Electronic Notes in Theoretical Computer Science 70 No. 3 (2002)
URL: http://www.elsevier.nl/locate/entcs/volume70.html 20 pages

Refactoring by Transformation

Márcio Cornélio [1]

Centre of Informatics
Federal University of Pernambuco
P.O. Box 7851, 50732-970 Recife-PE, Brazil

Ana Cavalcanti [2]

Computing Laboratory
University of Kent at Canterbury
Canterbury, Kent CT2 7NF, UK

Augusto Sampaio [3]

Centre of Informatics
Federal University of Pernambuco
P.O. Box 7851, 50732-970 Recife-PE, Brazil

Abstract

In this paper we present how refactoring of object-oriented programs can be accomplished by using refinement. Our approach is based on algebraic laws of an object-oriented language for refinement similar to Java. We follow a strategy involving data and algorithmic refinement of classes.

1 Introduction

Object-oriented programming has been acclaimed as a means to obtain software that is easier to modify [16]. However, maintaining an object-oriented program often requires structural changes such as moving attributes and methods between classes, and partitioning a complex class into several ones. This activity is called *refactoring* [13]. Work on refactoring usually describes the steps used for program modification in a rather informal way [13,18,20]. Other refactorings are still on the minds of object-oriented programmers that use them intuitively.

[1] Email: mlc2@cin.ufpe.br
[2] Email: A.Cavalcanti@ukc.ac.uk
[3] Email: acas@cin.ufpe.br

The work presented in this paper is in the context of the Co-op (Calculus of Object-Oriented Programming) project, jointly funded by CNPq and NSF, which aims at proposing and proving basic, design and compilation laws for object-oriented programming. In particular, we are interested in formal refactoring of object-oriented programs and formalisation of design patterns.

In our approach, refactoring is achieved by the application of object-oriented programming rules that transform programs preserving correctness. In this paper we present rules for refactoring of object-oriented programs. These rules precisely indicate the modifications that can be done to a program, with corresponding proof obligations. With the use of such rules, a program update is justified and documented. Moreover, in a rule-based approach a suite of tests is not compulsory. Also, it is not necessary to rely on compiling in order to check if errors were introduced. Clearly, to ensure the correctness of the transformations, each rule must be formally justified itself.

In this paper, we consider a subset of sequential Java called rool [6,7,5], an acronym for Refinement Object-Oriented Language. We prove each rule from more basic algebraic laws [3,4,11] of rool. These laws constitute an algebraic semantics of the language, and are themselves justified [11] from a denotational semantics in the weakest precondition style [6,5]. Some rules express very simple transformations, and are proved direct from the weakest precondition semantics.

This paper is organised as follows. We first present an overview of rool with some basic laws. After that, we present rules for object-oriented programs refactoring. This is followed by the derivation of a refactoring rule from the basic laws of rool. Finally, we summarise our results and indicate some directions for future work.

2 rool and Its Laws

rool [6,7,5] is an object-oriented language based on Java, that allows reasoning about object-oriented programs. A program in rool is a sequence of classes followed by a main command. Classes are related by single inheritance. Attributes can be private, protected, or public, like in Java. Methods are public and can be recursive; they are defined using procedure abstractions [1,8] and can have the form **val** $x : T \bullet c$, **res** $x : T \bullet c$, or **vres** $x : T \bullet c$, which correspond to the call-by-value, call-by-result, and call-by-value-result parameter passing mechanisms.

Presently, rool has a copy semantics rather than a reference semantics. Of course, pointers are ubiquitous in practice. We decided, however, to concentrate initially on other aspects of object-orientation like inheritance, dynamic binding, visibility, and type tests and casts. The results we obtain are still valid in the presence of pointers, but, in general, would need to be revised to consider sharing; this is left as future work. In particular, the use of pointers does not intrude in the formalisation of many refactorings we have already

$$cds, C \rhd \qquad (\mathbf{res}\ vl : T \bullet c[vl/x])(x) = c$$

provided
vl is a list of fresh variables whose types are given by the list T, where the types of the variables in list x are subtypes of, or equal to, the types in T.

Fig. 1. Law $\langle$*pcom elimination-res*$\rangle$

considered.

