Using CafeOBJ to Mechanise Refactoring Proofs and Application

Antonio Carvalho Júnior¹, Leila Silva¹, Márcio Cornélio²

¹Departamento de Ciência da Computação e Estatística
Universidade Federal de Segipe (UFS)
CEP 41900-000 – São Cristóvão – SE, Brazil

²Departamento de Sistemas Computacionais
Escola Politécnica – Universidade de Pernambuco (UPE)
Rua Benfica, 455 – 50750-410, Recife – PE, Brazil
acjr@dcce.ufs.br, leila@ufs.br, mlc@dsc.upe.br

Abstract. In this paper we show how rewriting systems, in particular CafeOBJ, can be used to automatically prove refactoring rules. In addition, a small case study that illustrates the application of a refactoring rule in an arbitrary program is also developed. Our approach is based on a sequential object-oriented language of refinement (ROOL) similar to Java. We have implemented the ROOL grammar in CafeOBJ, as well as the laws that define its semantics. Each refactoring rule is derived by the application of these laws, in a constructive way. The refactorings are also implemented in CafeOBJ, allowing the reduction of an arbitrary program.

1. Introduction

Changes are common to software. Many practitioners recognize that changing an object-oriented software is easier than conventional ones [14]. Refactoring is the activity of modifying a software system by preserving the external behaviour, perceived by the user. In fact, just the internal software structure is affected, by for example, moving attributes and methods between classes.

In [10, 20] several refactorings are introduced in a rather informal way. Opdyke [18] formalises refactorings to the degree that they can be encoded in tools. Cinnéide and Nixon [6] present a semi-formal approach to demonstrate behaviour preservation for design patterns transformations, defined in terms of refactorings. Lano et al [12] formally justify design patterns by relating two set of classes, the “before” and “after” systems. The “after” system consists of a collection of classes organised according to patterns. Selected axioms are used to prove that the “after” system is an extension of the “before” one. Cornélio [8] adopts a transformational approach, that is constructively based on rules, to formalise several refactorings introduced previously by Fowler [10].

The work proposed by Cornélio [8] is based on transformation rules between meta-programs in ROOL (Refinement Object-Oriented Language) [3], which is a subset of sequential Java [1] with classes, inheritance, visibility control for attributes, dynamic binding and recursion. Refactorings are described in ROOL as rules, relating the meta-program on the left-hand side to another one on the right-hand side. Each rule is derived from programming and refinement laws, also expressed in ROOL.

In this paper we show how the refactoring rules can be proved using the rewriting system CafeOBJ [17]. Moreover, a case study showing the application of the refactorings is also developed. In this way, this work complements the formal rigour of [8], as manual proofs can
easily hold mistakes. In addition, the results achieved suggest that rewriting systems can be used as supporting tools for the construction of refactoring environments.

Although CafeOBJ does not seem to be used in the context of refactorings, the use of rewriting systems to implement algebraic reduction strategies have already been proposed. Lira et al [13] use Maude [7] to implement a reduction strategy for object-oriented languages and Silva et al [19] implement in CafeOBJ a reduction strategy in the context of hardware/software partitioning.

This paper is organised as follows. Section 2 introduces ROOL, the language used to formalise the refactoring. In Section 3 we present two refactoring rules used to illustrate the work here developed. The rewriting system CafeOBJ is briefly introduced in Section 4. Section 5 presents the mechanisation of the refactoring proofs, as well as the development of a small case study. Finally, Section 6 gives the conclusions and directions for future work.

2. ROOL and Laws

ROOL [4, 3] is an object-oriented language based on sequential Java. It allows reasoning about object-oriented programs and specifications, as both kind of constructs are mixed in the style of Morgan’s refinement calculus [15, 16]. The semantics of ROOL, as usual for refinement calculi, is based on weakest preconditions. The imperative constructs of ROOL are based on the language of Morgan’s refinement calculus [15], which is an extension of Dijkstra’s language of guarded commands [9]. The meaning of ROOL constructs is expressed by algebraic laws, in the style of [15].

A program \(\text{cds} \bullet \text{c}\) in ROOL is a sequence of classes \(\text{cds}\) followed by a main command \(\text{c}\). Classes are declared as in the following example, where we define a class \(\text{Employee}\).

```rool
class \text{Employee} extends \text{object}
  pri salary : \text{int};
  meth \text{getSalary} = (\text{res} r : \text{int} \bullet r := \text{self}.\text{salary})
  meth \text{setSalary} = (\text{val} s : \text{int} \bullet \text{self}.\text{salary} := s)
  new = \text{self}.\text{salary} := 0
end
```

Classes are related by single inheritance, which is indicated by the clause \text{extends}. The class \text{object} is the default superclass of classes. So, the \text{extends} clause could have been omitted in the declaration of \text{Employee}. The class \text{Employee} includes a private attribute named \text{salary}; this is indicated by the use of the \text{pri} qualifier. Attributes can also be protected (\text{prot}) or public (\text{pub}). ROOL allows only redefinition of methods which are public and can be recursive; they are defined using procedure abstraction in the form of Back’s parameterized commands [2, 5].

