Electronic Notes in Theoretical Computer Science 184 (2007) 39–61 

[www.elsevier.com/locate/entcs](http://www.elsevier.com/locate/entcs)

Using CafeOBJ to Mechanise Refactoring Proofs and Application

Antonio Carvalho Ju´nior[1](#_bookmark0)

*Departamento de Ciˆencia da Computac¸a˜o e Estat´ıstica Universidade Federal de Sergipe (UFS)*

*CEP 41900-000 – Sa˜o Cristo´va˜o – SE, Brazil*

Leila Silva[2](#_bookmark0)

*Departamento de Ciˆencia da Computac¸a˜o e Estat´ıstica Universidade Federal de Sergipe (UFS)*

*CEP 41900-000 – Sa˜o Cristo´va˜o – SE, Brazil*

M´arcio Corn´elio[3](#_bookmark0)

*Departamento de Sistemas Computacionais Escola Polit´ecnica – 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

# Introduction

Changes are common to software. Many practitioners recognize that changing an object-oriented software is easier than conventional ones [[15](#_bookmark83)]. Refactoring is the

1 Email: [acjr@dcce.ufs.br](mailto:acjr@dcce.ufs.br)

2 Email: [leila@ufs.br](mailto:leila@ufs.br)

3 Email: [mlc@dsc.upe.br](mailto:mlc@dsc.upe.br)

1571-0661 © 2007 Elsevier B.V. Open access under [CC BY-NC-ND license.](http://creativecommons.org/licenses/by-nc-nd/3.0/)

doi:10.1016/j.entcs.2007.03.014

activity of modifying a software system by preserving the external behaviour, per- ceived by the user. In fact, just the internal software structure is affected, by for example, moving attributes and methods between classes.

In [[10](#_bookmark76),[21](#_bookmark89)] several refactorings are introduced in a rather informal way. Opdyke

[[19](#_bookmark87)] formalises refactorings to the degree that they can be encoded in tools. Cinn´eide and Nixon [[6](#_bookmark74)] present a semi-formal approach to demonstrate behaviour preservation for design patterns transformations, defined in terms of refactorings. Lano *et al*

[[12](#_bookmark77)] 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. Corn´elio [[8](#_bookmark78)] adopts a transformational approach, that is constructively based on rules, to formalise several refactorings introduced previously by Fowler [[10](#_bookmark76)].

The work proposed by Corn´elio [[8](#_bookmark78)] is based on transformation rules between meta-programs in rool (Refinement Object-Oriented Language) [[3](#_bookmark71)], which is a subset of sequential Java [[1](#_bookmark69)] 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 rewrit- ing system CafeOBJ [[18](#_bookmark86)]. Moreover, a case study showing the application of the refactorings is also developed. In this way, this work complements the formal rigour of [[8](#_bookmark78)], 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](#_bookmark82)] use Maude [[7](#_bookmark75)] to implement a reduction strategy for

object-oriented languages and Silva *et al* [[20](#_bookmark88)] implement in CafeOBJ a reduction

strategy in the context of hardware/software partitioning.

This paper is organised as follows. Section [2](#_bookmark1) introduces rool, the language used to formalise the refactorings. In Section [3](#_bookmark2) we present three refactoring rules used to illustrate the work here developed. The rewriting system CafeOBJ is briefly introduced in Section [4](#_bookmark3). Section [5](#_bookmark5) 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](#_bookmark60). Finally, Section [7](#_bookmark65) gives the conclusions and directions for future work.

# ROOL and Laws

ROOL [[4](#_bookmark72),[3](#_bookmark71)] is an object-oriented language based on sequential Java. It allows reas- oning about object-oriented programs and specifications, as both kind of constructs are mixed in the style of Morgan’s refinement calculus [[16](#_bookmark84),[17](#_bookmark85)]. 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](#_bookmark84)], which is an extension of Dijkstra’s language of guarded commands [[9](#_bookmark79)]. The meaning of rool constructs is expressed by algebraic laws, in the style of [[16](#_bookmark84)].

A program *cds • 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*.

**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 **ex- tends**. 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* in- cludes 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](#_bookmark70),[5](#_bookmark73)]. A parameterised command can have the form **val** *x* : *T • c* or **res** *x* : *T • 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](#_bookmark84)], 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 • c* = *cds*2*cds • c*, which is fomalised in [[4](#_bookmark72),[3](#_bookmark71)]. 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](#_bookmark78)]. Here we only introduce some laws of command and laws of classes, necessary to understand the mechanisation of the proofs, presented in Section [5](#_bookmark5). The explanations of these laws are extracted from [[8](#_bookmark78)].

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

**Law 1** *⟨*:= *− D right dist⟩* If *e* is total, then

*le* := *e*; **if** []*i • ψi → ci* **fi** = **if** []*i • ψi*[*e/le*] *→* (*le* := *e*; *ci*) **fi**

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 2** *⟨var elim⟩* If *x* is not free in *c*, then **var** *x* : *T • c* **end** = *c*

**Law 3** *⟨var*- := *final value⟩* If *x* is not free in *c*, then

**var** *x* : *T • c*; *x* := *e* **end** = **var** *x* : *T • c* **end**

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 4** *⟨var*- := *initial value⟩* If *e* has no type errors, then

**var** *x* : *T • c* **end** *±* **var** *x* : *T • x* := *e*; *c* **end**

**Law 5** *⟨order independent assignment⟩* 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 *⟨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⟩*

=*cds,c*

**class** *C* **extends** *D ads*

*mts*

**end**

**class** *C* **extends** *D ads*

**meth** *m* =^ *pc mts*

**end**

## provided

(*→*) *B.m* does not appear in *cds, c* nor in *mts*, for any *B* such that

*B ≤ C*.