Classes are declared in rool as follows.

class N_1 **extends** N_2
  pri $x_1 : M$;
  prot $x_2 : T_2$;
  pub $x_3 : T_3$;
  meth $m \mathrel{\widehat{=}} (pds \ \bullet c)$
  new $\mathrel{\widehat{=}} (pds \ \bullet c)$
end

The class N_1 is a subclass of N_2; this is expressed by the clause **extends** which determines the immediate superclass of N_1. If this clause is omitted, N_1 extends the class **object** which is a superclass of every class in rool. The **pri**, **prot**, and **pub** clauses introduces private, protected and public attributes of N_1, respectively. The visibility mechanism is similar to that of Java. The method introduced by **meth** $m \mathrel{\widehat{=}} (pds \ \bullet c)$ has name m and its body is $(pds \ \bullet c)$, where pds is a parameter declaration as explained above. All methods in rool are considered to be public. Initialisers (class constructors) are declared with the **new** clause.

A set of algebraic laws for rool has already been defined in [3,4,11]. Many laws of commands are similar to the laws of imperative programming presented, for example, in [15], but rool has laws that support object-oriented features such as method calls, classes, and type cast and test.

Some laws involving object-oriented features of rool depend on the context in which they are applied. We use the notation $cds, N \rhd c \sqsubseteq c'$ to mean that in the class N declared in the sequence of class declarations cds, the command c is refined by c'. In other words, c' satisfies every specification satisfied by c. A refinement relation between class declarations, denoted by $cds_1, c \rhd cds_2 \preceq_= cds_2'$, is defined for when, in the program $cds_1\, cds_2 \bullet c$, the class declaration sequence cds_2 can be replaced with cds_2'. We omit the command c when there are no restrictions on it. These refinement relations are formalised in [6].

Some conditions may be associated to a law application; we use the following notation to express the conditions. Using the notation $C' \leq_{cds} C$ we

$$cds, C \rhd \qquad le.m(e) = \{le \neq \textbf{null}\};\ \ D.m[le/\textbf{self}](e)$$

provided
$cds, C \rhd le : D$, *notRedefinedMethod*(cds, D, m), *methodSuperFree*(m), *and all attributes free in* $D.m$ *are public.*

Fig. 2. Law ⟨*method call elimination*⟩

express that class C' is a subclass of C in the sequence of class declarations *cds*. Expressions *le* (allowed to appear as target of assignments and method calls, and as result and value-result arguments) are called left expressions. The notation $cds, N \rhd e : T$ asserts that in the class N present in *cds*, the expression e has type T. These notations are formally defined in [7,5].

In the following we present three basic algebraic laws of rool in order to briefly illustrate its algebraic semantics. The first law allows changing a non-parameterised command into a parameterised one. For instance, we use law ⟨*pcom elimination-res*⟩ (Figure 1) to introduce a parameterised command with result arguments.

To illustrate the behaviour of a method call in rool, we present a simplified version of the law for method calls (Figure 2). The first condition to be satisfied for the application of this law is that the type of *le* is D. This law applies when the call $le.m$ does not require dynamic binding. The function *notRedefinedMethod*(cds, D, m) assures that the method m declared in class D is not redefined in any of its subclasses. In order to assure that **super** does not appear in the body of method m of class D which is present in *cds*, we use the function *methodSuperFree*(cds, D, m). In this situation, if *le* does not denote a null object, we can replace the call with the body of m in D, after a few changes. The command $\{le \neq \textbf{null}\}$ is an assumption; it checks if *le* is non-null and aborts if it is null. The notation $D.m$ stands for the body of method m in class D. With the substitution, we replace every occurrence of **self** in $D.m$ with *le*; every reference to a method n and to an attribute a, declared in the class itself, must be replaced with $le.n$ and $le.a$, as well. This is indicated by the notation $D.m[le/\textbf{self}]$. If the attribute a is private, this replacement would cause a compilation error. This is the reason for requiring the attributes to be public.