A parameterised command can have the form \text{val} \(x : T \bullet \text{c}\) or \text{res} \(x : T \bullet \text{c}\), which correspond to the call-by-value and call-by-result parameter passing mechanisms, respectively. For instance, the method \text{getSalary} has a result parameter \(r\), whereas \text{setSalary} has a value parameter \(s\). Initialisers are declared by the \text{new} clause. ROOL also includes specification constructs from Morgan’s refinement calculus [15], like the specification statement.

The laws of ROOL, mainly those related to object-oriented features, address context issues. We use \(\text{cds}_1 =_{\text{cds}, \text{c}} \text{cds}_2\), where \(\text{cds}\) is a context of class declarations \(\text{cds}_1\) and \(\text{cds}_2\), and \(\text{c}\) is the main command to denote the equivalence of set of class declarations \(\text{cds}_1\) and \(\text{cds}_2\). This notation is just an abbreviation for the program equivalence \(\text{cds}_1 \text{cds} \bullet \text{c} = \text{cds}_2 \text{cds} \bullet \text{c}\), which is formalised in [4, 3]. As an example, in what follows we present some laws of ROOL. We
write ‘(→)’ when some conditions must be satisfied for the application of the law from left to right. We use ‘(←)’ to indicate the conditions that are necessary for applying a law from right to left. By writing ‘(↔)’ we indicate the conditions necessary in both directions. Conditions are described in the provided clause of laws.

A complete set of ROOL laws can be found in [8]. Here we only introduce some laws of command and laws of classes, necessary to understand the mechanisation of the proofs, presented in Section 5. The explanations of these laws are extracted from [8].

Law 1 ⟨:= − ⋚ right dist⟩ distributes an assignment over a list of conditional commands, provided e is total.

Law 2 ⟨var elim⟩ eliminates a variable that does not appear in c and Law 3 ⟨var- := final value⟩ removes an assignment at the very end of its scope, as this assignment introduces no effect.

Law 4 ⟨var- := initial value⟩ assigns an expression to a variable before its first use and Law 5 ⟨order independent assignment⟩ reorders assignments, if they are not data dependent. The symbol ⊑ means the standard refinement symbol.

Law 6 ⟨method elimination⟩, allows the remotion of a method from a class if it is not called by any classes in cds, in the main command c, nor inside class C. To apply this law from right to left, the method m cannot be already declared in C nor in any of its superclasses or subclasses, so that we can introduce a new method in a class. The notation b.m refers to calls to a method m via expressions whose static type is exactly b. The subclass relation is denoted by ≤. We write B ≤ A to denote that a class B is a subclass of a class A.
Law 6 *(method elimination)*

\[
\begin{array}{|c|c|}
\hline
\text{class } C \text{ extends } D \\
\text{ads} \\
\text{meth } m \triangleq pc \\
\text{mts} \\
\text{end} \\
\hline
\end{array} =_{edsc,c}
\begin{array}{|c|c|}
\hline
\text{class } C \text{ extends } D \\
\text{ads} \\
\text{mts} \\
\text{end} \\
\hline
\end{array}
\]

\text{provided}

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

\[\square\]

Law 7 *(move original method to superclass)*, allows us to move an original method to a superclass. Between other requirements, this law demands that occurrences of \textbf{self} in methods to be moved are cast.

Law 7 *(move original method to superclass)*

\[
\begin{array}{|c|c|}
\hline
\text{class } B \text{ extends } A \\
\text{ads} \\
\text{mts} \\
\text{end} \\
\hline
\end{array} =_{edsc,c}
\begin{array}{|c|c|}
\hline
\text{class } C \text{ extends } B \\
\text{ads}' \\
\text{meth } m \triangleq pc \\
\text{mts}' \\
\text{end} \\
\hline
\end{array}
\]

\text{provided}

\((\rightarrow) (1) \textbf{super} \text{ and private attributes do not appear in } pc; (2) m \text{ is not declared in any superclass of } B \text{ in } cds;\)
\((\rightarrow) (1) m \text{ is not declared in } mts, \text{ and can only be declared in a class } D, \text{ for any } D \leq B \text{ and } D \nsubseteq C, \text{ if it has the same parameters as } pc; (2) pc \text{ does not contain uncast occurrences of } \textbf{self} \text{ nor expressions in the form } ((C)\textbf{self}).a \text{ for any private attribute } a \text{ in } ads';\)
\((\leftarrow) (1) m \text{ is not declared in } mts'; (2) D.m, \text{ for any } D \leq B, \text{ does not appear in } cds, c, mts \text{ or } mts'.\)

\[\square\]

3. Refactoring Rules

A comprehensive set of refactoring rules which captures and formalises most of the refactorings informally introduced in [10] is presented in [8]. Refactoring rules are described by means of two boxes written side by side, along with \textit{where} and \textit{provided} clauses. We use the \textit{where} clause, when necessary, to write abbreviations. The provisos for applying a refactoring rule are listed in the \textit{provided} clause of the rules. Here we present two examples of refactoring rules. The first one allows extracting and inlining a method, and is used to illustrate the mechanisation of a refactoring proof in CafeOBJ. The second one introduces set and get methods and is used to show an implementation of a refactoring rule in CafeOBJ, as well as its application in an arbitrary example.
3.1. Extract and Inline Method

