traversal and programmable transformations. Examples of these approaches are those in Stratego [32] and ASF+SDF [31]. However, these approaches to program transformation differ from refactoring in that transformation rules are usually applied to the entire program, with the objective of *normalizing* the program or optimizing it [33].

Another formalism for program transformation is the method based on WSL (Wide Spectrum Language) [34]. It has an associated tool with a library of transformations that have been proved correct, and the transformations are applied to *refine* a specification or to *abstract* a program written in WSL. Translators to/from WSL exist for IBM 370 Assembler and Jovial [34].

Ahrend, Roth and Sasse have used Maude and the formal specification of the Java language written in Maude [12] to cross-validate the rules of a programming language proof calculus called *KeY* [29, 5]. The rules in *KeY* are program transformation rules that apply only to the first statement of the remaining program [29, 5].

2. Maude Specification of the Java Semantics

Rewriting logic provides a powerful framework for specifying the semantics of both sequential and concurrent programming languages by unifying SOS and equational semantics [24]. Moreover, the Maude environment [9, 10] allows the direct execution of semantic specifications as interpreters with high efficiency.

The semantics of the Java programming language is specified as a rewrite theory $(\Sigma_{Java}, E_{Java}, R_{Java})$, where the signature Σ_{Java} specifies Java's syntax, E_{Java} is a set of equations that specify the semantics of all the sequential features of Java and of the auxiliary operations, and R_{Java} is a set of labelled rules that specify the semantics of all the concurrent features of Java [12]. The complete specification can be found in [17]. In this work we will not consider the concurrent aspects of Java, so we restrict ourselves to the equational theory $(\Sigma_{Java}, E_{Java})$.

The formal specification of a programming language is defined in Maude by a sequence of *modules* [24]. Since we do not consider the concurrent aspects of Java in this presentation, we will only deal with *functional modules*. A Maude's functional module is a set of definitions that specify an equational theory (Σ, E), with Σ a signature specifying a collection of sorts and operations on these sorts, and E a collection of equational axioms. Such functional modules are defined between the keywords `fmod` and `endfm`. Figure 1 shows three Maude modules specifying the syntax of Java classes (`CLASS-SYNTAX`), field declarations (`FIELD-DECLARATION-SYNTAX`) and methods (`METHOD-DECLARATION-SYNTAX`), which have been extracted from [17] but are simplified for this presentation (excluding for example the syntax of exception handling). In order to help understand Figure 1, Maude's basic syntax and semantics is described below.

A Maude module extends a previously defined module by importing it with the keyword `pr` (for 'protecting') or `ex` (for 'extending'). There is a subtle difference between the two, but it is out of the scope of this presentation; a detailed explanation can be found in [10]. Sorts are declared with the keywords `sort` or `sorts`. For example, Class, ClassMember and FieldDeclaration are some of the declared sorts in the Java syntax specification of Figure 1. The modules in the figure use sorts declared in other modules, like Type for basic Java types, CType for class types, Declaration, which represents a variable declaration, and Qid, a sort defined in Maude to represent quoted identifiers.

The set of declared sorts can be partially ordered by a *subsort* relationship. The keywords `subsort` and the "<" character are used for this purpose. The subsort relationship $s \leq s'$ is interpreted semantically by the subset inclusion $A_s \subseteq A_{s'}$ between the sets A_s and $A_{s'}$ of data elements associated to s and s' in an algebra A [10]. In Figure 1, the

```
fmod CLASS-SYNTAX is pr TYPE .
  sorts Modifier ClassMember ClassMembers ClassBody
    Supers Class .
  subsort ClassMember < ClassMembers .
  ops final static abstract public private protected
    transient native: -> Modifier .
  op __ : Modifier Modifier -> Modifier [comm assoc] .
  op {} : -> ClassBody .
  op {_} : ClassMembers -> ClassBody .
  op noMember : -> ClassMembers .
  op __ : ClassMembers ClassMembers -> ClassMembers
                      [assoc id: noMember] .
  op extends_ : CType -> Supers .
  op extends_implements_ : CType ITypes -> Supers .
  op _Class___ : Modifier Qid Supers ClassBody-> Class.
endfm

fmod FIELD-DECLARATION-SYNTAX is ex CLASS-SYNTAX .
  ex DECLARATION-SYNTAX .
  sort FieldDeclaration .
  subsort FieldDeclaration < ClassMember .
  op __; : Modifier Declaration -> FieldDeclaration .
endfm

fmod METHOD-DECLARATION-SYNTAX is ex CLASS-SYNTAX .
  ex DECLARATION-SYNTAX .
  ex TYPE .
  sorts MethodDeclaration Parameters .
  subsort MethodDeclaration < ClassMember .
  subsort Declaration < Parameters .
  op `(`) : -> Parameters .
  op _,_ : Parameters Parameters -> Parameters
                         [assoc id: ()] .
  op _____ : Modifier Type Qid Parameters Block
                      -> MethodDeclaration .
endfm
```

Figure 1. Specification of the syntax of classes, methods and field declarations

expression

`subsort FieldDeclaration < ClassMember`

means that a FieldDeclaration *is a* ClassMember (note that the same is true for a MethodDeclaration). Moreover, the expression

`subsort ClassMember < ClassMembers`

says that a ClassMember is a list of ClassMembers (a list with one element), which makes it easy to deal with lists or single elements in the same way.