As an example of a law for classes, we present the law for introducing private attributes in an existing class (Figure 3). With this law we can introduce a fresh private attribute or remove a private attribute that is not used. The notation **pri** $a : T$; *ads* denotes the set of attribute declarations containing **pri** $a : T$ and all declarations in *ads*; the declaration of operations (object initialiser and methods) is denoted by *mts*. For declaring the new attribute a in class C, a cannot be declared in C nor in its superclass and subclasses. For removing the attribute a from C; it cannot be referred to inside C. The arrows $\rightarrow$ and $\leftarrow$ annotate the conditions required for the right to left and

class C *ads mts* **end** $=_{cds,c}$ **class** C **pri** $a : T$; *ads mts* **end**
provided $(\rightarrow)$ $notDeclaredAttr(cds, C, a)$, $notDeclaredAttrSuperclass(cds, C, a)$, $notDeclaredAttrSubclasses(cds, C, a)$ $(\leftarrow)$ $notReferredAttribute(cds, C, a)$

Fig. 3. Law $\langle$*introduce private attribute*$\rangle$

left to right application of the law, respectively. A comprehensive set of laws can be found in [3,4,11].

3 Refactoring Rules

Refactoring programs has been an activity in which transformed programs are not guaranteed to preserve the behaviour of the original programs. Refactoring usually relies on program compilation and on a suite of tests. Program compilation intends to detect any type error introduced during the refactoring; the suite of tests is used in order to check that no functional error was introduced. These two activities, however, are insufficient to assure that a refactored program preserves the behaviour of the original program as compilation is related only to type errors and a suite of tests is not reliable enough.

In our rule-based approach, refactoring corresponds to the application of rules that can only be applied if a set of conditions associated to each rule is satisfied. Moreover, each rule is proved in the weakest precondition semantics of rool [6,7,5] or is derived from laws [10]. Here we present rules for program transformation that correspond to refactorings described by Fowler [13].

Extract Method. This rule turns a command c_2, which is present in a method, into a new method m_2. The occurrences of the command in the original method m_1 are replaced by calls to the new method. This rule is represented in Figure 4.

The variable a represents the free variables that appear in the command c_1 of method m_1 which are not attributes of class A. This is obtained by the function $freeLocalVariablesExcptAttr(N, m, c)$ that gives the free variables of a command c that occurs in a method m which is present in a class N. We use the notation $c[c']$ to express that in the command c there is an occurrence of the command c'. The notation $c[exp]$ expresses that we have an occurrence of the expression exp in the command c. In the same way, $exp_1[exp_2]$ expresses that the expression exp_2 occurs in the expression exp_1. On the left-hand side of this rule, $c_1[c_2[a]]$ represents the occurrence of the variable a in the command c_2 that occurs in the command c_1. We assume that a is read and written in the command c_2. The class C that appears as argument of the **extends** clause in this rule, and in the others that follow, is present in the sequence of

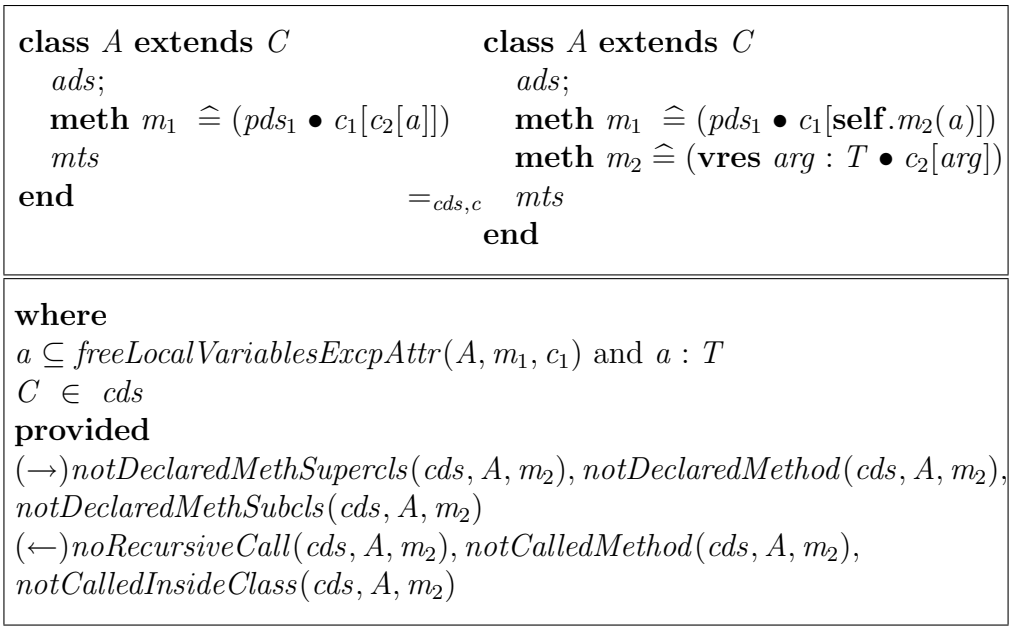

Fig. 4. Rule ⟨*Extract Method*⟩

class declarations *cds*.