Rule 1, when considered from left to right, coincides with the refactoring Extract Method presented by Fowler [10, p. 110], whereas the application in the reverse direction corresponds to the refactoring Inline Method [10, p. 117]. When applied from left to right, it turns a command \( c_2 \), which is present in a method \( m_1 \), into a new method \( m_2 \). Occurrences of the command \( c_2 \) in the original method \( m_1 \) are replaced by calls to the newly introduced method.

\[ \text{Rule 1} \ \langle \text{Extract/InlineMethod} \rangle \]

\[
\begin{align*}
\text{class} \ A & \text{ extends } C \\
\text{ads}; \\
\text{meth} \ m_1 & = (pds_1 \bullet c_1[c_2[a]]) \\
\text{mts} & \\
\text{end} \\
=_{cds,c} \\
\text{class} \ A & \text{ extends } C \\
\text{ads}; \\
\text{meth} \ m_1 & = (pds_1 \bullet c'_1) \\
\text{meth} \ m_2 & = (pds_2 \bullet c_2[\alpha(pds_2)]) \\
\text{mts'} & \\
\text{end}
\end{align*}
\]

where

\[
\begin{align*}
c'_1 & = c_1[\text{self}.m_2(a)/c_2[a]] \\
mts & = mts'[c_2[a]/\text{self}.m_2(a)]
\end{align*}
\]

\( a \) is the finite set of free variables of command \( c_2 \), not including attributes of class \( A \);

provided

\[
\langle \rightarrow \rangle \ (1) \ \text{Variables in} \ a \ \text{have basic types}; \ (2) \ \text{Parameters in} \ pds_2 \ \text{must have the same types as those of} \ a; \\
\langle \rightarrow \rangle \ m_2 \ \text{is not declared in} \ mts \ \text{nor in any superclass or subclass of} \ A \ \text{in} \ cds; \\
\langle \leftarrow \rangle \ m_2 \ (1) \ \text{is not recursively called}; \ (2) \ B.m_2 \ \text{does not appear in} \ cds, \ c \ \text{nor in} \ mts, \ \text{for any} \ B \ \text{such that} \ B \leq A.
\]

The meta-variable \( a \) represents a list containing the free variables that appear in the command \( c_2 \) of method \( m_1 \) which are not attributes of class \( A \). On the left-hand side of this rule, \( c_1[c_2[a]] \) represents the command \( c_1 \), which may have occurrences of the command \( c_2 \), which in turn may have occurrences of \( a \). The class \( C \) that appears in the \textit{extends} clause, and in the others that follow, is present in the sequence of class declarations \( cds \) or is the class \textit{object}.

On the right-hand side of this rule, the method call \( \text{self}.m_2(a) \) replaces the occurrences of the command \( c_2[a] \) in the command \( c_1 \) of method \( m_1 \); the resulting command is \( c'_1 \). In the command \( c_2 \) of method \( m_2 \), the variables indicated by \( a \) are replaced with the parameters \( \alpha(pds) \), where \( \alpha(pds) \) denotes the list of parameter names declared in \( pds \). If a variable is only read in \( c_2 \), it could be passed as a value argument. A variable that is only written could be passed as a result argument. Variables that are both read and written must be passed as both value and result arguments. The free variables, represented by \( a \), that are passed as arguments in the call to \( m_2 \) may involve all arguments of method \( m_1 \) as well as local variables that appear in \( c_2 \). Of course, all free variables that appear in \( c_2 \) must become parameters of \( m_2 \).

To apply this rule from left to right, the method \( m_2 \) must be new: not declared in a superclass of \( A \), in \( A \) itself, nor in any of its subclasses. We also require the types of variables in \( a \) to be basic, and parameters in \( pds_2 \) must have the same types as those of \( a \). Applying this rule from right to left replaces method calls to \( m_2 \) with the body of this method and removes \( m_2 \) from class \( A \). To apply this rule in this direction, there must be no recursive calls in the method.
Before presenting the derivation of this refactoring, we need to introduce two results, proved in \[8\].

**Lemma 1** \((\text{method call elimination-self})\) replaces the call for a parameterised command which is the body of a local method with a call to the method itself.

Consider that the following class declaration

\[
\text{class } C \text{ extends } D
\]

\[
\text{meth } m \triangleq pc
\]

\[
\text{end}
\]

is included in \(cds\). Then \(cds, C \triangleright pc(e) = \text{self}.m(e)\)

\[\square\]

**Derivation.** We begin the derivation of Rule 1 with the class \(A\) that appears on the left-hand side of Rule 1. We assume that the required conditions for the application of this rule hold. However, it is not possible to have a fully general derivation, in the sense that we do not define a parameter for the method being extracted, rather we have to introduce a parameterised command along with a specific parameter passing mechanism in order to be able to conduct the proof. Moreover, in the derivation we consider that the set of variables \(a\) has just one element, a variable named \(a\). Also, we consider that such variable is only read, implying that the value parameter passing mechanism is the one applicable for the parameter to be defined in the extracted method. The derivation for a result parameter is similar.