Operations are declared using the keyword `op` or `ops` followed by the name of the operation(s), then a colon, then the sorts of the arguments, then an arrow, and finally the sort of the result. Maude understands both prefix and mixfix notation for operations. When declaring an operation with mixfix notation, underscore characters are used to specify the places for the arguments. For example, the operation

`op {_} : ClassMembers -> ClassBody`

constructs a ClassBody by placing a list of ClassMembers between curly braces. Although not shown in Fig. 1, a Java program is defined to have sort Pgm and is constructed with the following operation:

`op __ : Classes Exp -> Pgm .`

that is, a set of classes and an expression to evaluate.

A binary operation in Maude can be declared to satisfy some equational axioms like associativity (with the keyword `assoc`), commutativity (with the keyword `comm`), identity with respect to an identity element (keyword `id`), etc. An example is the operation:

```
op __ : ClassMembers ClassMembers
        -> ClassMembers [assoc id: noMember]
```

which concatenates ClassMembers by using empty juxtaposition syntax and declaring the operation as associative and having noMember as the identity element.

Other modules presented later in the paper use variable declarations and equations. Mathematical variables are declared inside a module with the keywords `var` or `vars`, followed by the variable name(s), a colon and the sort to which the variable(s) belong. Equations define the properties that the operations should satisfy. Equations start with the keyword `eq` followed by two expressions separated by an "=" character. As a final note on the Maude syntax, comments can be added by preceding them with three asterisks or three dashes.

The semantics of Java in Maude uses continuation-passing style [24] to capture the next statement or expression to execute. Continuations in Maude are first-order structures resembling stacks [24]. The continuation is part of a State data structure, which also includes the current state of the memory and environment. The elements of the state have sort StateAttribute and are specified with constructor operators that take as argument the value that each one stores. The main StateAttributes are:

- Context: specified with the constructor c, which takes three elements:
  - Continuation: wrapped with operation k, it includes ContinuationItems that are concatenated with the operator `->`.
  - Environment: wrapped with operation e, and mapping variable names to locations.
  - Current object: on which the current method is executed. It is specified with three components: the static type, the dynamic type and the object environment, all wrapped by the operation o.
- Memory: specified with the constructor m, it maps locations to values.
- Classes: the cl operation wraps the list of all class definitions used in the program.
- Static environment: wrapped with the operation s, the static environment includes all static attributes of all classes.
- Output: this is the accumulated output that is wrapped inside the constructor out and its value is returned at the end of the computation.

```
fmod JAVA-REF is
  pr PGM-SYNTAX .
  sorts JavaRefactoring JavaBlockRefactoring
                            JavaClassesRefactoring .
  op _<-_ : Pgm JavaRefactoring -> Pgm .
  op _<-_ : Classes JavaClassesRefactoring -> Classes .
  op _<-_ : Block JavaBlockRefactoring -> Block .
endfm
```

Figure 2. Module JAVA-REF

To execute a program, the operation run is called on a Pgm, that is, a set of classes `Cl` and an expression `E`, and the operation creates the initial state, which includes `Cl` and a continuation with `E` as the next expression to evaluate. The result is the final value of the state attribute out. In [18], we present more details about semantic definitions. The full specification used for the Java semantics can be found in [17].

3. Formal Semantics of Java Refactorings

This section presents the formal specifications of three Java refactorings: '*Push Down Method*', '*Pull Up Field*' and '*Rename Temporary*'. We have also specified '*Pull Up Method*' and'*Push Down Field*' in [18] . Their preconditions and transformations are based on the formal specification of the Java syntax presented in the previous section.

The module `JAVA-REF` in Figure 2 specifies the basic syntax of refactoring operations. It defines three sorts and three overloaded versions of the operation `<-` that applies a refactoring to different parts of the code:

- a JavaRefactoring is applied to a Pgm (a Java program) and returns a transformed Pgm (or the same Pgm if the preconditions do not hold); an example is the refactoring '*Rename Field*';
- a JavaClassesRefactoring is applied to a set of Classes and returns the same or a transformed set of Classes; an example is the refactoring '*Push Down Method*';
- a JavaBlockRefactoring is applied to a Block and returns the same or a transformed Block. An example is '*Rename Temporary*' refactoring.

During the course of specifying refactorings, we have created some generic operations that were found applicable in many refactorings. These operations are important, since they ease the introduction of new refactorings. We present a few of them in the first subsection. Due to space limitations, we do not give details about the formal specification of auxiliary operations, but they can be found, together with the full specification of the refactorings, in [17].