On the right-hand side of this rule, in the command c_1 of method m_1, the method call **self**.$m_2(a)$ replaces the command $c_2[a]$. In the command c_2 of method m_2, the variable a is replaced with the argument *arg*. The parameter passing mechanism for *arg* is value-result because we assume that the variable a is read and written on the left-hand side of the rule. Variables that are only read or written are treated as special cases of the situation we describe here. If a variable is only read, it should be passed as a value argument. A variable that is only written should be passed as a result argument.

This rule applies when the method m_2 is new: not declared in the superclass of A, in A itself, nor in any of its subclasses. 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 should be no recursive call in the method m_2. Also, the method m_2 cannot be called in *cds*, c, and A. These conditions and others that follow are formalised in [10].

The application of this rule improves the legibility and maintenance of a class. If a command is present in several methods of a class, this command can be extracted into a new method. The methods in which the command occurred just call this new method. Consequently, these methods are shorter than they were before the application of the rule. This improves legibility. On the other hand, changing a command that appears in several methods can lead to inconsistency if one of the methods is not properly modified, whereas changing only one method avoids inconsistency. This improves maintenance.

Move Method. One of the most disseminated practices along the development of object-oriented programs is moving methods between classes. The

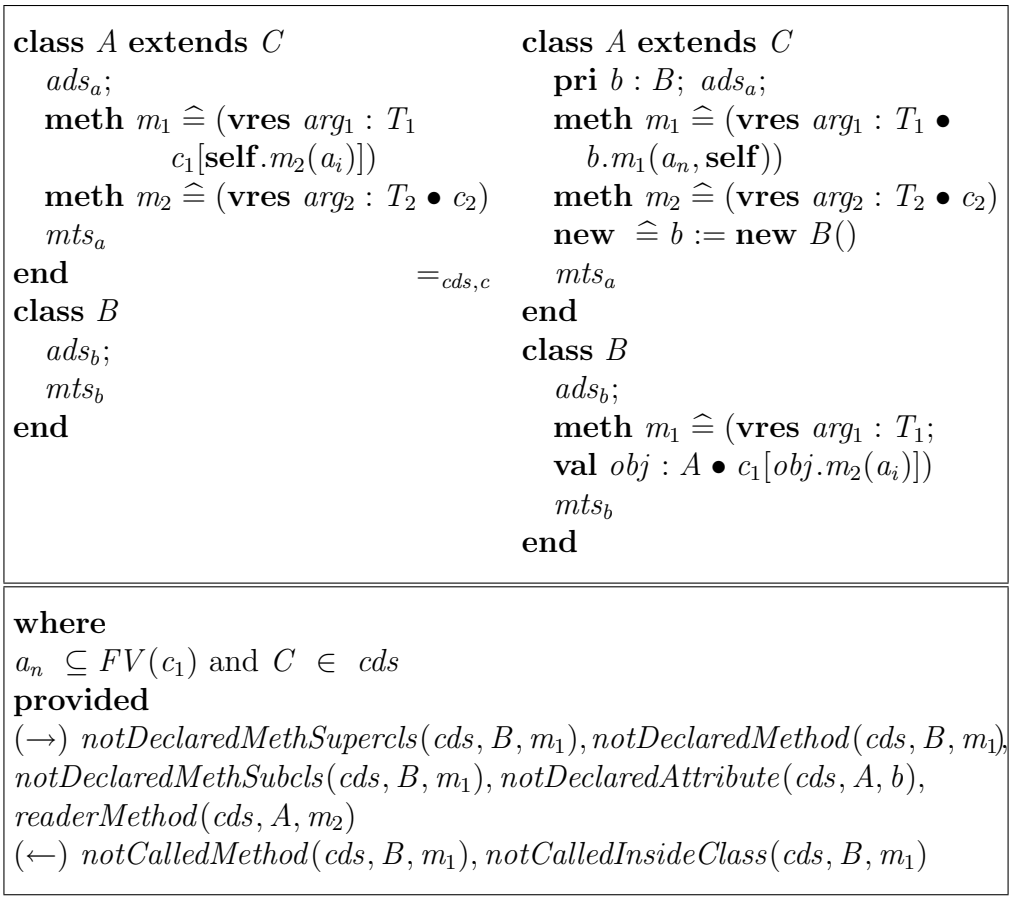

Fig. 5. Rule ⟨*Move Method*⟩

rule ⟨*Move Method*⟩ (Figure 5) moves the method m_1 declared in class A to the class B. This can only be done if m_1 is not declared in B. Also, it cannot be declared in the superclass of B nor in any of its subclasses. The attribute b, whose type is B, cannot be already declared in class A. Another condition is that the method m_2 is a reader method: it only reads the attributes of class A.