By using Law 6 \((\text{method elimination})\), from right to left, we introduce the following method \(m_2\) in class \(A\).

\[
\text{meth } m_2 \triangleq (\text{val } arg : T \bullet c_2[\text{arg}])
\]

This requires that the method to be introduced is not declared in the superclass of the class to which the law is applied, in the class itself nor in any of its subclasses. We can apply Law 6 \((\text{method elimination})\) because the refactoring rule requires the same conditions to be satisfied.

The command \(c_2\) that is present in the method \(m_1\) also appears in the method \(m_2\). Our aim is to introduce a call to the method \(m_2\). We introduce a parameterised command that is applied to the argument \(a\) by using Lemma 2 \((\text{pcom value} - \text{argument})\). The occurrences
of variable \( a \), in command \( c_2 \), are replaced with the parameter \( arg \). This lemma requires that the argument that is applied to the parameterised command is not a method call target. Consequently, there is no sense in having as argument an object on which we cannot call a method inside the parameterised command. For this reason \( a \) is restricted to have basic type. By the application of Lemma 2 \((\text{pcom value} - \text{argument})\), we obtain the following parameterised command.

\[
(\text{val } arg : T \cdot c_2[\text{arg}]) (a)
\]

The parameterised command that occurs in command \( m_1 \) is the same as that of method \( m_2 \). By using Lemma 1 \((\text{method call elimination-self})\), from left to right, we introduce in \( m_1 \) a call to \( m_2 \), obtaining the following class.

\[
= \text{Lemma 1 } (\text{method call elimination-self})
\]

\[
\begin{array}{l}
\text{class } A \\
\quad \text{ads};
\end{array}
\]

\[
\begin{array}{l}
\quad \text{meth } m_1 \triangleq (pds_1 \cdot c_1[\text{self.m2}(a)])
\end{array}
\]

\[
\begin{array}{l}
\quad \text{meth } m_2 \triangleq (\text{val } arg : T \cdot c_2[\text{arg}])
\end{array}
\]

\[
\text{mts}
\]

\[
\text{end}
\]

This completes the derivation for a value parameter. The derivation for an arbitrary number of parameters is similar, but it involves the application of Law \(\langle \text{pcom merge} \rangle \) [8], which merges two parameterised commands into one.

### 3.2. Self Encapsulate Field

Rule 2 \(\langle \text{SelfEncapsulateField} \rangle\) introduces get and set methods for an attribute \( x \) declared in the class \( A \). The derivation of this rule can be found in [8].

#### Rule 2 \(\langle \text{SelfEncapsulateField} \rangle\)

\[
\begin{array}{ll}
\text{class } A \text{ extends } C \\
\quad \text{pri } x : T ;
\end{array}
\]

\[
\begin{array}{l}
\quad \text{ads}_{a};
\end{array}
\]

\[
\begin{array}{l}
\quad \text{meth } m_1 \triangleq (pds_1 \cdot c_1[\text{self.x}],
\quad \text{self.x} := \text{exp}_2)
\end{array}
\]

\[
\text{mts}_a
\]

\[
\text{end}
\]

\[
\begin{array}{ll}
\text{class } A \text{ extends } C \\
\quad \text{pri } x : T ;
\end{array}
\]

\[
\begin{array}{l}
\quad \text{ads}_{a};
\end{array}
\]

\[
\begin{array}{l}
\quad \text{meth } m_1 \triangleq (pds_1 \cdot c'_1)
\quad \text{meth } \text{getX} \triangleq (\text{res } arg : T \cdot arg := \text{self.x})
\quad \text{meth } \text{setX} \triangleq (\text{val } arg : T \cdot \text{self.x} := \text{arg})
\end{array}
\]

\[
\text{mts}'_a
\]

\[
\text{end}
\]

where

\[
\begin{array}{l}
\quad c'_1 \triangleq c_1[\text{var } aux : T \cdot \text{self.getX(aux)}; \text{le}_1 := \text{exp}_1[\text{aux}] \text{ end},
\quad \text{self.setX} (\text{exp}_2) / \text{le}_1 := \text{exp}_1[\text{self.x}], \text{self.x} := \text{exp}_2]
\end{array}
\]

\[
\begin{array}{l}
\quad mts'_a \triangleq mts_a[\text{var } aux : T \cdot \text{self.getX(aux)}; \text{le}_1 := \text{exp}_1[\text{aux}] \text{ end},
\quad \text{self.setX} (\text{exp}_2) / \text{le}_1 := \text{exp}_1[\text{self.x}], \text{self.x} := \text{exp}_2]
\end{array}
\]