(*←*) *m* is not declared in ops nor in any superclass or subclass of

*C* in *cds*.

Law 7 *⟨move original method to superclass⟩*, allows us to move an original method to a superclass. Between other requirements, this law demands that occur- rences of **self** in methods to be moved are cast.

**Law 7** *⟨move original method to superclass⟩*

=*cds,c*

**class** *B* **extends** *A ads*

**meth** *m* =^ *pc mts*

**end**

**class** *C* **extends** *B ads'*

*mts'*

**end**

**class** *B* **extends** *A ads*

*mts*

**end**

**class** *C* **extends** *B ads'*

**meth** *m* =^ *pc mts'*

**end**

## provided

(*↔*) (1) **super** and private attributes do not appear in *pc*; (2) *m*

is not declared in any superclass of *B* in *cds*;

(*→*) (1) *m* is not declared in *mts*, and can only be declared ina class

*D*, for any *D ≤ B* and *D* # *C*, if it has the same parameters as *pc*;

(2) *pc* does not contain uncast occurrences of **self** nor expressions in the form ((*C*)**self** )*.a* for any private attribute *a* in *ads'*;

(*←*) (1) *m* is not declared in *mts'*; (2) *D.m*, for any *D ≤ B*, does not appear in *cds, c*, *mts* or *mts'*.

# Refactoring Rules

A comprehensive set of refactoring rules which captures and formalises most of the refactorings informally introduced in [[10](#_bookmark76)] is presented in [[8](#_bookmark78)]. 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 abbrevi- ations. 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 mechan- isation 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 onde moves a method from a class to another, and is used in the case study developed here.

* 1. *Extract and Inline Method*

Rule 1, when considered from left to right, coincides with the refactoring Extract Method presented by Fowler [[10](#_bookmark76), p. 110], whereas the application in the reverse direction corresponds to the refactoring Inline Method [[10](#_bookmark76), 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** *⟨Extract/InlineMethod⟩*

=*cds,c*

**class** *A* **extends** *C ads*;

1

**meth** *m*2 =^ (*pds*2 *• c*2[*α*(*pds*2)])

*mts'*

**end**

**meth** *m*1 =^ (*pds*1 *• c'* )

**class** *A* **extends** *C ads*;

**meth** *m*1 =^

(*pds*1 *• c*1[*c*2[*a*]]) *mts*

**end**

## where

*'* =^ *c*1[**self** *.m*2(*a*)*/c*2[*a*]]

*c*

1

*mts* =^ *mts'*[*c*2[*a*]*/***self** *.m*2(*a*)]

*a* a is the finite set of free variables of command *c*2, not including attributes of class *A*;

## provided

(*↔*) (1) Variables in *a* have basic types; (2) Parameters in *pds*2

must have the same types as those of variables in *a*;

(*→*) *m*2 is not declared in *mts* nor in any superclass or subclass of

*A* in *cds*;

(*←*) *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 ≤ 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 **extends** clause, and in the others that follow, is present in the sequence of class declarations *cds* or is the class **object**.

On the right-hand side of this rule, the method call **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'* . In the command *c*2 of method *m*2, the variables indicated by *a* are replaced with the parameters *α*(*pds*), where *α*(*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.

1

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 **rec** *X • c*3 **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](#_bookmark78)].

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 *D* in *cds, C D 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 declar- ation*

## class *C* extends *D*

*ads*

**meth** *m* =^ *pc mts*

## end

*is included in cds. Then cds, C D pc*(*e*) = **self** *.m*(*e*)

Lemma 2 (*pcom value − argument*) transforms a single command into a para- meterised one with a value argument.

**Lemma 2** (*pcom value − argument*) *c* = (**val** *vl* : *T • 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 /*= *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 =^ (**val** *arg* : *T • c*2[*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.

(**val** *arg* : *T • c*2[*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*** )

**class** *A ads*;

**meth** *m*1 =^ (*pds*1 *• c*1[**self***.m*2(*a*)])

**meth** *m*2 =^ (**val** *arg* : *T • c*2[*arg*]) *mts*

## 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](#_bookmark78)], which merges two parameterised commands into one.

* 1. *Self Encapsulate Field*

Rule 2 *⟨Self EncapsulateField⟩* introduces get and set methods for an attribute *x*

declared in the class *A*. The derivation of this rule can be found in [[8](#_bookmark78)].

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 **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 *mtsa* . 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 *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.

