Using CafeOBJ to Mechanise Refactoring Proofs and Application

Antonio Carvalho Júnior
Departamento de Ciência da Computação e Estatística
Universidade Federal de Sergipe (UFS)
CEP 41900-000 – São Cristóvão – SE, Brazil

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

Márcio Cornélio
Departamento de Sistemas Computacionais
Escola Politécnica – Universidade de Pernambuco (UPE)
Rua Benfica, 455 – 50750-410, Recife – PE, Brazil

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.

Keywords: Rewriting Systems, Refactorings, CafeOBJ

1 Introduction

Changes are common to software. Many practitioners recognize that changing an object-oriented software is easier than conventional ones [15]. 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 [10,21] several refactorings are introduced in a rather informal way. Opdyke [19] 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 [18]. 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 [14] use Maude [7] to implement a reduction strategy for object-oriented languages and Silva et al [20] 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 refactorings. In Section 3 we present three 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 implementation of two refactorings in CafeOBJ. A case study developed to validate the work is described in Section 6. Finally, Section 7 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 [16,17]. The semantics of ROOL, as usual for refinement calculi, is based on weakest preconditions. The imper-
ative constructs of ROOL are based on the language of Morgan’s refinement calculus [16], 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 [16].

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

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

Classes are related by single inheritance, which is indicated by the clause $extends$. The class $object$ is the default superclass of classes. So, the $extends$ clause could have been omitted in the declaration of $Employee$. The class $Employee$ includes a private attribute named $salary$; this is indicated by the use of the $pri$ qualifier. Attributes can also be protected ($prot$) or public ($pub$). 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 $val x : T \bullet c$ or $res x : T \bullet c$, which correspond to the call-by-value and call-by-result parameter passing mechanisms, respectively. For instance, the method $getSalary$ has a result parameter $r$, whereas $setSalary$ has a value parameter $s$. Initialisers are declared by the $new$ clause.

ROOL also includes specification constructs from Morgan’s refinement calculus [16], like the specification statement. The laws of ROOL, mainly those related to object-oriented features, address context issues. We use $cds_1 =_{cds, c} cds_2$, where $cds$ is a context of class declarations $cds_1$ and $cds_2$, and $c$ is the main command to denote the equivalence of set of class declarations $cds_1$ and $cds_2$. This notation is just an abbreviation for the program equivalence $cds_1 cds \bullet c = cds_2 cds \bullet c$, which is formalised in [4,3]. As an example, in what follows we present some laws of ROOL. We write ‘$(\rightarrow)$’ when some conditions must be satisfied for the application of the law from left to right. We use ‘$(\leftarrow)$’ to indicate the conditions that are necessary for applying a law from right to left. By writing ‘$(\leftrightarrow)$’ 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 $⟨:= − \triangleleft right dist⟩$ distributes an assignment over a list of conditional commands, provided $e$ is total.

Law 1 $⟨:= − \triangleleft right dist⟩$ If $e$ is total, then

$$le := e; \ if \ [i \bullet \psi_i \rightarrow c_i \ fi = \ if \ [i \bullet \psi_i[e/le] \rightarrow (le := e; c_i) \ fi$$
Law 2 \langle \text{var elim} \rangle \) eliminates a variable that does not appear in \( c \) and Law 3 \langle \text{var- := final value} \rangle \) removes an assignment at the very end of its scope, as this assignment introduces no effect.

**Law 2 \langle \text{var elim} \rangle** \ If \( x \) is not free in \( c \), then \( \text{var } x : T \cdot c \text{ end} = c \ □ \)

**Law 3 \langle \text{var- := final value} \rangle** \ If \( x \) is not free in \( c \), then

\[
\text{var } x : T \cdot c; \ x := e \text{ end} = \text{var } x : T \cdot c \text{ end} □
\]

Law 4 \langle \text{var- := initial value} \rangle \) assigns an expression to a variable before its first use and Law 5 \langle \text{order independent assignment} \rangle \) reorders assignments, if they are not data dependent. The symbol \( □ \) means the standard refinement symbol.