provided

\[
\begin{array}{l}
\quad (\rightarrow) \text{getX is not declared in any superclass or subclass of } A \text{ in } \text{cds};
\quad \text{setX is not declared in any superclass or subclass of } A \text{ in } \text{cds};
\quad (\rightarrow) \text{le.getX and le.setX do not appear in } mts'_a, \text{cds or } c, \text{ for any } le \text{ such that } le \leq A.
\end{array}
\]
In method \( m_1 \) on the left-hand side of the rule, the attribute \( x \) appears in the expression \( \text{exp}_1 \) and there is also an assignment to this attribute. On the right-hand side of the rule, the occurrence of \( \text{self}.x \) in expression \( \text{exp}_1 \) is replaced by the local variable \( \text{aux} \) declared in \( m_1 \). This variable receives the result of the call to method \( \text{getX} \). The assignment is accomplished by a call to method \( \text{setX} \), passing by value the expression \( \text{exp}_2 \). These changes also occur in \( \text{mts}_a \). To apply this rule from left to right, the methods \( \text{getX} \) and \( \text{setX} \) cannot be declared in the superclass of \( A \), in \( A \) itself, nor in any of its subclasses. To apply this rule in the reverse direction, the methods \( \text{getX} \) and \( \text{setX} \) cannot be called in \( \text{cds}, \text{c}, \text{or} A \).

In this rule, just one attribute is considered. To encapsulate fields inside a class implies in the application of this rule the same number of times as the number of attributes to be encapsulated.

4. CafeOBJ

CafeOBJ [17] is a new generation algebraic specification and programming language. As a successor of the OBJ family (OBJ1, OBJ2, OBJ3) [11], it inherits features such as: powerful typing system with sub-types; sophisticated module composition system, featuring several kinds of imports; parameterised modules; views for instantiating parameters and the module expressions, among other issues. CafeOBJ implements new paradigms, such as rewriting logic and hidden algebra, as well as their combinations. It is mainly used for system specification, formal verification of specifications, rapid prototyping, programming and automatic theorem proving.

CafeOBJ is chosen due to some characteristics, among them, the availability of documentation, the facility in the use of the reduction mechanism, the possibility of applying the rules in two ways and in subterms of the term to be reduced.

```plaintext
module PEANO-NAT {
  imports {
    protecting (NAT)
    protecting (INT)
  }
  signature {
    [ Peano-Nat, Nat < Int ]
    op 0 : -> Peano-Nat
    op s : Peano-Nat -> Peano-Nat
    op (_+_): Peano-Nat Peano-Nat -> Peano-Nat
  }
  axioms {
    var N : Peano-Nat
    eq 0 + N = N.
    eq s(M:Peano-Nat) + N = s(M + N).
  }
}
```

**Figure 1. The module PEANO-NAT in CafeOBJ.**

A module in CafeOBJ has the syntax defined by `module < mod_id > mod_elem*`, where `< mod_id >` is the name of the module and `mod_elem` is an element of the module. A module element is either an import declaration, a sort declaration, an operator declaration, a record declaration, a variable declaration, an equation declaration or a transition declaration. These elements are structured into three main parts. The first part, imports, specifies which modules should be imported, that is, inherited. There are three forms of importing modules: protecting (the imported module can not be changed), extending (the imported module can be extended, but the original description remains unchanged) and using
(the imported module can be extended or can change the original description). The second part, signature, declares sorts, operators, records and subsorts used by the module. Finally, axioms includes declaration of variables, equations and transitions and expresses the behavior of the module.

To illustrate a module description in CafeOBJ, consider the example of Figure 1, which defines the Peano notation for natural numbers. The module PEANO-NAT inherits the sorts and the operators defined in modules INT and NAT. Section signature declares the sorts Peano-Nat, Nat and Int. The symbol < means that Nat is a subsort of Int. The zero constant, the s and the (infix) + operators are introduced by op. The behaviour of the + operator is given by two expressions, introduced by eq.

5. Proving and Implementing Refactoring Rules in CafeOBJ

The mechanisation of the proofs of refactoring rules comprises two steps: the implementation of the ROOL grammar and laws (Section 5.1) and the automatic derivation of the rules from the laws already implemented (Section 5.2). After proved, the refactoring can be implemented as a rule and used to perform program transformation (Section 5.3).

5.1. The Implementation of ROOL Grammar and Laws

To implement the grammar, a module ROOL-GRAMMAR is defined, including the language operators and constructors. Every expression or command in ROOL is thus defined by using these operators and constructors. The full implementation of the grammar comprises 92 operators and a small fragment of this module is depicted in Figure 2, where the lines are numbered for didactic reasons. This facility is used in the remainder of this paper.

```
module ROOL-GRAMMAR {
  [ ClassName PrimitiveType < Type ]
  op bool : -> PrimitiveType .
  op int : -> PrimitiveType .
  op char : -> PrimitiveType .
  ...
  op _is_ : Expression ClassName -> Bool .
  op (_) : ClassName Expression -> Expression .
  op _._ : Expression Variable -> Expression .
  ...
  op ,_ : Variable VariableList -> VariableList { r-assoc } .
  ...
  op _[]_ : GuardedCommandList GuardedCommandList -> GuardedCommandList { ... } .
  ...
  op (if_fi) : GuardedCommandList -> Command .
  ...
  op (val_:_) : VariableList TypeList -> ParDecl .
  ...
  op (pri_:;) : VariableList TypeList -> ParDecl .
  ...
  op class_extends__end : ClassName ClassName ClassBody -> ClassDecl {prec: 40} .
  op class__end : ClassName ClassBody -> ClassDecl {prec: 40} .
  ...
  op (pri_:;) : Variable Type -> Attr { prec: 0 } .
  ...
  op (meth_"-"_) : MethodName ParCommand -> Meth { prec: 0 } .
  ...
}
```

Figure 2. A fragment of the module ROOL-GRAMMAR in CafeOBJ.