**Rule 2** *⟨Self EncapsulateField⟩*

=*cds,c*

**class** *A* **extends** *C*

**pri** *x* : *T* ;

*adsa*;

**meth** *m*1 =b (*pds*1 *• c'* )

1

**meth** *getX* =b

(**res** *arg* : *T • arg* := **self***.x*) **meth** *setX* =b

(**val** *arg* : *T •* **self***.x* := *arg*)

*mts'*

*a*

**end**

**class** *A* **extends** *C*

**pri** *x* : *T* ;

*adsa*;

**meth** *m*1 =b (*pds*1*• c*1[*le*1 := *exp*1[**self***.x*]*,* **self***.x* := *exp*2])

*mtsa*

**end**

**where**

*c'* =b *c*1[**var** *aux* : *T •* **self***.getX*(*aux*); *le*1 := *exp*1[*aux*] **end***,*

1

*mts'*

*a*

**self***.setX*(*exp*2) */ le*1 := *exp*1[**self***.x*]*,* **self***.x* := *exp*2]

=b *mtsa*[**var** *aux* : *T •* **self***.getX*(*aux*); *le*1 := *exp*1[*aux*]

**end***,* **self***.setX*(*exp*2) */ le*1 := *exp*1[**self***.x*]*,* **self***.x* := *exp*2]

**provided**

(*→*) *getX* is not declared in any superclass or subclass of *A* in *cds*; *setX* is not declared in any superclass or subclass of *A* in *cds*;

(*←*) *le.getX* and *le.setX* do not appear in *mts'* , *cds* or *c*, for any *le* such that *le ≤ A*.

*a*

* 1. *Move Method*

Moving methods between classes helps to make them simpler and clearer, improving legibility. Rule 3 *⟨Move M ethod⟩* 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 *α*(*pds*), where *α*(*pds*) gives the list of parameter names declared in *pds*. For instance, if *pds* is the declaration (**val** *x* : *T* ; **res** *y* : *T* ), the list given by *α*(*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 /*= **null** and *b /*= **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 M ethod⟩* 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 **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

**Rule 3** *⟨Move Method⟩*

=*cds,c*

**class** *A* **extends** *C*

**pri** *b* : *B* ; *adsa*

**meth** *m*1 =b (*pds • c*1[**var** *aux* : *T •* **self***.b.getX*(*aux*);

*le* := *exp*1[*aux*] **end***,* **self***.b.setX*(*exp*2)*, α*(*pds*)]))

**new** =b (**self***.b* := *new B*())

*mtsa*

**end**

**class** *B* **extends** *D* **pri** *x* : *T* ; *adsb* **meth** *getX* =b

(**res** *arg* : *T •* arg := **self***.x*) **meth** *setX* =b

(**val** *arg* : *T •* **self***.x* := *arg*) *mtsb*

**end**

**class** *A* **extends** *C*

**pri** *b* : *B* ; *adsa*

**meth** *m*1 =b (*pds •*

**self***.b.m*1(*α*(*pds*))) **new** =b (**self***.b* := *new B*()) *mtsa*

**end**

**class** *B* **extends** *D* **pri** *x* : *T* ; *adsb* **meth** *getX* =b

(**res** *arg* : *T •* arg := **self***.x*) **meth** *setX* =b

(**val** *arg* : *T •* **self***.x* := *arg*)

**meth** *m*1 =b

(*pds • c*1[**var** *aux* : *T •*

**self***.getX*(*aux*);

*le* := *exp*1[*aux*] **end***,* **self***.setX*(*exp*2)*, α*(*pds*)]))

*mtsb*

**end**

**where**

*α*(*pds*) gives the names of parameters in *pds*;

**provided**

(*↔*) *b /*= **null** *∧ b /*= **error** is an invariant of *A*; *arg*1 and *arg*2 have basic types;

(*→*) *m*1 is not declared in *mtsb* 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 *adsa*; **self** *.p* does not appear in *m*1, for any method *p* in *mtsa*;

(*←*) *N.m*1 does not appear in *mtsb*, *cds* or *c*, for any *N* such that *N ≤ B*.

(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 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 *mtsb*, *cds*, or *c*, as this method is removed from *B*.

# CafeOBJ

CafeOBJ [[18](#_bookmark86)] is a new generation algebraic specification and programming language. As a successor of the OBJ family (OBJ1, OBJ2, OBJ3) [[11](#_bookmark80)], it inherits features such as: powerful typing system with sub-types; sophisticated module composition system, featuring several kinds of imports; parameterised modules; views for in- stantiating 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 > 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

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 impor- ted module can be extended or can change the original description). The second part, signature, declares sorts, operators, records and subsorts used by the mod- ule. 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](#_bookmark4), 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.

# Proving and Implementing Refactoring Rules in CafeOBJ

The mechanisation of the proofs of refactoring rules comprises two steps: the imple- mentation of the rool grammar and laws (Section [5.1](#_bookmark6)) and the automatic derivation of the rules from the laws already implemented (Section [5.2](#_bookmark35)). After proved, the re- factoring can be implemented as a rule and used to perform program transformation (sections [5.3](#_bookmark37) and [5.4](#_bookmark43)).

* 1. *The Implementation of* rool *Grammar and Laws*

To implement the grammar, a module ROOL-GRAMMAR is defined, including the lan- guage 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](#_bookmark21), where the lines are numbered for didactic reasons. This facility is used in the remainder of this paper.