The attribute b of type B is introduced in class A so that it can call on b the method that was moved from A to B. The attribute b has to be fresh, not declared in A. This attribute is initialised in the method **new** with an object of class B. We turn the old method m_1 into a delegating method, it just calls the method m_1 of class B, thus delegating its previous functionality to m_1. The argument a_n of this method call are the free variables of the command c_1 originally present in the method m_1 of class A. The extra parameter *obj* of method m_1 of class B is target of calls to methods of class A. These methods were originally called having **self** as target. To apply this rule from right to left, we have to guarantee that m_1 is not called in *cds*, c, and B.

The method m_1 of class B, on the right-hand side, calls the method m_2 of class A with the object denoted by *obj* as target of the call. The parameter *obj* has a copy of the object denoted by **self** because the call to the method

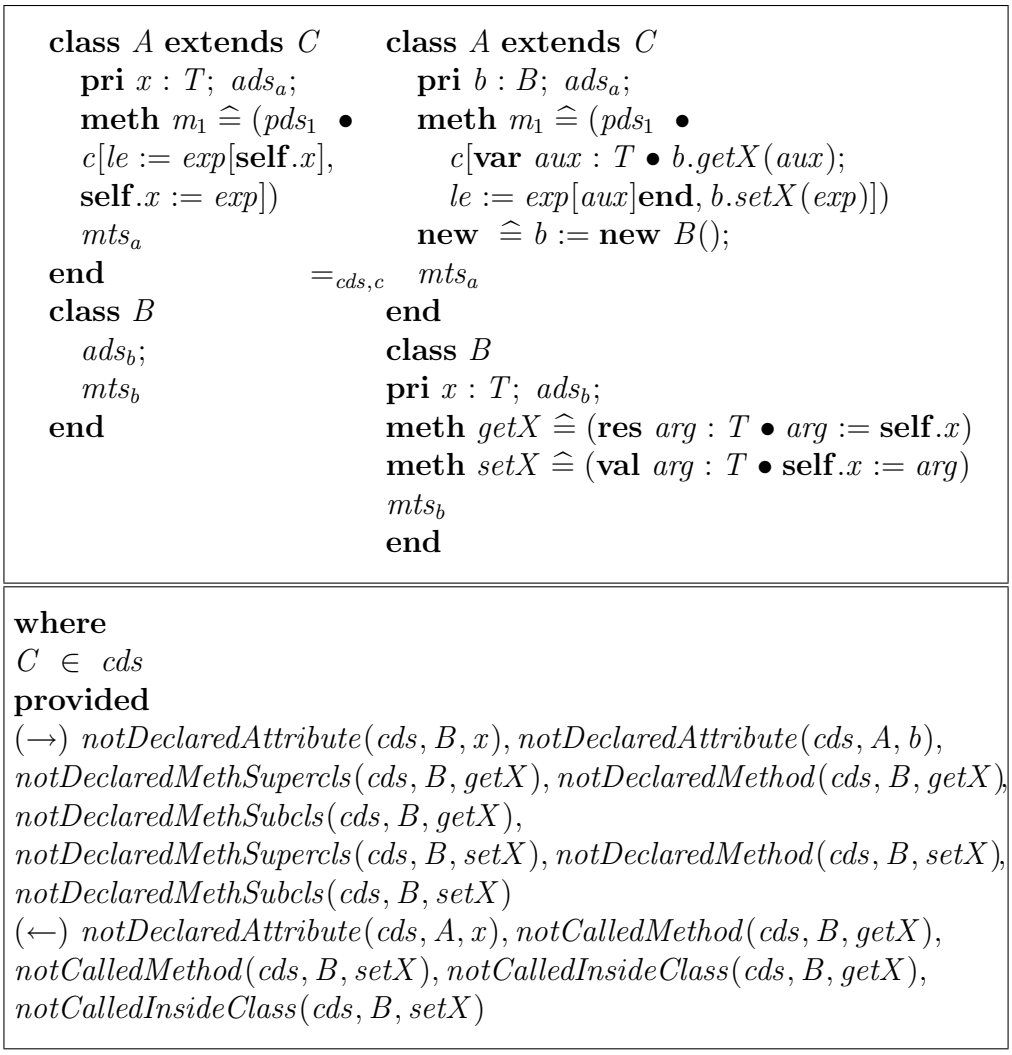

Fig. 6. Rule ⟨*Move Attribute*⟩

m_1 that occurs in class A passes **self** as argument. If the method m_2 could change the values of the attributes of the object on which the call is done, these changes would not be reflected on the object passed as argument in the call $b.m_1(a_n, \mathbf{self})$. This is the reason for requiring the method m_2 to be a reader method.

The purpose of this refactoring is to reduce the dependence between classes. This may occur when a method is more used by methods of another class than by those of the class in which it is declared. This suggests that the method should be moved to another one. This can improve reuse of a class as this is more independent of the functionality provided by another class. The coupling between classes is reduced.

Move Attribute. Moving attributes between classes is also a common activity done during program development. If an attribute is more used by methods of another class—through getting and setting methods—than by those of the