Finally, the last subsection describes how refactorings can be composed to create new refactorings.

3.1 Generic Auxiliary Operations

In this subsection we list a few of the generic auxiliary operations used in the refactorings described later.

getMethod. This is an example of a query operation. The typing of this operation is:

```
op getMethod:Class Qid Types ->ClassMembers
```

where the first argument is the class that defines the method, the second is the method name, and the third is the parameter types. The return value has sort ClassMembers to account for the possibility of a noMember value (see Figure 1).

noSuperCalls. This is an example of the kind of test operations invoked during the checking of preconditions. Its typing is: `op noSuperCalls : Block -> Bool.` and it checks whether the method body represented by the parameter contains any method calls using "super".

usesVar. This operation checks whether a block refers to a variable. Its signature is:

```
op usesVar : Block Var -> Bool .
```

moveClassMemberMult. This operation is called from every refactoring where there is a class member (a field or a method) that should be removed from a set of classes and added to another set of classes. Examples of these refactorings are *Push Down Method* and *Pull Up Field*. The typing of this operation is:

```
op moveClassMemberMult : ClassMember
                     Classes Classes -> Classes.
```

where the first parameter is the field or method to be moved, the second represents the class(es) from where the member is to be removed, the third parameter is the class(es) to which the member is added, and the return value is the set of all transformed classes.

removeAll. This operation is called after the previous one to remove from the set of all classes those that have been modified, and return the remaining, unchanged classes. The semantics is very general and just removes from the first set of classes the ones in the second set given as parameter.

3.2 Push Down Method Refactoring

With this refactoring, a user selects a method `MN` in a class `CN` and, if the preconditions hold, `MN` is moved from `CN` to all subclasses of `CN`. Figure 3 shows the module specifying this refactoring. The main operation is specified with:

```
op PushDownMethod : Qid Qid Types
            -> JavaClassesRefactoring.
```

```
fmod PUSH-DOWN-METHOD is
  pr JAVA-REF . pr CLASS-REF-HELPERS .
  var md:Modifier. vars CN MN:Qid. var cb:ClassBody.
  --- other variable declarations omitted

  op PushDownMethod : Qid Qid Types
              -> JavaClassesRefactoring .
  eq Cl <- PushDownMethod(CN, MN, TS)
   = if precondsPushDownMethodHold(Cl, CN, MN, TS)
     then applyPushDownMethod(Cl, CN, MN, TS)
     else Cl fi .

  op precondsPushDownMethodHold : Classes Qid Qid
                                    Types -> Bool .
  eq precondsPushDownMethodHold(noClass, CN, MN, TS)
  = false .            --- no classes in the program ***1
  eq precondsPushDownMethodHold(((md Class CN sp cb)
        Cl), CN, MN, TS)
   = isAbstract(md) and                               ***2
      precondsPushDownMethodHold((md Class CN sp cb),
           getMethod(cb,MN,TS), subclasses(CN,Cl), Cl).
  eq precondsPushDownMethodHold(Cl, CN, MN, TS)
   = false [owise] .              --- no class CN   ***1
  op precondsPushDownMethodHold : Class ClassMembers
                             Classes Classes -> Bool .
  eq precondsPushDownMethodHold(C, noMember, SubCl, Cl)
   = false .                      --- no method MN  ***1
  eq precondsPushDownMethodHold(C, CM, noClass, Cl)
   = false .                      --- no subclasses ***5
  eq precondsPushDownMethodHold(C, (md T MN pl block),
        SubCl, Cl)
   = not isStatic(md) and noSuperCalls(block) and  ***3,4
        noCallsToSuper(SubCl, MN, GetTypes(pl)).     ***5

  op applyPushDownMethod : Classes Qid Qid Types
                            -> Classes .
  eq applyPushDownMethod(((md Class CN sp cb) Cl),
                            CN, MN, TS)
   = applyPushDownMethod(getMethod(cb, MN, TS),
        (md Class CN sp cb), subclasses(CN, Cl), Cl).
  op applyPushDownMethod : MethodDeclaration Class
                            Classes Classes -> Classes .
  eq applyPushDownMethod(MD, C, SubCl, Cl)
   = (moveClassMemberMult(MD, C, SubCl)
          removeAll(Cl, SubCl)) .
endfm
```

Figure 3. Specification of Push Down Method

where the first argument is the class name (`CN`), the second is the method name (`MN`) and the third argument are the types of the method parameters (`TS`), to distinguish it from other possible overloaded names for `MN`. The operation PushDownMethod, when applied to a set of classes `Cl`, first checks the preconditions by calling precondsPushDownMethodHold, and if that operation returns true then applies the transformation by calling applyPushDownMethod. Otherwise, it just returns the same set of classes `Cl`. This is simply specified with the first equation in module `PUSH-DOWN-METHOD` (see Figure 3).