1

2

3

4

5

6

7

8

9

10

11

12

13

14

module ROOL - GRAMMAR {

[ ClassName PrimitiveType < Type ] op bool : - > PrimitiveType .

op int op char

...

op \_is\_ op ( \_) \_ op \_. \_

...

op \_ , \_

...

op \_ - >\_

: - > PrimitiveType .

: - > PrimitiveType .

: Expression ClassName

: ClassName Expression

: Expression Variable

: Variable VariableList

- > Bool .

- > Expression .

- > Expression .

- > VariableList { r - assoc } .

: Bool Command - > Guarded Command { prec : 39 l - assoc } .

op \_ [] \_

: GuardedCommandList GuardedCommandList - >

GuardedCommandList { ... } .

15

16

17

18

19

20

...

op ( if\_fi ) : GuardedCommandList - > Command .

...

op ( val\_ : \_) : VariableList TypeList - > ParDecl .

...

op class\_extends end : ClassName ClassName ClassBody - > ClassDecl { prec : 40} .

21 op class end : ClassName ClassBody - > ClassDecl { prec : 40} .

22

23

24

25

26

27

...

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](#_bookmark7) the sorts ClassName and PrimitiveType are declared as subsorts of Type. Lines [3](#_bookmark8) to [5](#_bookmark9) define some primitive types of rool. The type test, cast and the attribute selection operators are defined in lines [7](#_bookmark10) to [9](#_bookmark11), 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](#_bookmark12), which creates lists of variables. As an example of command implementation, the guarded command is depicted in lines [13](#_bookmark13) and [14](#_bookmark14), and the alternation in Line [16](#_bookmark15). Parameters declaration is implemented as in Line [18](#_bookmark16), which shows the call-by-value mechanism. Class declarations, considering superclasses or not, are shown in lines [20](#_bookmark17) and [21](#_bookmark18), respectively. In Line [20](#_bookmark17) the second ClassName corresponds to the superclass. Private attributes are declared using the operator in Line [23](#_bookmark19) and Line [25](#_bookmark20) 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 auxili- ary 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 com- prises the implementation of seventy laws, the module ROOL-UTILS the implementa- tion of seven auxiliary operators and the module ROOL-LEMMAS the implementation of six intermediary results, including lemmas 1 and 2 of Section [3](#_bookmark2).

1

2

3

4

5

6

7

8

9

10

11

12

13

14

15

16

17

18

19

20

21

22

23

24

25

26

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 [ e / le ]) - > ( le := e ; c)

.

eq replaceAndAppend (( p - > c) [] gcl , le , e) =

( p [ e / le ]) - > ( le := e ; c) [] replaceAndAppend ( gcl , le , e) .

eq [:=-<>- right - dist ] : le := e ; if gcl fi = if replaceAndAppend ( gcl

, le , e) fi .

...

eq [ var - elim ] : var x : T @ c end = c .

...

eq [ var -:= - final - value ] : var x : T @ c ; x := e 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 = 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 ] :

27

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 .

28 ...

29

}

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

To illustrate this module, Figure [3](#_bookmark34) shows a small fragment of the generated code. Basically some variables used in the laws equations are introduced, as the ones in Lines [5](#_bookmark22) to [7](#_bookmark23). After that, the command laws are implemented. Lines [8](#_bookmark24) through [12](#_bookmark25) depicts the implementation of Law 1 *⟨*:= *− D right dist⟩*, given in Section

[2](#_bookmark1). 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](#_bookmark26), [16](#_bookmark27) and [20](#_bookmark29) show, respectively, the implementation of Law 2 *⟨var elim⟩*, 3 *⟨var*- := *final value⟩* and, 5 *⟨order independent assignment⟩*, explained in Section [2](#_bookmark1). The implementation of these laws is immediate where the symbol “@” means “*•*” in the formal definition. Law 4 *⟨var*- := *initial value⟩* is implemented in Line [18](#_bookmark28), 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](#_bookmark30) to [24](#_bookmark31) introduce Law 6 *⟨method elimination⟩*, whereas lines [25](#_bookmark32) to [27](#_bookmark33) define Law 7 *⟨move original method to superclass⟩*. The implementation of both laws follows directly from the formal definition presented in Section [2](#_bookmark1). 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.

* 1. *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 Corn´elio in [[8](#_bookmark78)], is reproduced in Section [3](#_bookmark2).

start class A:ClassName extends C:ClassName ads:AttrS meth m:MethodName *∧*= (pds1:ParDecls @ c1:Command [ c2:Command [ a:Variable ] ]) mts:MethS end .

apply -.method-elimination with m = m2:MethodName, pc = (val arg:Variable : T:Type @ c2:Command [ arg ] ) 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 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 *∧*= (pds1:ParDecls @ (c1:Command [ (((val arg:Variable : T:Type) @ (c2:Command [ arg:Variable

])) < a:Variable >) ])) mts:MethS) end : ClassDecl

apply .lemma1 with m = m2 at (3 2 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 *∧*= (pds1:ParDecls @ (c1:Command [ (self. m2:MethodName < a:Variable >) ])) mts:MethS) end

: ClassDecl

Figure 4. Automatic derivation of Rule 1 *⟨*Extract/Inline Method*⟩*.