**Law 4 \langle \text{var- := initial value} \rangle** \ If \( e \) has no type errors, then

\[
\text{var } x : T \cdot c \text{ end} □ \)

**Law 5 \langle \text{order independent assignment} \rangle** \ If \( x \) and \( y \) are not free in \( e_2 \) and \( e_1 \), respectively, then \( (x := e_1; y := e_2) = (y := e_2; x := e_1) □ \)

Law 6 \langle \text{method elimination} \rangle , allows the remotion of a method from a class if it is not called by any classes in \( \text{cds} \), 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 \langle \text{method elimination} \rangle**

\[
\begin{array}{c}
\text{class } C \text{ extends } D \\
\text{ads} \\
\text{meth } m \equiv pc \\
\text{mts} \\
\text{end}
\end{array}
=_{\text{cds,c}}
\begin{array}{c}
\text{class } C \text{ extends } D \\
\text{ads} \\
\text{mts} \\
\text{end}
\end{array}
\]

provided

\( (→) \) \( B.m \) does not appear in \( \text{cds}, c \) nor in \( \text{mts} \), for any \( B \) such that \( B ≤ C \).

\( (←) \) \( m \) is not declared in \( \text{ops} \) nor in any superclass or subclass of \( C \) in \( \text{cds} \).

\( □ \)

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

**Law 7 \langle \text{move original method to superclass} \rangle**
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 where and provided clauses. We use the where clause, when necessary, to write abbreviations. The provisos for applying a refactoring rule are listed in the provided clause of the rules. Here we present three 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. The last one moves a method from a class to another, and is used in the case study developed here.

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.
Rule 1 \textit{(Extract/InlineMethod)}

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

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

\begin{itemize}
\item[a a is the finite set of free variables of command \(c_2\), not including attributes of class \(A\);
\item[provided]
\item[(\(\leftrightarrow\)) (1) Variables in \(a\) have basic types; (2) Parameters in \(pds_2\) must have the same types as those of variables in \(a\);
\item[(\(\rightarrow\)) \(m_2\) is not declared in \(mts\) nor in any superclass or subclass of \(A\) in \(cds\);
\item[(\(\leftarrow\)) \(m_2\) (1) is not recursively called; (2) \(B.m_2\) does not appear in \(cds,c\) nor in \(mts\), for any \(B\) such that \(B \leq A\).
\end{itemize}

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 \textbf{extends} clause, and in the others that follow, is present in the sequence of class declarations \(cds\) or is the class \textbf{object}.

On the right-hand side of this rule, the method call \texttt{self.m2(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 $m_2$ (note that this is not a limitation in practice because $c_2$ can be a recursive command of the form $\text{rec } X \bullet c_3 \text{ end}$). Also, the method $m_2$ cannot be called in $cds, c$ nor in $mts$.

Before presenting the derivation of this refactoring, we need to introduce two results, proved in [8].

Lemma 1 (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. The symbol $\triangleright$ in $cds, C \triangleright c$ asserts that $c$ is a command that can appear in the body of a method in class $C$, in the context of the sequence of class declarations $cds$.

**Lemma 1 (method call elimination-self)** Consider that the following class declaration

```
class C extends D
  ads
  meth m ≡ pc
  mts
end
```

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

Lemma 2 (pcom value−argument) transforms a single command into a parameterised one with a value argument.

**Lemma 2 (pcom value−argument)** $c = (\text{val } vl : T \bullet c[vl/x])(x)$, provided $vl$ is not free in $c$, $x$ is not on the left-hand side of assignments, it is not a result argument, $x$ does not occur in the frame of specification statements in $c$, $x$ is not a method call target, and $x \neq \text{error}$