The module is introduced by the reserved word module. In Line 2 the sorts ClassName and PrimitiveType are declared as subsorts of Type. Lines 3 to 5 define
some primitive types of ROOL. The type test, cast and the attribute selection operators are defined in lines 7 to 9, respectively. For example, the type test operator receives an expression and a name of a class, and returns true whenever the type of the expressions is equal to the given class name. After that, some auxiliary operators are defined, such as the one in Line 11, which creates lists of variables. As an example of command implementation, the guarded command is depicted in lines 13 and 14, and the alternation in Line 16. Parameters declaration is implemented as in Line 18, which shows the call-by-value mechanism. Class declarations, considering superclasses or not, are shown in lines 20 and 21, respectively. In Line 20 the second ClassName corresponds to the superclass. Private attributes are declared using the operator in Line 23 and Line 25 introduces a method declaration. The reserved word \texttt{prec} introduces the precedence of the operator.

\begin{verbatim}
1 module ROOL-LAWS {
2   protecting (ROOL-LEMMAS)
3   protecting (ROOL-UTILS)
4   ...
5 var x : Variable .
6 var T : Type .
7 var c : Command .
8 op replaceAndAppend : GuardedCommandList LeftExp Expression -> GuardedCommandList .
9 eq replaceAndAppend (p -> c, le, e) = (p [ e / le ]) -> (le := e ; c) .
10 eq replaceAndAppend ((p -> c) [] gcl , le, e) = 
11 (p [ e / le ]) -> (le := e ; c) [] replaceAndAppend(gcl, le, e) .
12 eq [=:=<>-right-dist] : le := e ; if gcl fi = if replaceAndAppend(gcl, le, e) fi .
13 ...
14 eq [var-elim] : var x : T @ c end = c .
15 ...
16 eq [var-:=final-value] : var x : T @ c ; x := e end = var x : T @ c end .
17 ...
18 trans [var-:=initial-value] : var x : T @ c end => var x : T @ x := e ; c end .
19 ...
20 eq [order-independent-assignment] : x := el ; y := e2 = y := e2 ; x := el .
21 ...
22 eq [method-elimination] : 
23 class C extends D ads meth m ^= pc ops end = 
24 class C extends D ads ops end .
25 eq [move-original-method-to-superclass] : 
26 class B extends A ads ops end class C extends B ads’ meth m ^= pc ops’ end = 
27 class B extends A ads meth m ^= pc ops end class C extends B ads’ ops’ end .
28 ...
29 }
\end{verbatim}

Figure 3. A fragment of the module \texttt{ROOL-LAWS} in CafeOBJ.

To implement the laws, a module \texttt{ROOL-LAWS} is defined, which imports modules \texttt{ROOL-UTILS} and \texttt{ROOL-LEMMAS}. The former imported module includes some auxiliary operators, like the constructors for integer lists, necessary to implement the laws, whereas the latter one includes some partial results, used to derive the refactoring rules. Each law is defined as a equation in CafeOBJ. The module \texttt{ROOL-LAWS} comprises the implementation of seventy laws, the module \texttt{ROOL-UTILS} the implementation of seven auxiliary operators and the module \texttt{ROOL-LEMMAS} the implementation of six intermediary results, including lemmas 1 and 2 of Section 3.

To illustrate this module, Figure 3 shows a small fragment of the generated code. Basically some variables used in the laws equations are introduced, as the ones in Lines 5 to 7. After that, the command laws are implemented. Lines 8 through 12 depicts the implementation of Law 1 ($\langle= -- <> right dist \rangle$), given in Section 2. An extra operator, \texttt{replaceAndAppend}, which is responsible for inserting the substitution and moving the assignment, is necessary, as
the law is applied in a list of guarded commands with an arbitrary number of elements and therefore cannot be directly implemented. Lines 14, 16 and 20 show, respectively, the implementation of Law 2 \(\langle \text{var elim} \rangle\), 3 \(\langle \text{var} \leftarrow \text{final value} \rangle\) and, 5 \(\langle \text{order independent assignment} \rangle\), explained in Section 2. The implementation of these laws is immediate where the symbol “@” means “•” in the formal definition. Law 4 \(\langle \text{var} \leftarrow \text{initial value} \rangle\) is implemented in Line 18, as a transition by using the reserved word \text{trans}, as Law 4 is a refinement and not an equality. The laws of classes are also implemented in this module. For example, lines 22 to 24 introduce Law 6 \(\langle \text{method elimination} \rangle\), whereas lines 25 to 27 define Law 7 \(\langle \text{move original method to superclass} \rangle\). The implementation of both laws follows directly from the formal definition presented in Section 2. For now we have not implemented the verification of the conditions that must be satisfied to apply some laws and we suppose that they are true.