Figure [4](#_bookmark36) 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](#_bookmark2), 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](#_bookmark1). 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](#_bookmark78)] are reduced in a similar way.

* 1. *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](#_bookmark40).

This module imports all operators from ROOL-GRAMMAR. Lines [4](#_bookmark38) to [11](#_bookmark39) 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

* + 1. module ROOL - SELFENCAPSULATE {
    2. protecting ( ROOL - GRAMMAR )
    3. ...
    4. op SelfEncapsulate : 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 |  | Command Variable MethodName MethodName Type - > Command . |  |
| 9 | op | replaceOnExpression : Expression Variable - > Expression . |  |
| 10 | op | containsAttribute : Expression Variable - > Bool . |  |
| 11 | ops | fromListToConcat fromConcatToList : Command - > Command . |  |
| 12 | ... |  |  |
| 13 | eq | SelfEncapsulate ( class A extends B as ms end , x , get X , set X ) |  |
|  |  | = |  |
| 14 |  | class A extends B |  |
| 15 |  | as |  |
| 16 |  | ( meth get X ^= ( ( res x : getType ( as , x)) @ ( x := self . x) ) |  |
|  |  | ) |  |
| 17 |  | ( meth set X ^= ( ( val x : getType ( as , x)) @ ( self . x := x) ) |  |
|  |  | ) |  |
| 18 |  | replaceOnMethod ( ms , x , getX , setX , getType ( as , x)) |  |
| 19 |  | end . |  |
| 20 | ... |  |  |
| 21 | eq | getType ( ( pri a : tipo : Type ;) as , a ) = tipo . |  |
| 22 | ... |  |  |
| 23 | eq | replaceOnMethod ( ( meth m ^= ( pd @ c)) ms , x , getX , setX , T) = |  |
| 24 |  | ( meth m ^= ( pd @ replaceOnCommand ( c , x , getX , setX , T) )) |  |
| 25 |  | replaceOnMethod ( ms , x , getX , setX , T) . |  |
| 26 | ... |  |  |
| 27 | eq | replaceOnCommand ( leL := eL , x , getX , setX , T) = |  |
| 28 |  | fromConcatToList ( replaceOnAssignmentList ( |  |
| 29 |  | fromListToConcat ( leL := eL ) , x , getX , setX , T) ) |  |
|  |  | . |  |
| 30 | ... |  |  |
| 31 | eq | replaceOnAssignmentList ( le := e ; c , x , getX , setX , T) = |  |
| 32 |  | replaceOnAssignment ( le := e , x , getX , setX , T) |  |

.

|  |  |  |
| --- | --- | --- |
| 33 |  | ; replaceOnAssignmentList ( c , x , getX , setX , T) |
|  |  | . |
| 34 | ... |  |
| 35 | ceq | replaceOnAssignment ( le := e , x , getX , setX , T) = |
| 36 |  | var x : T @ self . getX < x > end ; le := replaceOnExpression ( e , |
|  |  | x) |
| 37 |  | if containsAttribute ( e , x) |

1. ...
2. eq replaceOnAssignment ( self . x := e , x , getX , setX , T) = self . set X < e >

.

1. ...
2. eq replaceOnExpression ( e <= e2 , x ) = replaceOnExpression ( e , x)

<=

1. replaceOnExpression (e2 , x)

.

1. ...
2. eq replaceOnExpression ( self . x , x ) = x .
3. eq replaceOnExpression ( e , x ) = e .
4. ...

47 }

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

of the newly introduced get and set methods. The getType operator simply re- turns the type of the given attribute present in the list of attributes as. Operators fromListToConcat and fromConcatToList make the term simpler for the applic- ations 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 as- signed to its corresponding expression in the list. The function containsAttribute receives an expression and an attribute name and returns true if such attribute oc- curs 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.

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 , get Low , setLow ) .

1. apply red at term .
2. result class IntRange : ClassName pri \_low : Variable : int ; pri \_high : Variable : int ; meth getLow : MethodName ^= (( res \_low : Variable : int ) @ ( \_low := self . \_low )) ( meth set Low : 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 . get Low < \_low > ; ret : Variable := ( arg

: Variable >= \_low ) && ( arg : Variable <= \_high )) end ) end ) meth grow

: MethodName ^= (( val factor : Variable : int ) @ var \_high : int @ ( self . getHigh < \_high > ; self . setHigh < ( \_high \* factor :

Variable ) >) end ))))) end : ClassDecl

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

**class** *IntRange* **pri** *low* : **int**; **pri** *high* : **int**;

**meth** *includes* =b (**val** *arg* : **int**; **res** *ret* : *bool •*

*ret* := (*arg >*= **self***. low*) && (*arg <*= **self***. high*))

**meth** *grow* =b (**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, con- sidering that the provisos are satisfied, is shown in Figure [6](#_bookmark41). The example intro- duced by the start command is extracted from Fowler [[10](#_bookmark76)], and reproduced in Figure [7](#_bookmark42). 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 trans- formed program all assignments to the encapsulated attributes use set methods. Moreover, where these attributes are only used, a get method is introduced.