**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 ⟨method elimination⟩, from right to left, we introduce the following method $m_2$ in class $A$.

```
meth m_2 ≡ (\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 ⟨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 (pcom value – 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 (pcom value – argument), we obtain the following parameterised command.

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

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

= Lemma 1 (method call elimination-self)

$$\text{class } A$$
$$\text{ads;}$$
$$\text{meth } m_1 \equiv (pds_1 \bullet c_1[\text{self}.m_2(a)])$$
$$\text{meth } m_2 \equiv (\text{val } arg : T \bullet c_2[\text{arg}])$$
$$\text{mts}$$
$$\text{end}$$

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 ⟨pcom merge⟩ [8], which merges two parameterised commands into one.

### 3.2 Self Encapsulate Field

Rule 2 ⟨SelfEncapsulateField⟩ introduces get and set methods for an attribute $x$ declared in the class $A$. The derivation of this rule can be found in [8].

In method $m_1$ on the left-hand side of the rule, the attribute $x$ appears in the expression $exp_1$ 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 $exp_1$ is replaced by the local variable $aux$ declared in $m_1$. This variable receives the result of the call to method $getX$. The assignment is accomplished by a call to method $setX$, passing by value the expression $exp_2$. These changes also occur in $\text{mts}_a$. To apply this rule from left to right, the methods $getX$ and $setX$ 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 $getX$ and $setX$ cannot be called in $\text{cds}$, $c$, 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.
3.3 Move Method

Moving methods between classes helps to make them simpler and clearer, improving legibility. Rule 3 ⟨Move Method⟩ allows us to move a method that uses more features of another class than the class on which it is defined.

On the left-hand side of this rule, method \( m_1 \) of \( A \) reads and writes to the attribute \( x \) of \( B \) by means of its get and set methods. The method calls occur inside the command \( c_1 \) of the method \( m_1 \) and have as target the attribute \( b \). The variable \( aux \) holds the result of the call to the method \( getX \) of \( B \). Such variable may occur on the right-hand side of an assignment to an expression \( le \). In the command \( c_1 \), there may also be occurrences of the parameters that appear in \( pds \), which is indicated by \( \alpha(pds) \), where \( \alpha(pds) \) gives the list of parameter names declared in \( pds \). For instance, if \( pds \) is the declaration \( \langle \text{val } x : T; \text{res } y : T \rangle \), the list given by \( \alpha(pds) \) is \( x; y \). The class \( B \) on the left-hand side declares no method like \( m_1 \) of the class \( A \).

On the right-hand side of this rule, the method \( m_1 \) in \( A \) becomes just a delegating method: it calls the corresponding method of class \( B \) and passes the parameters in \( pds \) as arguments. The class \( B \) declares a method called \( m_1 \) similar to the method \( m_1 \) that appears on the left-hand side of the rule. The attribute \( x \) is accessed by means of the get and set methods declared in \( B \) itself.

To apply this rule, we assume that \( b \neq \text{null} \) and \( b \neq \text{error} \) along the lifetime of an object of type \( A \), otherwise a call to the method \( m_1 \) of the class \( A \) would abort because this method calls methods of the class \( B \). Also, the parameters \( arg_1 \) and \( arg_2 \) must have basic types.

In order to apply Rule 3 ⟨Move Method⟩ from left to right, the method \( m_1 \) must not be declared in \( B \), nor in any of its superclasses or subclasses, and there should be no occurrences of \textbf{super} in \( m_1 \). Also, attributes declared in \( A \) are not accessed in \( m_1 \), and methods declared in \( A \) are not called inside \( m_1 \). The reason for the last two provisos is that, to move a method in which there are occurrences of attributes or calls to methods of the source class (the one that introduces the method), it would be necessary to pass self as argument in the calls to the method of the target class.
class A extends C
pri b : B : ads_a
meth m_1 \equiv (pds \bullet c_1 var aux : T\bullet
   self.b.g getX(aux);
le := exp_1[aux] end,
self.b.setX(exp_2), a(pds)])
new \equiv (self.b := new B())
mts_a
end
class B extends D
pri x : T : ads_b
meth getX \equiv
   (res arg : T \bullet arg := self.x)
meth setX \equiv
   (val arg : T \bullet self.x := arg)
meth m_1 \equiv
   (pds \bullet c_1 var aux : T\bullet
   self.g getX(aux);
   le := exp_1[aux] end,
   self.setX(exp_2), a(pds)])
mts_b
end

where
\alpha(pds) gives the names of parameters in pds;
provided
(\Rightarrow) b \neq null \land b \neq error is an invariant of A; arg_1 and arg_2 have basic types;
(\Rightarrow) m_1 is not declared in mts_b nor in any superclass or subclass of B; super does not appear in m_1; self.a does not appear in m_1, for any a declared in ads_a; self.p does not appear in m_1, for any method p in mts_b;
(\Rightarrow) N,m_1 does not appear in mts_b, cds or c, for any N such that m \leq N.

(the class to where we move the method). In this way, the parameter declaration of the method in the target class would include an additional parameter: one whose type is the source class. Also, clients of the target class that call the method that was moved would also need to declare an object of the source class and pass such an object as an argument in the method call. To avoid the additional parameter, we restrict the method m_1 not to access attributes nor call methods declared in A. For applying this rule from right to left, the method m_1 must not be called in mts_b, cds, or c, as this method is removed from B.

4 CafeOBJ

CafeOBJ [18] 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.

A module in CafeOBJ has the syntax defined by module < mod id > modElem*, where < mod id > is the name of the module and modElem 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
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.

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 (sections 5.3 and 5.4).

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.