The preconditions for this refactoring are the following (note that some equations in Fig. 3 are numbered on the right to provide easy reference with the following list):

1. The input is valid, i.e., there is a class named `CN` which defines a method `MN` with parameter types `TS`.

2. Class `CN` is abstract, so `MN` will never be called on an

instance of CN.

3. Method MN(TS) is not static, so it will not be called on the class CN.

4. The body of MN does not call other methods using super, to avoid changes in method look-up.

5. Class CN has subclasses, and none of the subclasses call MN by way of super, for the same reason as the previous one.

Note the use of the attribute [owise] (otherwise) in the fourth equation in Fig. 3, which makes it applicable when all the previous equations have failed to apply, i.e., the set Cl is not empty but it does not contain a class named CN. The attribute [owise] can be desugared into an equivalent conditional specification [10].

The mechanics of applyPushDownMethod are to retrieve the method MN(TS) from the class CN, retrieve CN's subclasses, and call the overloaded version of applyPushDownMethod. The latter calls the auxiliary operations moveClassMemberMult, to move the MethodDeclaration MD from the superclass C to all subclasses SubCl, and the operation removeAll to append to the result of moveClassMemberMult (the changed classes), the rest of the classes that have not been changed.

3.3 Pull Up Field Refactoring

This refactoring is used when all subclasses of a class CN define the same field FN, which should be therefore abstracted to the superclass CN. Figure 4 shows the formal specification of this refactoring. The operation that applies the refactoring is PullUpField, which receives the class name CN and the field name FN as parameters, and in the same way as in the previous refactorings, carries out the transformation if the preconditions hold.

The preconditions for this refactoring are:

1. There is a class named CN in the set of classes.

2. Class CN has at least one subclass.

3. Class CN does not define the field FN.

4. All subclasses of CN define the field FN.

These preconditions are checked by operation precondsPullUpFieldHold (again, the equations in Fig. 4 are numbered to show which equation checks each precondition).

The transformation is carried out by operation applyPullUpField, which in turn calls moveClassMemberMult to move the field from the subclasses to the superclass, and calls removeAll to get the subset of unchanged classes, just like in previous cases.

```
fmod PULL-UP-FIELD is
  pr JAVA-REF . pr CLASS-REF-HELPERS .
    ---variable declarations omitted
  op PullUpField : Qid Qid -> JavaClassesRefactoring .
  eq Cl <- PullUpField(CN, FN)
   = if precondsPullUpFieldHold(Cl, CN, FN)
     then applyPullUpField(Cl, CN, FN)
     else Cl fi .

  op precondsPullUpFieldHold : Classes Qid Qid -> Bool .
  eq precondsPullUpFieldHold(noClass,CN,FN) = false. ***1
  eq precondsPullUpFieldHold(((md Class CN sp cb) Cl),
        CN, FN)
   = subclasses(CN, Cl) =/= noClass   and            ***2
     getField((md Class CN sp cb), FN) == noMember   ***3
     and
     allClassesDefineField(subclasses(CN,Cl),FN).    ***4
  eq precondsPullUpFieldHold(Cl,CN,FN)= false[owise].***1

  op applyPullUpField : Classes Qid Qid -> Classes .
  eq applyPullUpField(((md Class CN sp cb) Cl), CN, FN)
   = applyPullUpField(subclasses(CN, Cl),
                      (md Class CN sp cb), FN, Cl) .
  op applyPullUpField : Classes Class Qid Classes
                        -> Classes .
  eq applyPullUpField((SubC SubCl), SupC, FN, Cl)
   = (moveClassMemberMult(getField(SubC, FN),
            (SubC SubCl), SupC)
      removeAll(Cl, (SubC SubCl))) .
endfm
```

Figure 4. Specification of Pull Up Field Refactoring

3.4 Rename Temporary Refactoring

Renaming is probably the best known and most used refactoring. Figure 5 shows the Maude specification of *Rename Temporary Variable* for Java. It differs from the previous refactorings in several aspects: it is an example of a JavaBlockRefactoring, it does not involve code movement, and it requires the construction of a symbol table of the block on which the refactoring is applied, to check variable declarations and visibility. The operation that carries out this refactoring is RenameTemp. It receives as parameters the Old name and the New name for the variable, and the code location L of the selected declaration for Old. This location helps to distinguish between different possible declarations of Old. The location L is specified as a list of numbers (of sort NatList) that represents a path from the root in the syntax tree and identifies the positions of the nested scopes that contain the declaration for Old, with the entire list of numbers indicating the position for the declaration itself. Each entry in the symbol table has an associated NatList specifying the location of its declaration.

The preconditions for this refactoring are checked with operation precondsRenTempHold, which requires the construction of the symbol table. The operation computeSymbolTable is specified in module SYMBOL-TABLE. This module is extended by the module ST-QUERIES, which specifies the operations isDeclara-

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

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

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

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

Figure 5. Specification of Rename Temporary Refactoring

tionAt, used to check if there is a declaration for `Old` at location `L`, and isNameVisible, to check that the `New` name is not visible in the scope of `Old`.