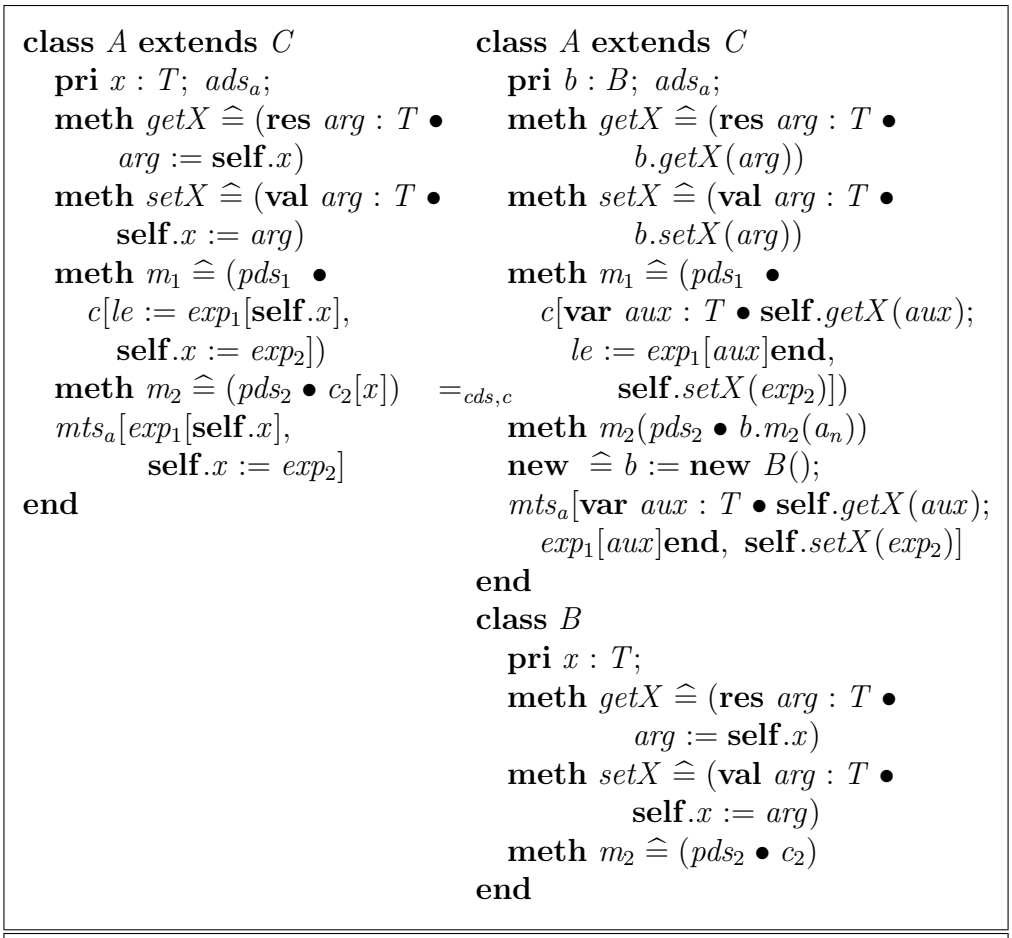

Fig. 7. Rule ⟨*Extract Class*⟩

class in which it is declared, this suggests that the attribute belongs to another class, it is intrinsic to the concept described by another class. We can apply the rule ⟨*Move Attribute*⟩ (Figure 6) for moving attributes between classes. The class A, from which we move the attribute x, after the application of this rule, has an attribute b of type B that has to be not declared in A. This attribute is initialised in the method **new** of class A with an object of B. The attribute to be moved cannot be declared in class B, the target class. Getting and setting methods are declared in B in order to access x. These methods could not be declared in the superclass of B, in B itself, and in any of its subclasses. Commands that read and write the attribute x in the source class

class A **extends** C	**class** A **extends** C
pri $x : T;\ ads_a$	**pri** $x : T;\ ads_a$
meth $m_1\ \widehat{=}\ (pds_1\ \bullet$	**meth** $m_1\ \widehat{=}\ (pds_1\ \bullet$
$c[le_1 := exp_1[\mathbf{self}.x],$	$c[\mathbf{var}\ aux : T \bullet \mathbf{self}.getX(aux);$
$\mathbf{self}.x := exp_2])$ $=_{cds,c}$	$le_1 := exp_1[aux]\mathbf{end},\ \mathbf{self}.setX(exp_2)])$
mts_a	**meth** $getX\ \widehat{=}\ (\mathbf{res}\ arg : T \bullet arg := \mathbf{self}.x)$
end	**meth** $setX\ \widehat{=}\ (\mathbf{val}\ arg : T \bullet \mathbf{self}.x := arg)$
	mts_a
	end

where
$C \in cds$
provided
$(\rightarrow)$
$notDeclaredMethSupercls(cds, A, getX), notDeclaredMethod(cds, A, getX),$
$notDeclaredMethSubcls(cds, A, getX),$
$notDeclaredMethSupercls(cds, A, setX), notDeclaredMethod(cds, A, setX),$
$notDeclaredMethSubcls(cds, A, setX)$