```plaintext
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 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 prec introduces the precedence of the operator.

To implement the laws, a module ROOL-LAWS is defined, which imports modules ROOL-UTILS and 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 ROOL-LAWS comprises the implementation of seventy laws, the module ROOL-UTILS the implementation of seven auxiliary operators and the module ROOL-LEMMAS the implementation of six intermediary results, including lemmas 1 and 2 of Section 3.
module ROOL-LAWS {
    protecting (ROOL-LEMMAS)
    protecting (ROOL-UTILS)
    ...
    var x : Variable.
    var T : Type.
    var c : Command.
    op replaceAndAppend : GuardedCommandList LeftExp Expression -> GuardedCommandList.
    eq replaceAndAppend (p -> c, le, e) = (p [ le = e ]) -> (le := e ; c).
    eq replaceAndAppend ((p -> c) [ gcl, le, e) =
        (p [ le = e ]) -> (le := e ; c) [ ]
        replaceAndAppend(gcl, le, e).
    eq [ var-elim ] : var x : T @ c end = c.
    ...
    eq [ var-:=final-value ] : var x : T @ c end = var x : T @ c end .
    ...
    trans [ var-:=initial-value ] : var x : T @ c end => var x : T @ x := e ; c end.
    ...
    eq [ order-independent-assignment ] : x := e1 ; y := e2 ; x := e1.
    ...
    eq [ method-elimination ] : class C extends D ads meth m := pc ops end =
        class C extends D ads ops end .
    eq [ move-original-method-to-superclass ] : class B extends A ads ops end class C extends B ads' meth m :=
        pc ops' end =
        class B extends A ads meth m := pc ops end class C extends B ads' ops' end .
    ...
}

Figure 3. A fragment of the module ROOL-LAWS in CafeOBJ.

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 \( \langle \ := - < \leftarrow \rangle \ \text{right dist} \rangle \), given in Section 2. An extra operator, 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 \( \langle \ \text{var- := final value} \rangle \) 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 \( \langle \ \text{var- := initial value} \rangle \) is implemented in Line 18, as a transition by using the reserved word 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.

\[
\text{start class A:ClassName extends C:ClassName ads:AttrS meth m:MethodName} \\
\quad \land \quad (pds1:ParDecls \at c1:Command \at c2:Command [ a:Variable ])) \quad mts\text{MethS} \quad \text{end .}
\]

apply -.method-elimination with m = m2:MethodName, pc = (val arg:Variable : T:Type @ c2:Command [ arg ]) \at \text{term} .

result class A:ClassName extends C:ClassName ads:AttrS meth m2:MethodName \land \\
\quad ( \quad (v \quad \text{val} \quad \text{arg:Variable} : \quad \text{T:Type} \at (c2:Command [\text{arg:Variable}])) \quad \text{(meth m:MethodName} \\
\quad \land \quad (pds1:ParDecls \at (c1:Command [ (c2:Command [ a:Variable ])])) \quad \text{mts\text{MethS}) end : ClassDecl .}
\]

apply .lemma2 with vl = arg, x = a:Variable at (3 2 2 2 2 2) .