The operation applyRenTemp traverses nested blocks until the scope for the selected variable `Old` is found (in the third equation for applyRenTemp), when it calls replace to change each reference to `Old` by `New`. Intermediate scopes are replaced by operation replaceSubtree. In turn, the operation subterm(bs,N) returns the N-th subterm inside bs. The equations for these operations are omitted due to space limitations; they can be found in [17].

3.5 User-Definable Refactorings

Kniesel and Koch [20] argue that refactoring tools should allow the composition of refactorings by end users. With our approach, composition of refactorings is not only possible but easy, by arranging refactorings in a sequence with the `<-` operator. For example, take the refactoring '*Pull Up Field*'; as described in [16], it is possible that originally, the fields to be pulled up do not have the same name, so it is first necessary to give all fields the same name and then pull the field up. Therefore, a user may want to define a refactoring '*Rename And Pull Up Field*' applicable to this more general situation. It takes a class name, the list of different field names in the subclasses and the target name for all fields, and then first applies '*Rename Field*' to the fields in the subclasses and then applies '*Pull Up Field*'. This can be easily defined by the equation:

```
eq Cl <- RenameAndPullUpField(CN, LNs, TN)
 = (Cl <- RenameFieldAny(subclasses(#c(CN),
        Cl), LNs, TN))
       <- PullUpField(CN, TN) .
```

where `Cl`:Classes, `CN`,`TN`:Qid, `LNs`:QidList, and RenameFieldAny renames, in each class received as first parameter, the fields with any of the names in the second parameter, to the target name in the third parameter.

Note that any user-definable refactoring constructed this way, as successive application of a finite number of basic refactorings, *will preserve program behavior by construction*, provided we have already verified that the basic refactorings it uses do preserve such behavior.

4. Proving Correctness of Java Refactorings

4.1 Correctness of Push Down Method

Theorem 1. *Applying* PushDownMethod *does not change the output of the program:*

```
run(Cl E) = run((Cl <-
    PushDownMethod(CN, MN, TS)) E)
```

where `Cl`*:Classes,* `E`*:Exp,* `CN`,`MN`*:Qid and* `TS`*:Types.*

PROOF. Note that if the return value of precondsPushDownMethodHold is false, no changes are applied to the set of classes `Cl` and the theorem trivially holds. Otherwise, we know that there is a class named `CN` in `Cl`, that it is abstract, it has a non-static method `MN(TS)` and has at least one subclass. Let us call that subclass `SubCN`. Using this information and by applying the equations in module `PUSH-DOWN-METHOD`, we can derive the following:

```
Cl <- PushDownMethod(CN, MN, TS)
= ((md Class CN sp
      {CMs (m T MN pl block)})
    (mds Class SubCN sps {CMsub}) Cl')
         <- PushDownMethod(CN, MN, TS)
= (moveClassMemberMult((m T MN pl block),
     (md Class CN sp
        {CMs (m T MN pl block)}),
     (mds Class SubCN sps {CMsub}))
  removeAll(
     (mds Class SubCN sps {CMsub}) Cl',
     (mds Class SubCN sps {CMsub})))
= ((md Class CN sp {CMs})
   (mds Class SubCN sps
      {CMsub (m T MN pl block)}) Cl')
```

assuming variables `md`,`mds`,`m`:Modifier, `sp`,`sps`:Supers, `CMs`,`CMSub`:ClassMembers, `T`:Type, `pl`:Parameters, `block`:Block.

As described in Section 2, the operation `run(Cl E)` creates the initial program State, which in turn creates a continuation where the expression `E` is the next step to execute. From there, it may eventually happen that the method `MN(TS)` is called. If it is never called, the theorem holds trivially. Otherwise, we know that `MN(TS)` cannot be called on an instance of `CN`, because `CN` is abstract. Therefore, let us assume that it is called on an instance of `SubCN`. The semantics of a method call is formally specified with the equations that appear in Figure 6. Upon an expression `(E . mn El)`, the semantics is to evaluate `E` to an object of the form `o(CT,CT',oEnv)`, then evaluate the arguments `El` to a list of values `Vl`, and as specified in the second equation that appears in Fig. 6, call GetMethod to obtain a MethodAux, which is a representation of a method body to execute.

The key to proving this theorem lies in the operation GetMethod, which before or after applying the refactoring, should return the same MethodAux. We therefore need to prove the following equality:

```
GetMethod(CT, mn, Vl,
  ((md Class CN sp {CMs (m T MN pl block)})
   (mds Class SubCN sps { CMsub }) Cl'))
= GetMethod(CT, mn, Vl,
  ((md Class CN sp { CMs })
   (mds Class SubCN sps
        { CMsub (m T MN pl block) }) Cl'))
```

which is easily proven by evaluation of the equations in Fig. 6. Moreover, the precondition that `SubCN` methods do not call `super.MN` ensures that `MN` will not be searched starting from `CN`.