5.2. Proving a Refactoring Rule

In this section we present the derivation of the Extract/Inline Method refactoring, using CafeOBJ. The manual proof of this refactoring, given by Cornélio in [8], is reproduced in Section 3.

```plaintext
start class A:ClassName extends C:ClassName ads:AttrS meth m:MethodName
\& (pds1:ParDecls @ c1:Command [ c2:Command ]) mts:MethS end .
apply -.method-elimination with m = m2:MethodName, pc = (val arg:Variable : T:Type @ c2:Command [ arg:Variable ])
within term .
result class A:ClassName extends C:ClassName ads:AttrS meth m2:MethodName
\& (val arg:Variable : T:Type) @ (c2:Command [ arg:Variable ])
( meth m:MethodName ^= (pds1:ParDecls @ (c1:Command [ (c2:Command [ a:Variable ])])) mts:MethS) end : ClassDecl
apply .lemma2 with vl = arg, x = a:Variable at (3 2 1 2 2 2) .
result class A:ClassName extends C:ClassName ads:AttrS meth m2:MethodName
\& (val arg:Variable : T:Type) @ (c2:Command [ arg:Variable ])
( meth m:MethodName ^= ((val arg:Variable : T:Type) @ (c2:Command [ arg:Variable ]))) mts:MethS) end : ClassDecl
apply .lemma1 with m = m2 at (3 2 1 2 2 2) .
result class A:ClassName extends C:ClassName ads:AttrS meth m2:MethodName
\& ((val arg:Variable : T:Type) @ (c2:Command [ arg:Variable ])) mts:MethS) end : ClassDecl
```

Figure 4. Automatic derivation of Rule 1 \(\langle\text{Extract/Inline Method}\rangle\).

Figure 4 shows the automatic derivation of that refactoring. Firstly, the left-hand side of the rule is introduced by the command start. After that, each law and lemma used in the derivation process is applied, using the command apply. The results achieved after each application follows the reserved word result. By comparing this figure with the proof given in Section 3, observe that the first result shows the transformation of the left-hand side of the rule after applying Law 6 \(\langle\text{method elimination}\rangle\). To perform the application of this law the generic variables \(m\) and \(pc\) are instantiated with the corresponding terms on the left-hand side of the refactoring. The result achieved is very similar with the one of Section 2. The major difference is that each variable is associated with its type. After that, lemmas 1 and 2 of module ROOL-LEMMAS reduce this partial result to the right-hand side. The numbers that appear after the reserved word at are parameters used to capture the term in which the transformation is applied. For example, Lemma 2 is applied to transform the term \(c2[a]\) inside the body of method \(m\). The rest of the refactoring rules proposed in [8] are reduced in a similar way.
5.3. Implementation and Application of a Refactoring Rule

After proving the refactoring rules, they can be implemented as equations in CafeOBJ. To implement Rule 2, in CafeOBJ, a module ROOL-SELFENCAPSULATE is defined, whose fragment is depicted in Figure 5.

```
1 module ROOL-SELFENCAPSULATE {
2   protecting (ROOL-GRAMMAR)
3   ...
4   op selfencap : ClassDecl Variable MethodName MethodName -> ClassDecl .
5   op getType : AttrS Variable -> Type .
6   op replaceOnMethod : MethS Variable MethodName MethodName Type -> MethS .
7   ops replaceOnCommand replaceOnAssignmentList replaceOnAssignment .
8   op replaceOnExpression : Expression Variable -> Expression .
9   op containsAttribute : Expression Variable -> Bool .
10  ops fromListToConcat fromConcatToList : Command -> Command .
11  ...
12  eq selfencap( class A extends B as ms end , x , getX , setX ) =
13    class A extends B
14    as
15    (meth getX "= ( (res x : getType(as, x)) @ (x := self. x) ) )
16    (meth setX "= ( (val x : getType(as, x)) @ (self. x := x) ) )
17    replaceOnMethod(ms, x, getX, setX, getType(as, x))
18    end .
19  ...
20  eq getType( (pri a : tipo:Type ;) as , a ) = tipo .
21  ...
22  eq replaceOnMethod( (meth m "= (pd @ c)) ms, x,getX, setX, T ) =
23    (meth m "= ( pd @ replaceOnCommand(c, x, getX, setX, T) )
24    replaceOnMethod(ms, x, getX, setX, T) .
25  ...
26  eq replaceOnCommand( leL := eL , x, getX, setX, T) =
27    fromConcatToList(replaceOnAssignmentList(
28       fromListToConcat(leL := eL), x, getX, setX, T) ) .
29  ...
30  eq replaceOnAssignmentList( le := e ; c , x, getX, setX, T) =
31    replaceOnAssignment( le := e , x, getX, setX, T)
32    ; replaceOnAssignmentList( c , x, getX, setX, T) .
33  ...
34  ceq replaceOnAssignment( le := e , x, getX, setX, T) =
35    var x : T @ self. getX < x > end ; le := replaceOnExpression(e, x)
36    if containsAttribute(e, x) .
37  ...
38  eq replaceOnAssignment( self. x := e , x, getX, setX, T) = self. setX < e > .
39  ...
40  eq replaceOnExpression( e <= e2 , x ) = replaceOnExpression(e, x) <=
41    replaceOnExpression(e2, x) .
42  ...
43  eq replaceOnExpression( self. x , x ) = x .
44  eq replaceOnExpression( e , x ) = e .
45  ...
46  }
```

Figure 5. A fragment of the module ROOL-SELFENCAPSULATE in CafeOBJ.