$(\leftarrow)$ $notCalledMethod(cds, A, getX), notCalledInsideClass(cds, A, getX),$
$notCalledMethod(cds, A, setX), notCalledInsideClass(cds, A, setX)$

Fig. 8. Rule $\langle$*Self Encapsulate Field*$\rangle$

A do these operations using the getting and setting methods introduced in B. In order to call these methods in class A, we use the attribute b. To apply this rule from right to left, x cannot be an attribute of class A, and the getting and setting methods cannot be called inside class B, nor in cds and c.

Extract Class. Another common practice is partitioning a class into several ones. This can be done by using the rule $\langle$*Extract Class*$\rangle$ (Figure 7). To apply this rule, we have to satisfy the condition that the class B, which is being extracted from class A, is not declared in the sequence of class declarations cds. Also, the attribute b cannot be already declared in A.

Some attributes declared in A, which are represented here by x, have to be declared in B along with getting and setting methods. An attribute b of type B is introduced in the class A and initialised in the **new** method with an object of B. This attribute is the target of calls to the getting and setting methods of class B. Original getting and setting methods that act on attributes that are declared in B must use the corresponding methods of class B. In this way, there are no impacts to clients of class A that calls the getting and setting methods. Methods declared in A that act on attributes present in B uses the getting and setting methods present in A itself. Applying this rule from right to left requires that the class B is client only of class A, and that B is not type of parameters of methods declared in A. Also, the attribute x