result class A:ClassName extends C:ClassName ads:AttrS meth m2:MethodName \land \\
\quad ( \quad (v \quad \text{val} \quad \text{arg:Variable} : \quad \text{T:Type} \at (c2:Command [\text{arg:Variable}])) \quad \text{(meth m:MethodName} \\
\quad \land \quad (pds1:ParDecls \at (c1:Command [ ((\text{val arg:Variable} : \quad \text{T:Type} \at (c2:Command [\text{arg:Variable}]])) \at a:Variable >))) \quad \text{mts\text{MethS}) end : ClassDecl .}
\]

apply .lemma1 with m = m2 at (3 2 2 2 2 2) .

result class A:ClassName extends C:ClassName ads:AttrS meth m2:MethodName \land \\
\quad ( \quad (v \quad \text{val} \quad \text{arg:Variable} : \quad \text{T:Type} \at (c2:Command [\text{arg:Variable}])) \quad \text{(meth m:MethodName} \\
\quad \land \quad (pds1:ParDecls \at (c1:Command [ (self. m2:MethodName < a:Variable >)])) \quad \text{mts\text{MethS}) end : ClassDecl .}
\]

Figure 4. Automatic derivation of Rule 1 (Extract/Inline Method).

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 (method elimination). 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 Rule 2 < Self Encapsulate Field >

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

This module imports all operators from ROOL-GRAMMAR. Lines 4 to 11 define all operators necessary to implement this rule. SelfEncapsulate is the main operator, which receives the class declaration to be refactored, the attribute and the names
module ROOL-SELFENCAPSULATE {
  protecting (ROOL-GRAMMAR)
  op SelfEncapsulate : ClassDecl Variable MethodName MethodName -> ClassDecl
  op getType : AttrS Variable -> Type
  op replaceOnMethod : MethS Variable MethodName MethodName Type -> MethS
  ops replaceOnCommand replaceOnAssignmentList replaceOnAssignment :
    Command Variable MethodName MethodName Type -> Command
  op replaceOnExpression : Expression Variable -> Expression
  op containsAttribute : Expression Variable -> Bool
  ops fromListToConcat fromConcatToList : Command -> Command
  ...
  eq SelfEncapsulate( class A extends B as ms end , x , getX , setX ) =
   class A extends B as
     (meth getX ^= ( (res x : getType(as, x)) @ (x := self. x) ) )
     (meth setX ^= ( (val x : getType(as, x)) @ (self. x := x) ) ) )
   replaceOnMethod(ms, x, getX, setX, getType(as, x))
   end .
   ...
   eq getType( (prior a : tipo:Type ;) as , a ) = tipo .
   ...
   eq replaceOnMethod( (meth m ^= (pd @ c)) ms, x, getX, setX, T) =
     (meth m ^= ( pd @ replaceOnCommand(c, x, getX, setX, T) ))
   replaceOnMethod(ms, x, getX, setX, T) .
   ...
   eq replaceOnCommand( leL := eL , x, getX, setX, T) =
     fromConcatToList(replaceOnAssignmentList(       
       fromListToConcat(leL := eL), x, getX, setX, T) )
   ...
   eq replaceOnAssignmentList( le := e ; c , x, getX, setX, T) =
     replaceOnAssignment( le := e , x, getX, setX, T)
     ; replaceOnAssignmentList( c , x, getX, setX, T) .
   ...
   eq replaceOnAssignment( le := e , x, getX, setX, T) =
     var x : T @ self. getX < x > end ; le := replaceOnExpression(e, x)
     if containsAttribute(e, x) .
   ...
   eq replaceOnAssignment(self. x := e,x,getX,setX,T)=self. setX < e >
   ...
   eq replaceOnExpression( e <= e2 , x ) = replaceOnExpression(e, x)
   <=
     replaceOnExpression(e2, x)
   ...
   eq replaceOnExpression( self. x , x ) = x .
   eq replaceOnExpression( e , x ) = e .
   ...
}