Also, the precondition that the body of `MN` does not call other methods using super ensures that no errors will occur during the execution of `MN`. □

4.2 Correctness of Pull Up Field

Theorem 2. *Applying PullUpField does not change the output of the program:*
`run(Cl E) = run((Cl <- PullUpField(CN, FN)) E)`
where `Cl`*:Classes,* `E`*:Exp, and* `CN`*,* `FN`*:Qid.*

PROOF. If the return value of precondsPullUpFieldHold is false, no changes are applied to the set of classes `Cl` and the theorem trivially holds. Otherwise, we know that there is a class named `CN` in `Cl`, and that every subclass of `CN` defines `FN` but `CN` does not. Let us call `SubCN` one of those subclasses. Using this information and by applying the equations in module `PULL-UP-FIELD`, we can derive the following:

```
Cl <- PullUpField(CN, FN)
= ((md Class CN sp { CMs })
   (mds Class SubCN sps
       {CMsub (m T FN ;)}) Cl')
           <- PullUpField(CN, FN)
= (moveClassMemberMult((m T FN ;),
      (mds Class SubCN sps
            {CMsub (m T FN ;)}),
      (md Class CN sp {CMs}))
   removeAll((mds Class SubCN sps
            {CMsub (m T FN ;)}) Cl',
      (mds Class SubCN sps
            {CMsub (m T FN ;)})))
  = ((md Class CN sp {CMs (m T FN ;)})
     (mds Class SubCN sps {CMsub}) Cl')
```

assuming variables `md,mds,m`:Modifier, `sp,sps`:Supers, `CMs,CMSub`:ClassMembers and `T`:Type, besides the variables previously defined. Note that `(m T FN ;)` represents the declaration for field `FN`.

As described in Section 2, the operation `run(Cl E)` creates the initial program State, which in turn creates a continuation where the expression `E` is the next step to execute. From there, if an object of type `CN` is created, it will have the additional field `FN` but it will not be used (assuming we start from a correct program). So there will not be any change in the functionality, and therefore in the output.

Let us suppose that an object of class `SubCN` is created. We give a brief description of the semantics of object creation here and more details of the specification can be found in [18]. The equation:

```
eq k((new CT (El)) -> K)
  = k((El) -> newObj(CT) -> K).
```

specifies that when a 'new' expression is found, the semantics is to first evaluate the arguments of the constructor and then apply the operation newObj. The operation newObj calls itself on each class in the hierarchy from `CT` to `Object`, and once `Object` is reached, it stacks the operation created in the continuation. The fields of each class in the path from `CT` to `Object` are then 'declared', i.e., are added to the environment. At the end of the field declarations of each class, the names in the global environment are moved to the current object environment. Let us then apply the equations involved in the creation of an instance of `SubCN`, assuming that the constructor does not take any arguments and that the superclass of `CN` is `Object`. Otherwise there would be extra steps involved below, but they do not interfere with the outcome of our proof.

```
k((new SubCN ()) -> K)
= k(noVal -> newObj(SubCN) -> K)
= k(newObj(SubCN) -> endnew(...
= k(newObj(SuperClass(SubCN, Cl)) ->
             newObj(SubCN) -> endnew(...
= k(newObj(CN)-> newObj(SubCN)-> endnew...
= k(newObj(SuperClass(CN, Cl)) ->
     newObj(CN)->newObj(SubCN)->endnew...
= k(newObj(Object) -> newObj(CN) ->
             newObj(SubCN) -> endnew...
= k(created-> newObj(CN) -> newObj(SubCN)
             -> endnew...
```

```
eq k((E . mn El) -> K) = k((E, El) -> . (mn) -> K) .
eq c(k((o(CT, CT', oEnv), Vl) -> . (mn) -> K), cnt), cl(Cl) =
   c(k(GetMethod(CT', mn, Vl, Cl) -> fn (o(CT, CT', oEnv), mn, Vl) -> K), cnt), cl(Cl) .

op GetMethod : CType MName ValueList Classes -> MethodAux .
eq GetMethod(CT, mn, Vl, Cl) = GetMethod(GetMethods(CT, mn, Cl), Vl, Cl) [owise] .
  --- first find all methods named mn and then filter by parameter types
op GetMethod : MethodList ValueList Classes -> MethodAux .
eq GetMethod((m(CT, Tl, pl, block), Ml), Vl, Cl) =
   if Compatible(pl, Vl, Cl) then m(CT, Tl, pl, block) else GetMethod(Ml, Vl, Cl) fi .
op GetMethods : CType MName Classes -> MethodList .
eq GetMethods(CT, mn, Cl) = Compact(GetMethodList(CT, mn, Cl, Cl), Cl) .
op GetMethodList : CType MName Classes Classes -> MethodList .
eq GetMethodList(CT, mn, noClass, Cl') = none .
eq GetMethodList(CT, mn, ((md Class Xc sp cb) Cl), Cl') =
   (if SuperOf(#c(Xc), CT, Cl') then GetMethodList(CT, mn, #c(Xc), cb) else none fi), GetMethodList(CT,mn,Cl,Cl') .
op GetMethodList : CType MName CType ClassBody -> MethodList .
eq GetMethodList(CT, mn, CT', {CM}) = GetMethodList(CT, mn, CT', CM) .
op GetMethodList : CType MName CType ClassMembers -> MethodList .
eq GetMethodList(CT, mn, CT', noMember) = none .
eq GetMethodList(CT, #m(mc), CT', ((md T mc pl block) CM)) =
   (m(CT', GetTypes(pl), pl, block) fi fi), GetMethodList(CT, #m(mc), CT', CM) .
op Compact : MethodList Classes -> MethodList .
eq Compact((m(CT, Tl, pl, block), m(CT', Tl, pl', block'), Ms), Cl) =
   Compact(((if SuperOf(CT, CT', Cl) then m(CT', Tl, pl', block') else m(CT, Tl, pl, block) fi), Ms), Cl) .
```