This module imports all operators from ROOL-GRAMMAR. Lines 4 to 11 define all operators necessary to implement this rule. selfencap is the main operator, which receives the class declaration to be refactored, the attribute and the names of the newly introduced get and set methods. The getType operator simply returns the type of the given attribute present in the list of attributes as. Operators fromListToConcat and fromConcatToList make the term simpler for the applications of other operations. Basically, they turn an assignment of a list of variables to a list of expressions into a concatenation of assignments, with each variable assigned to its corresponding expression in the list. The function containsAttribute
receives an expression and an attribute name and returns true if such attribute occurs inside the given expression. The function replaceOnMethod replaces inside the given methods all the occurrences of the given attribute by its get or set methods. All the remaining operators behave in a similar way.

```plaintext
start selfencap( selfencap( class IntRange pri_low : int ; pri_high : int ; meth includes ^= ( val arg : int ; res ret : bool @ ( ret := ((arg >= self. _low) && (arg <= self. _high)) ) ) meth grow ^= ( val factor : int @ ( self. _high := self. _high * factor ) ) end , _high, getHigh , setHigh ), _low, getLow , setLow ).
apply red at term .
result class IntRange:ClassName pri _low:Variable : int:PrimitiveType ; pri _high:Variable : int:PrimitiveType ; meth getLow:MethodName ^= ((res _low:Variable : int) @ (low := self. _low)) ( meth setLow:MethodName ^= {{ val _low : int @ (self. _low := _low()) (meth getHigh:MethodName ^= {{ val _high : int @ (self. _high := _high()) ) ( meth setHigh:MethodName ^= {{ val _high : int @ (self. _high := _high()) (meth includes:MethodName ^= ((( val arg:Variable : int:PrimitiveType) ; ( res ret:Variable : bool:PrimitiveType)) @ ( var _high : int @ (self. getHigh < _high >) end ; ( var _low : int @ (self. getLow < _low >) end ; ret:Variable := (arg:Variable := _low()) && (arg:Variable <= _high()))) meth grow:MethodName ^= ((( val factor:Variable : int:PrimitiveType) @ (var _high : int @ (self. getHigh < _high >) end ; self. setHigh < ( _high * factor:Variable : Variable) > ))) ) end : ClassDec
```

Figure 6. Application of the implementation of Rule 2 (Self Encapsulate Field).

The effective application of the implemented rule in an arbitrary example, considering that the provisos are satisfied, is shown in Figure 6. The example introduced by the start command is extracted from Fowler [10], and reproduced in Figure 7. The class declaration must be supplied to CafeOBJ wrapped in the operator selfencap, along with the name of the attribute to be encapsulated, as well as the names of the get and set methods to be created. Observe that in the transformed program all assignments to the encapsulated attributes use set methods. Moreover, where these attributes are only used, a get method is introduced.

```plaintext
class IntRange
  pri _low : int;
  pri _high : int;
  meth includes = (val arg : int; res ret : bool •
    ret := (arg >= self._low) && (arg <= self._high))
  meth grow = (val factor : int • self._high := self._high * factor)
end
```

Figure 7. Class IntRange, used in the application.

6. Conclusions

This paper illustrates how rewriting systems, in particular CafeOBJ, can be used to mechanise refactoring proofs, in the algebraic style developed by Cornélio in [8]. In this way, this work complements the formal rigour of that approach. Moreover, the grammar of ROOL as well as its programming laws are fully implemented. Nevertheless we have not implemented all refactorings proposed in [8], we dealt with a significant subset of them. Each refactoring is implemented as a rule in CafeOBJ, allowing the construction of a transformation environment.
The use of CafeOBJ was of great value to this work. The implementation of ROOL grammar and laws imposed no problems. Thus, this work could also be regarded as a contribution to the CafeOBJ community, as it could be considered as a programming transformation case study, in the context of this rewriting system. On the other hand, for the refactoring community, this work suggests that rewriting systems could be considered as supporting tools to construct refactoring environments.

The majority of refactoring rules and laws of classes includes side conditions. When deriving the refactoring rules, we considered that these conditions are satisfied. Nevertheless, to construct a refactoring environment using a rewriting system such as CafeOBJ, we need to provide an efficient way to check the validity of these conditions in an arbitrary program, before applying the reduction. This is a challenge task, as many condition that appear in refactoring rules rely on information provided by the ROOL type system. In order to implement these conditions in CafeOBJ, we also need to describe the ROOL type system or, at least, to obtain type information from the program. This is our immediate future work.

The case study introduced in this paper is simple enough to validate the work. To develop some realistic and more complex case studies is a direction of future work.

Acknowledgements. This work is partially supported by the Brazilian research agency CNPq.

References