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

of the newly introduced get and set methods. The \texttt{getType} operator simply returns the type of the given attribute present in the list of attributes as. Operators \texttt{fromListToConcat} and \texttt{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 \texttt{containsAttribute} receives an expression and an attribute name and returns true if such attribute occurs inside the given expression. The function \texttt{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
1 start SelfEncapsulate( SelfEncapsulate( 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 ) .
2 apply red at term .
3 result class IntRange :ClassName pri _low:Variable : int ; pri _high:Variable : int ; meth getLow:MethodName ^= (( res _low:Variable : int) @ (_low := self._low)) (meth setLow:MethodName ^= (( val _low : int) @ ( self._low := _low)) (meth getHigh:MethodName ^= (( res _high:Variable : int) @ ( _high := self._high)) (meth setHigh:MethodName ^= (( val _high : int) @ ( self._high := _high )) (meth includes:MethodName ^= ((( val arg:Variable : int);( res ret:Variable : bool))@ var _high : int @ ( self. getHigh < _high > ; var _low : int @ ( self. getLow < _low > ; ret:Variable := (arg :Variable => _low) && (arg:Variable <= _high)) end) end) end) end) end) end) end) end) end) end) end) end : ClassDecl
```

Figure 6. Application of the implementation of Rule 2 ⟨Self Encapsulate Field⟩.

```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.

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 inside the operator SelfEncapsulate, 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.

5.4 Implementation and Application of Rule 3 < Move Method >

Rule 3 ⟨Move Method⟩ is used for moving methods between classes, helping to make them simpler and clearer, improving legibility.

Figure 8 shows a fragment of the module ROOL-MOVEMETHOD. Like the previous implementation, the module ROOL-GRAMMAR is imported. Firstly, some variables to be used in the equations are defined, lines 3 to 7. Lines 9 to 18 show the declaration of the most important operators of this implementation. The main operator, MoveMethod, has its behaviour defined by two conditional equations. One of them is shown in lines 20 to 23, the other one is similar. Basically, this operators assigns the class declaration where the method to be moved is (source class) and the class to where it should be moved (target class). Lines 39 to 42 define the
createMethod operator, who creates in the target class the new method. The operator removeMethod is called upon the source class and deletes the method (through the operator deleteMethod, lines 35 to 37) or turns it into a delegation, using the function createDelegate, lines 30 to 33. All the remaining operators are auxiliary.

Considering the example of Figure 9, the application of this implementation is shown in Figure 10. We begin by introducing the classes using the start command, inside the MoveMethod operator. The print command, Line 1, denotes a built-in function of ROOL. The parameters of the operator MoveMethod are the class where the method to be moved is, the class to where the method should be moved and the
### class House
pri _owner : User;

meth printInfo ≡ (val header : string • print(header);
   var name : string • self_owner.getName(name); print(name) end;
   print(self_owner.childrenCount))

meth anotherMethod ≡ (• c)
end
class User
pri _name : string; pub _childrenCount : int;
meth getName ≡ (res n : String • n := self._name )
end

---

**Figure 9.** Initial House and User classes.

```
1 start MoveMethod( class House pri _owner : User ;
   meth printInfo ≈ (val header : String @ (print(header);
      var name : String @ (self._owner.getName(name));
      print(name) end;
      print(self._owner.childrenCount))))
meth anotherMethod ≈ (• c)
end )

2 apply red at term .
3 result class House:ClassName pri _owner:Variable : User:ClassName ;
   meth printInfo:MethodName ≈ ((val header:Variable : String:ClassName @ (print(header);
      var name:Variable : String:ClassName @ (self._owner.getName(name));
      print(name) end;
      print(self._owner.childrenCount)))
   meth anotherMethod:MethodName ≈ ( Variable:ClassName @ (self._owner.getMethodName(name);
      var name:Variable : String:ClassName @ (self._owner.getName(name));
      print(name) end ; print:ParCommand < (self._owner:Variable
      _childrenCount:Variable) )))
meth getName:MethodName ≈ ((res n:
      Variable : String:ClassName @ (n:Variable := self._name:Variable)))
end : ClassDecl
```

---

**Figure 10.** Application of Rule 3 (Move Method).

name of such method. In the example of Figure 10, House is the source class, User is the target class and printInfo is the method to be moved. Observe that in the transformed program the printInfo method in the House class is now a delegation to the printInfo method created in the User class.

### 6 Case Study

In order to illustrate the application of the implemented refactoring rules, a brief case study was developed: a program that maintains information about the products, suppliers and sales of a store.