Figure 6. Semantics of a method call

At this point, the next equation will call the operation newObj on the class CN and CN's body, and a subsequent equation will call newObj on SubCN and SubCN's body. Before the refactoring, the declaration for FN will be added to the environment when SubCN's body is processed. After the refactoring has been applied, the declaration for FN will be added to the environment earlier that before, when CN's body is processed, but since the environment is a commutative data structure, the order of field declarations does not change the semantics of the resulting object. □

5. Conclusions

We have presented an executable formal specification of three Java refactorings that were developed on top of the formal specification of the Java programming language, and we have given detailed proofs of correctness of two such refactorings based on the underlying Java semantics. This work shows how three important goals can be simultaneously achieved within the same framework: (1) formally specifying refactorings for a language; (2) proving them correct with respect to the language semantics; and (3) deriving a provably correct refactoring tool from the formal refactoring specifications. However, this is work in progress and further research is needed both for Java refactoring and to make the technology more generic.

For Java, the obvious tasks ahead include: (i) extending the current library of generic operations to facilitate the introduction of new refactorings; (ii) extending the library of basic refactorings to include most of the refactorings supported by other tools and entirely new ones, for example for multi-threaded programs; (iii) developing formal proofs of correctness for all those refactorings and also mechanized versions of such proofs; (iv) developing a user interface for the Java refactoring tool easing both refactoring application and introduction of new user-defined refactorings; (v) integrating this tool within the JavaFAN environment and experimenting with a substantial collection of case studies to evaluate the tool in practice and compare it with other tools.

With the intention of generalizing our approach, we will also work on introducing Maude strategies [10] in the definition of refactorings. These strategies will help defining generalized traversal mechanisms, like ASF+SDF traversal functions [31] or Stratego strategies [32], simplifying replacement operations in refactorings like RenameTemp, which currently requires one equation for each kind of statement and each kind of expression in the language.

The semantics-based approach to refactoring is part of a broader effort to base software tools on semantic definitions (see [24, 14, 13, 12, 5, 8]). A key emphasis in this broader effort is the development of *generic* techniques, that can be applied to many concrete language instances. For example, the same methodology applied here to Java has been applied in [19] to formally specify C preprocessor refactorings and prove them correct. A longer-term goal is to develop a *generic library of provably correct refactorings*, based on modular semantic definitions of language features, so that a correct refactoring tool for a given language will be derived automatically from a modular semantics for it.

Acknowledgements. We would like to thank Ralf Sasse for his help in this project. This research has been supported in part by ONR Grant N00014-02-1-0715.

References

[1] Eclipse.org main page. http://www.eclipse.org.

[2] IntelliJ IDEA: the most intelligent Java IDE around. http://www.intellij.com/idea/.

[3] jFactor. http://www.instantiations.com/jfactor/.

[4] Xrefactory for Java. http://xref-tech.com/xrefactory-java/main.html.

[5] W. Ahrendt, A. Roth, and R. Sasse. Automatic Validation of Transformation Rules for Java Verification against a Rewriting Semantics. In *LPAR'05*, Jamaica, 2005.

[6] P. Borba, A. Sampaio, A. Cavalcanti, and M. Cornélio. Algebraic Reasoning for Object-Oriented Programming. *Science of Computer Programming*, 52(1-3), 2004.

[7] P. Bottoni, F. P. Presicce, and G. Taentzer. Specifying Integrated Refactoring with Distributed Graph Transformations. In *Proc. of AGTIVE 2003 (LNCS 3062)*, 2003.