* 1. *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](#_bookmark56) 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](#_bookmark44) to [7](#_bookmark45). Lines [9](#_bookmark46) to [18](#_bookmark47) 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](#_bookmark48) to [23](#_bookmark49), 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](#_bookmark54) to [42](#_bookmark55) define the

* + 1. module ROOL - MOVEMETHOD {
    2. protecting ( ROOL - GRAMMAR )
    3. var bd : ClassBody .
    4. vars mm m : MethodName .
    5. vars Co Ct : ClassName .
    6. vars cdl cdl1 cdl2 : ClassDecl .
    7. var pard : ParDecl .
    8. ...
    9. op MoveMethod : ClassDecls ClassName ClassName MethodName - > ClassDecls .
    10. op uses Attribute Of Target Class : ClassDecl MethodName - > Bool .
    11. ...
    12. op removeMethod : ClassDecl Bool ClassName MethodName - > ClassDecl

.

* + 1. op create Delegate : MethS MethodName LeftExp - > MethS .
    2. op deleteMethod : MethS MethodName - > MethS .
    3. ...
    4. op createMethod : ClassDecl Bool ClassDecl MethodName - > ClassDecl

.

* + 1. op getMethodPC : ClassDecl MethodName - > ParCommand .
    2. op getClassName : ClassDecl - > ClassName .
    3. ...
    4. ceq MoveMethod ( cdl1 cdl2 , Co , Ct , m ) =
    5. remove Method ( cdl1 , uses Attribute Of Target Class ( cdl1 , m) , Ct , m )
    6. create Method ( cdl2 , uses Attribute Of Target Class ( cdl1 , m) , cdl1 , m

)

* + 1. if ( getClassName ( cdl1 ) == Co ) and ( getClassName ( cdl2 ) == Ct ).
    2. ...
    3. eq remove Method ( class A extends B as ms end , false , Ct , m) =
    4. class A extends B as delete Method ( ms , m) end .
    5. eq remove Method ( class A extends B as ms end , true , Ct , m) =
    6. class A extends B as create Delegate ( ms , m , self . getCTattr ( as , Ct ) ) end .
    7. ...
    8. eq createDelegate ( ( meth mm ^= ( pd @ c)) ms , m , le ) =
    9. ( meth mm ^= ( pd @ c)) createDelegate ( ms , m , le ) .
    10. eq createDelegate ( ( meth mm ^= ( pd @ c)) , mm , le ) =
    11. ( meth mm ^= ( pd @ ( le . mm < alpha ( pd ) >)))

.

* + 1. ...
    2. eq delete Method ( ( meth mm ^= ( @ c)) ms , m ) =
    3. ( meth mm ^= ( @ c)) deleteMethod ( ms , m)

.

* + 1. eq deleteMethod ( ( meth mm ^= ( pd @ c)) ms , mm ) = ms .
    2. ...
    3. eq create Method ( class A extends B as ms end , false , cdl , m) =
    4. class A extends B as (( meth m ^= getMethodPC ( cdl , m)) ms ) end

.

* + 1. eq create Method ( class A extends B as ms end , true , cdl , m) =
    2. class A extends B as (( meth m ^= replaceSelfs ( getMethodPC ( cdl , m))) ms ) end .
    3. ...
    4. eq getMethodPC ( class A extends B as ms end , m) = getMethodPC2 ( ms , m)

.

* + 1. eq get Class Name ( class A extends B as ms end ) = A .

46 }

Figure 8. O m´odulo ROOL-MOVEMETHOD em CafeOBJ.

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](#_bookmark52) to [37](#_bookmark53)) or turns it into a delegation, using the function createDelegate, lines [30](#_bookmark50) to [33](#_bookmark51). All the remaining operators are auxiliary.

Considering the example of Figure [9](#_bookmark57), the application of this implementation is shown in Figure [10](#_bookmark59). We begin by introducing the classes using the start command,

inside the MoveMethod operator. The print command, Line [1](#_bookmark58), 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* =b (**val** *header* : **string** *• print*(*header*);

**var** *name* : **string** *•* **self***. owner.getName*(*name*); *print*(*name*) **end**;

*print*(**self***. owner. childrenCount*))

**meth** *anotherMethod* =b ( *• c*)

**end**

**class** *User*

**pri** *name* : **string**; **pub** *childrenCount* : **int**;

**meth** *getName* =b (**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 class User pri \_name : String ; pub \_children Count : int ; meth getName ^= (( res n : String ) @ ( n := self . \_name )) end , House , User

, printInfo ) .

1. apply red at term .
2. result class House : ClassName pri \_owner : Variable : User : ClassName ; meth printInfo : MethodName ^= (( val header : Variable : String : ClassName ) @ (( self . \_owner . printInfo ) < header >)) meth anotherMethod :

MethodName ^= (@ c: Command ) end class User : ClassName pri \_name : Variable : String : ClassName ; pub \_childrenCount : Variable : int : Type

; meth printInfo ^= (( val header : String ) @ ( print : ParCommand < header : Variable > ; ( var name : Variable : String : ClassName @ ( self . getName : MethodName < name : Variable > ; print : ParCommand < name : Variable >) end ; print : ParCommand < ( self . \_owner : Variable .