A class diagram of the initial design, described using UML (Unified Modeling Language) [13], is depicted in Figure 11, where we can see several classes, representing the various concepts of the program. In the context of this case study, the classes Client, Supplier, Brand, Sale and Salesman are merely illustrative. Therefore, their implementations are not relevant and we consider that all their attributes are public. The remaining classes – Product and Item – will be transformed by applying refactorings. Their initial declaration are shown in Figure 12, where print is a built-in function of ROOL.

In this design, the class Product only defines something that can be sold, it does not include price or quantity. To deal with the price and quantity, there is a class called Item, which has some properties like acquisitionDate, purchasePrice, profit,
Figure 11. Design of a product and sales control program.

```plaintext
class Item
  pri acquisitionDate : string; pri purchasePrice : double;
  pri profit : double; pri supp : Supplier; pri prod : Product;
  meth printData ≡ (print(self.acquisitionDate); print(self.purchasePrice);
    print(self.profit); print(self.supp.name);
    var n : string • self.prod.getName(n); print(n) end;
    var d : string • self.prod.getDescription(d); print(d) end;
    var m : Brand • self.prod.getBrand(m); print(m.name) end)
end

class Product
  pri name : string; pri brand : Brand; pri description : string;
  meth getName ≡ (res r : string • r := self.name)
  meth getBrand ≡ (res r : Brand • r := self.brand)
  meth getDescription ≡ (res r : string • r := self.description)
end
```

Figure 12. Classes Item and Product in ROOL.

prod e supp, which represent, respectively, the date when the product was acquired from the supplier, the price payed to the supplier, the value (percent) to be added to that price, the product itself and the supplier of this specific item. To represent the sale of an item, the class Sale is used. Among other attributes, this class maintains the sold item, who sold it and to whom it was sold. Therefore, when, for example, one wants to know if there are any products for sale it is possible to look for objects of Item that are not sold.

One of the first problems to be noticed in the code of Figure 12 is the size of the printData method in the Item class. Observe that its commands can be divided into two distinct logic blocks: the first two lines print information about the item itself, whereas the other ones deal with the product only. In order to obtain a clearer and more readable method definition, it is possible to extract the last three lines in to a separate command. To achieve this, Rule 1 (Extract Method) is used and a new method called printProd is created.

Figure 13 depicts how such refactoring is applied to the Item class. The first step is to introduce, through the start command, the class to be refactored, in this case the class Item. After that, we use two auxiliary equations: markCommandToBeExtracted and introduceExtractMethod, which assign, respectively, the list of commands to be extracted and the class in which the extracted commands are replaced by a call to the new method. These equations are used so that no manual treatment of the code is necessary before introducing it to the rewriting system. The last step is to request the reduction of the current term, in
Figure 13. Extracting the method printProd using Rule 1 (Extract Method).

this case, the class to be refactored.

class Item
  pri acquisitionDate : string; pri purchasePrice : double;
  pri profit : double; pri supp : Supplier; pri prod : Product;
  meth printProd ^= ( @ (print < self. acquisitionDate > ; print < self. purchasePrice > ; print < self. profit > ; print < self. supp . name > ;
    var n : string @ (self. prod . getName < n > ; print < n >) end ;
    var d : string @ (self. prod . getDescription < d > ; print < d >) end ;
    var m : Brand @ (self. prod . getBrand < m > ; print < m . name >) end ) ) end .

apply .markCommandToBeExtracted at (2 2 2 1 2 2 2).
apply .introduceExtractMethod with methodname = printProd at term .
apply red at term .
result class Item: ClassName pri acquisitionDate: Variable: string; (pri purchasePrice: Variable: double; ( pri profit: Variable: double; ( pri supp: Variable: Supplier: ClassName; pri prod: Variable: Product: ClassName;)))
  meth printProd: MethodName ^= ( @ (var n: Variable: string @ ((self. prod: Variable . getName: MethodName) < n: Variable > ;
    print: ParCommand < n: Variable >) end ;
    (var d: Variable : string @ ((self. prod: Variable . getDescription: MethodName) < d: Variable > ;
      print: ParCommand < d: Variable >) end ;
    var m: Variable : Brand: ClassName @ ((self. prod: Variable . getBrand: MethodName) < m: Variable > ;
      print: ParCommand < (m: Variable . name: Variable) >) end )))
  meth printData: MethodName ^= (@ (print: ParCommand < self. acquisitionDate: Variable > ; print: self. purchasePrice: Variable > ;
    print: self. profit: Variable > ;
    print: (self. supp: Variable . name: Variable) > ;
    self. printProd < >)) end : ClassDecl

Figure 14. Classe Item after extraction of method printProd.