[8] F. Chen, M. Hills, and G. Roşu. A Rewrite Logic Approach to Semantic Definition, Design and Analysis of Object-Oriented Languages. Technical Report UIUCDCS-R-2006-2702, Univ. of Illinois at Urbana-Champaign, 2006.

[9] M. Clavel, F. Durán, S. Eker, P. Lincoln, N. Martí-Oliet, J. Meseguer, and J. Quesada. Maude: specification and programming in rewriting logic. *Theoretical Computer Science*, 285:187–243, 2002.

[10] M. Clavel, F. Durán, S. Eker, P. Lincoln, N. Martí-Olliet, J. Meseguer, and C. Talcott. *Maude Manual (Ver. 2.2)*. http://maude.cs.uiuc.edu/maude2-manual/, 2005.

[11] M. Cornélio. *Refactorings as Formal Refinements*. PhD thesis, Univ. of Pernambuco, Brazil, 2004.

[12] A. Farzan, F. Chen, , J. Meseguer, and G. Roşu. Formal Analysis of Java Programs in JavaFAN. In *Int. Conf. on Computer Aided Verification*, Boston, Mass., 2004.

[13] A. Farzan and J. Meseguer. Partial Order Reduction for Rewriting Semantics of Programming Languages. Technical Report UIUCDCS-R-2005-2598, Univ. of Illinois at Urbana-Champaign, 2005.

[14] A. Farzan, J. Meseguer, and G. Roşu. Formal JVM Code Analysis in JavaFAN. In *Proc. of AMAST'04*, 2004.

[15] B. Foote and W. Opdyke. Lifecycle and refactoring patterns that support evolution and reuse. In *Pattern Languages of Program Design I*. Addison-Wesley, 1995.

[16] M. Fowler. *Refactoring. Improving the Design of Existing Code*. Addison-Wesley, 1999.

[17] A. Garrido. Java Refactoring in Maude. https://netfiles.uiuc.edu/garrido/www/JavaRef/.

[18] A. Garrido and J. Meseguer. Formal Specification and Verification of Java Refactorings. Technical Report UIUCDCS-R-2006-2731, Univ. of Illinois at Urbana-Champaign. https://netfiles.uiuc.edu/garrido/www/publications.html, 2006.

[19] A. Garrido, J. Meseguer, and R. Johnson. Algebraic Semantics of the C Preprocessor and Correctness of its Refactorings. Technical Report UIUCDCS-R-2006-2688, Univ. of Illinois at Urbana-Champaign. https://netfiles.uiuc.edu/garrido/www/publications.html, 2006.

[20] G. Kniesel and H. Koch. Static composition of refactorings. *Science of Computer Programming*, 52(1-3), 2004.

[21] R. Lämmel. Towards generic refactoring. In *Proc. of Workshop on Rule-based Programming*, 2002.

[22] T. Mens, N. V. Eetvelde, S. Demeyer, and D. Janssens. Formalizing Refactorings with Graph Transformations. *Journal of Software Maintenance and Evolution*, 17(4), 2005.

[23] T. Mens and T. Tourwé. A Survey of Software Refactoring. *IEEE Transactions on Software Engineering*, 30(2), 2004.

[24] J. Meseguer and G. Roşu. The Rewriting Logic Semantics Project. In *Proc. of Structural Operational Semantics (SOS'05)*, 2005.

[25] M. Ó Cinnéide. *Automated Application of Design Patterns: a Refactoring Approach*. PhD thesis, Univ. of Dublin, 2001.

[26] W. Opdyke. *Refactoring Object-Oriented Frameworks*. PhD thesis, University of Illinois at Urbana-Champaign, 1992.

[27] W. Opdyke and R. Johnson. Refactoring: An Aid in Designing Application Frameworks and Evolving Object-Oriented Systems. In *Proc. of Sym. on OO Programming Emphasizing Practical Applications (SOOPPA'90)*, 1990.

[28] D. Roberts. *Eliminating Analysis in Refactoring*. PhD thesis, University of Illinois at Urbana-Champaign, 1999.

[29] R. Sasse. Taclets vs. Rewriting Logic - Relating Semantics of Java. Master's thesis, University of Karlsruhe, 2005.

[30] L. Tokuda and D. Batory. Evolving object oriented designs with refactoring. In *ASE'99*, 1999.

[31] M. van den Brand, P. Klint, and J. Vinju. Term Rewriting with Traversal Functions. *ACM Transaction on Software Engineering and Methodology*.

[32] E. Visser. Program Transformation with Stratego/TX: Rules, Strategies, Tools, and Systems in StrategoXT-0.9. In *Domain-Specific Program Generation. In: LNCS vol.3016. Springer-Verlag*, 2004.

[33] E. Visser. A Survey of Strategies in Rule-Based Program Transformation Systems. *Journal of Symbolic Computation*, 40(1), 2005.

[34] M. Ward and K. Bennett. Formal Methods to Aid the Evolution of Software. *Int. Journal of Software Engineering and Knowledge Engineering*, 5(1), 1995.