\_childrenCount : Variable ) >))) meth getName : MethodName ^= (( res n: Variable : String : ClassName ) @ ( n: Variable := self . \_name : Variable ))

end : ClassDecls

Figure 10. Application of Rule 3 *⟨Move Method⟩*.

name of such method. In the example of Figure [10](#_bookmark59), 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.

# 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](#_bookmark81)], is depicted in Figure [11](#_bookmark61), where we can see several classes, repres-

enting the various concepts of the program. In the context of this case study, the classes *Client*, *Supplier*, *Brand*, *Sale* and *Salesman* are merely illustrative. There- fore, their implementations are not relevant and we consider that all their attributes

are public. The remaining classes – *Product* and *Item* – will be transformed by ap- plying refactorings. Their initial declaration are shown in Figure [12](#_bookmark62), 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*,

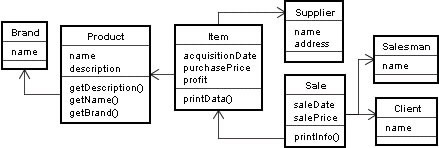


Figure 11. Design of a product and sales control program.

**class** *Item*

**pri** *acquisitionDate* : **string**; **pri** *purchasePrice* : **double**; **pri** *profit* : **double**; **pri** *supp* : *Supplier*; **pri** *prod* : *Product*;

**meth** *printData* =b (*•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* =b (**res** *r* : **string** *• r* := **self***.name*)

**meth** *getBrand* =b (**res** *r* : *Brand • r* := **self***.brand*)

**meth** *getDescription* =b (**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](#_bookmark62) 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](#_bookmark63) depicts how such refactoring is applied to the *Item* class. The first step is to introduce, through the start command, the class to be refact- ored, in this case the class Item. After that, we use two auxiliary equations: markCommandToBeExtracted and introduceExtractMethod, which assign, respect- ively, 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

start 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 . get Description < 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 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 : ParCommand < self . purchasePrice : Variable > ; ( print : ParCommand

< self . profit : Variable > ; ( print : ParCommand < ( self . supp : Variable

. name : Variable ) > ; self . printProd < >))))) end : ClassDecl

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* =b ( *•* **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**)

**meth** *printData* =b ( *• print*(**self***.acquisitionDate*); *print*(**self***.purchasePrice*); *print*(**self***.profit*); *print*(**self***.supp.name*);

**self** *.printProd* ())

**end**

Figure 14. Classe *Item* after extraction of method *printProd*.

The *Item* class obtained after the refactoring is shown, described in rool, in Figure [14](#_bookmark64), 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](#_bookmark66). 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](#_bookmark67), for didactic reasons. Notice that the method *printProd* now belongs to the class *Product*. The final class diagram is depicted in Figure [17](#_bookmark68).

# Conclusions

This paper illustrates how rewriting systems, in particular CafeOBJ, can be used to mechanise refactoring proofs, in the algebraic style developed by Corn´elio in [[8](#_bookmark78)]. 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-

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 get Description ^=

( 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 ^= (

@ ( var n : string @ (( self . prod . getName ) < n > ; print < n >) end

; ( var d : string @ (( self . prod . get Description ) < d > ; print < d >) end ; var m : Brand @ (( self . prod . get Brand ) < m > ; print < ( m .

name ) >) end ))) meth printData ^= (@ ( 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

, methodname = printProd : MethodName 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 ^= ( @ (( self . prod . printProd ) < >)) meth printData : MethodName ^= (@ ( print : ParCommand < self . acquisitionDate : Variable > ; ( print : ParCommand < self . purchasePrice : Variable > ; ( print : ParCommand < self . profit : Variable >

; ( print : ParCommand < ( self . supp : Variable . name : Variable ) > ; self . printProd : MethodName < >))))) end class Product : ClassName pri name : Variable : string ; ( pri brand : Variable : Brand : ClassName ; pri description : Variable : string ;) meth printProd ^= ( @ ( var n: Variable

: string @ ( self . getName : MethodName < n: Variable > ; print : ParCommand < n: Variable >) end ; ( var d: Variable : string @ ( self . getDescription : MethodName < d: Variable > ; print : ParCommand < d: Variable >) end ; var m: Variable : Brand : ClassName @ ( self . getBrand : MethodName < m: Variable > ; print : ParCommand < ( m: Variable . name : Variable ) >) end ))) ( meth getName : MethodName ^= (( res r: Variable : string ) @ ( r: Variable := self . name : Variable )) ( meth getBrand : MethodName ^= (( res r: Variable : Brand : ClassName ) @ ( r: Variable := self

. brand : Variable )) meth get Description : MethodName ^= (( res r: Variable : string ) @ ( r: Variable := self . description : Variable )))) end :

ClassDecls

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* =b (*•* **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* =b (**res** *r* : **string** *• r* := **self***.name*)

**meth** *getBrand* =b (**res** *r* : *Brand • r* := **self***.brand*)

**meth** *getDescription* =b (**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* =b ( *•* **self***.prod.printProd*())

**meth** *printData* =b (*• print*(**self***.acquisitionDate*); *print*(**self***.purchasePrice*); *print*(**self***.profit*); *print*(**self***.supp.name*); **self** *.printProd* ())

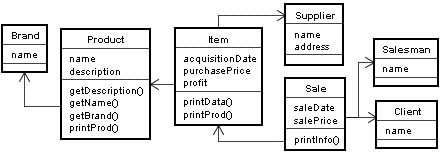
**end**

Figure 16. Classes *Product* and *Item* after the applcation of Rule 3 *⟨Move Method⟩*.

ertheless we have not implemented all refactorings proposed in [[8](#_bookmark78)], 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 re- garded as a contribution to the CafeOBJ community, as it could be considered

as a programming transformation case study, in the context of this rewriting sys- tem. On the other hand, for the refactoring community, this work suggests that rewriting systems could be considered as supporting tools to construct refactoring



environments.