The Item class obtained after the refactoring is shown, described in ROOL, in Figure 14, for didactic purposes. Observe that it now has a method called printProd which deals only with the attribute prod of type Product. Notice that the method printData now has a call to the newly introduced method printProd.

In order to reduce the coupling between the classes Item and Product, the method printProd can be moved to the class Product. To do so, Rule 3 ⟨Move Method⟩ is applied, as shown if Figure 15. Similarly to the previous application, it is necessary to use an auxiliary equation, introduceMoveMethod, to specify which classes will be refactored. The Product and Item classes after the refactoring are shown in Figure 16, for didactic reasons. Notice that the method printProd now belongs to the class Product. The final class diagram is depicted in Figure 17.

7 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. Nev-
rewriting systems could be considered as supporting tools to construct refactoring tem. On the other hand, for the refactoring community, this work suggests that as a programming transformation case study, in the context of this rewriting sys-garded as a contribution to the CafeOBJ community, as it could be considered grammar and laws imposed no problems. Thus, this work could also be re-rool allowing the construction of a transformation environment.

Each refactoring is implemented as a rule in CafeOBJ, applying red at term.

```start class Product pri name: string; pri brand: Brand; pri description : string; meth getName ^= (res r: string @ (r := self. name)) meth getBrand ^= (res r: Brand @ (r := self. brand)) meth getDescription ^= (res r: string @ (r := self. description)) end class Item pri acquisitionDate: string; pri purchasePrice: double; pri profit: double; pri supp: Supplier; pri prod: Product; meth printProd ^= (@ (self. prod . getBrand) < t >; print < t >; printProd < >; (print < self. supp . name >; self. printProd < >))) end class Product: ClassName pri name: string; (pri brand:Variable: Brand:ClassName; pri prod:Variable: Product: ClassDecls) meth getDescription ^= (res r: string @ (r := self. description)) meth printProd:MethodName ^= ( @ ((self. prod . getBrand) < t >; print < t >) result class Item: ClassName pri acquisitionDate:Variable: string; (pri supp:Variable: Supplier:ClassName; pri prod:Variable: Product: ClassDecls) meth getDescription ^= (res r: string @ (r := self. description)) meth printProd:MethodName ^= ( @ ((self. prod . getBrand) < t >; print < t >) end; var m: Brand @ ((self. prod . getBrand) < m >; print < (m. name) >) end)) meth printData:MethodName ^= (@ (print < self. acquisitionDate >; (print < self. purchasePrice >; (print < self. profit >; (print < (self. supp . name) >; self. printProd < >)))) end. apply .introduceMoveMethod with Co = Item:ClassName, Ct = Product:ClassName, methname = printProd:MethodName at term. apply red at term.
```

Figure 15. Application of Rule 3 (Move Method) in Product and Item.

```class Product
pri name: string; pri brand: Brand; pri description: string;
meth printProd ≡ ( • var n: string • self.getName(n); print(n) end; var d: string • self.getDescription(d); print(d) end; var m: Brand • self.getBrand(m); print(m.name) end)
meth getName ≡ (res r: string • r := self.name)
meth getBrand ≡ (res r: Brand • r := self.brand)
meth getDescription ≡ (res r: string • r := self.description)
end
```

```class Item
pri acquisitionDate: string; pri purchasePrice: double;
pri profit: double; pri supp: Supplier; pri prod: Product;
meth printProd ≡ ( • self.prod.printProd()) meth printData ≡ ( • print(self.acquisitionDate); print(self.purchasePrice);

print(self.profit); print(self.supp.name); self.printProd())
end
```

Figure 16. Classes Product and Item after the application of Rule 3 (Move Method).

etherelease 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
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 also a direction of future work.

Acknowledgement

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

References