Figure 17. Class diagram after retactorings.

The majority of refactoring rules and laws of classes includes side conditions. When deriving the refactoring rules, we considered that these conditions are satis- fied. 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 chal- lenge 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

1. Arnold, K. and J. Gosling, “The Java Programming Language,” Addison-Wesley, 1996.
2. Back, R. J. R., Procedural Abstraction in the Refinement Calculus, Technical report, Department of Computer Science, Abo - Finland (1987).
3. Cavalcanti, A. L. C. and D. Naumann, *A weakest precondition semantics for refinement of object-* *oriented programs*, IEEE Transactions on Software Engineering **26** (2000), pp. 713–728.
4. Cavalcanti, A. L. C. and D. A. Naumann, *A weakest precondition semantics for an object-oriented language of refinement*, in: J. M. Wing, J. Woodcock and J. Davies, editors, *FM’99 - Formal Methods, Volume II*, number 1709 in Springer LNCS, 1999, pp. 1439–1459.
5. Cavalcanti, A. L. C., A. Sampaio and J. C. P. Woodcock, *An Inconsistency in Procedures, Parameters and Substitution in the Refinement Calculus*, Science of Computer Programming **33** (1999), pp. 87–96.
6. Cinn´eide, M. O´ . and P. Nixon, *Automated application of design patterns to legacy code.*, in: *ECOOP Workshops*, 1999, p. 176.
7. Clavel, M., F. Dur´an, S. Eker, P. Lincoln, N. Mart´ı-Oliet, J. Meseguer and C. Talcott, “Maude Manual (Version 2.1),” SRI International (2004).
8. Corn´elio, M., “Refactoring as Formal Refinements,” Ph.D. thesis, Universidade Federal de Pernambuco (2004).
9. Dijkstra, E. W., “A Discipline of Programming,” Prentice Hall, 1976.
10. Fowler, M., “Refactoring: Improving the Design of Existing Code,” Addison-Wesley Object Technology Series, Addison-Wesley, 1999.
11. Goguen, J., T. Winkler, J. Meseguer, K. Futatsugi and J. Jouannaud, *Introducing OBJ*, in: J. Goguen, editor, *Applications of Algebraic Specification using OBJ*, Cambridge, 1993 .
12. Lano, K., J. Bicarregui and S. Goldsack, *Formalising design patterns*, in: D. Duke and A. Evans, editors, *1st BCS-FACS Northern Formal Methods Workshop, Ilkley, UK*, Electronic Workshops in Computing, 1996.

URL <http://www.ewic.org.uk/ewic/workshop/view.cfm/NFM-96>

1. Larman, C., “Applying UML and Patterns,” Prentice Hall, 2002.
2. Lira, B. O., A. L. C. Cavalcanti and A. C. A. Sampaio, *Automation of a Normal Form Reduction Strategy for Object-oriented Programming*, in: *Proceedings of the 5th Brazilian Workshop on Formal Methods*, 2002, pp. 193–208.
3. Meyer, B., “Object-Oriented Software Construction,” Prentice Hall, 1997, 2 edition.
4. Morgan, C. C., “Programming from Specifications,” Prentice Hall, 1994, 2 edition.
5. Morgan, C. C., K. A. Robinson and P. H. B. Gardiner, On the Refinement Calculus, Technical Monograph PRG-70, Oxford University Computing Laboratory, Wolfson Building, Parks Road, Oxford, UK (1988).
6. Nakagawa, A., T. Sawada and K. Futatsugi, “CafeOBJ User’s Manual - ver. 1.4.2, 1999. Dispon´ıvel em [http://www.ldl.jaist.ac.jp/cafeobj/doc/,”](http://www.ldl.jaist.ac.jp/cafeobj/doc/) (1999).

URL <http://www.ldl.jaist.ac.jp/cafeobj/doc/>

1. Opdyke, W. F., “Refactoring Object-Oriented Frameworks,” Ph.D. thesis, Urbana-Champaign, IL, USA (1992).

URL <citeseer.ist.psu.edu/article/opdyke92refactoring.html>

1. Silva, A. L., M. M. Menezes and L. Silva, *Using* C*afe*OBJ *to implement a reduction strategy in the context of hardware/software partitioning.*, Electronic Notes in Theoretical Computer Science **95** (2004), pp. 63–82.
2. Tokuda, L., “Evolving Object-Oriented Designs With Refactorings,” Ph.D. thesis, University of Texas at Austin (1999).