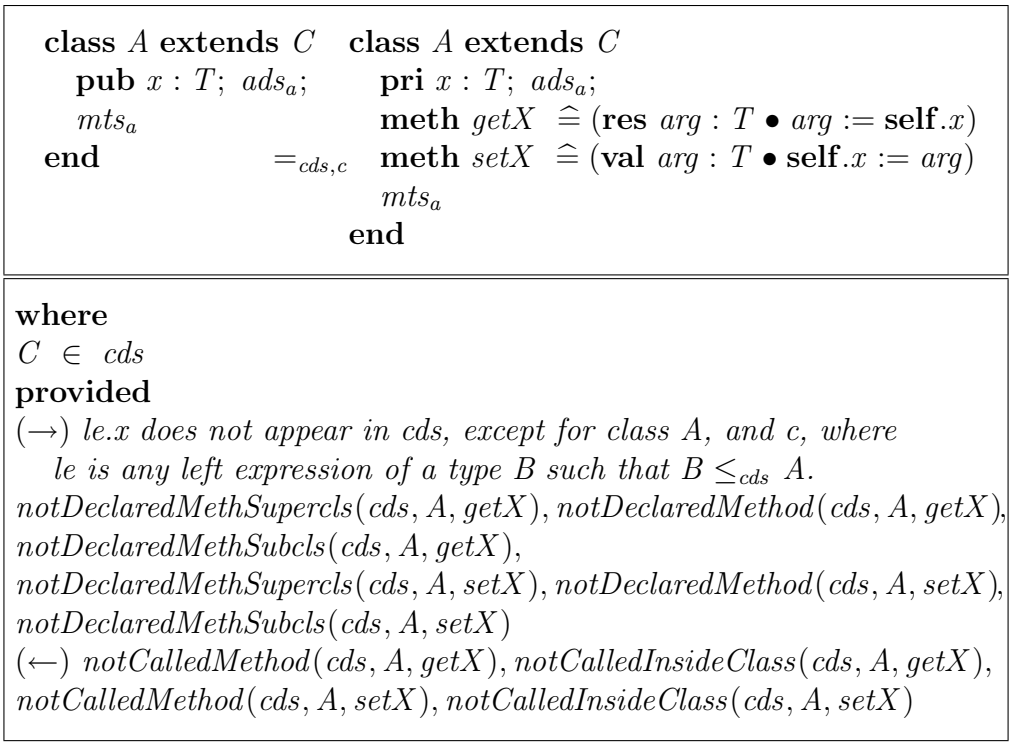

Fig. 9. Rule ⟨*Encapsulate Field*⟩

cannot be already declared in class A.

This refactoring improves reuse and extensibility. Classes should describe single concepts of the real world. Describing different concepts in one class mingles attributes and methods that are intrinsic to distinct concepts. This can result in complex classes that are hard to reuse and extend. Extracting a class from a complex class simplifies the last one and favours reuse and extensibility of both resultant classes.

Self Encapsulate Field. Accessing attributes of a superclass from a subclass is only allowed if the attributes are declared as protected or public. In order to allow subclasses—and all other classes—to have access to private attributes already declared in a superclass, getting and setting methods have to be declared. The rule ⟨*Self Encapsulate Field*⟩ (Figure 8) introduces getting and setting methods for an attribute x declared in class A. In the method m_1 on the left-hand side of the rule, the attribute x appears in the expression exp_1 and also there is an assignment to this attribute. On the right-hand side of the rule, the occurrence of **self**.x in the 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. To apply this rule from left to right, the method *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, and A.

class A **extends** C
ads_a;
meth $m_1 \mathrel{\widehat{=}} (pds_1 \bullet c_1$;
  var $x, y : T_1, T_2 \bullet c[x]$;
  self.$m_2(x, y)$**end**)
meth $m_2 \mathrel{\widehat{=}}$
  (**val** $arg_1 : T_1$; **res** $arg_2 : T_2 \bullet$
  $c_2[arg_1, arg_2]$)
end
	$=_{cds,c}$	**class** A **extends** C
		ads_a;
		meth $m_1 \mathrel{\widehat{=}} (pds_1 \bullet c_1$;
		  var $y : T_2 \bullet$ **self**.$m_2(y)$**end**)
		meth $m_2 \mathrel{\widehat{=}} ($**res** $arg_2 : T_2 \bullet$
		  var $x : T_1 \bullet$ **self**.$m_3(x)$;
		  $c_2[x, arg_2]$ **end**)
		meth $m_3 \mathrel{\widehat{=}} ($**res** $arg : T_1 \bullet c[arg])$
		end

where
$C \in cds$
provided
$(\rightarrow)$ $notDeclaredMethSupercls(cds, A, m_3)$, $notDeclaredMethod(cds, A, m_3)$, $notDeclaredMethSubcls(cds, A, m_3)$, $calledOnlyInsideClass(cds, A, m_2)$
$(\leftarrow)$ $notCalledMethod(cds, A, m_3)$, $notCalledInsideClass(cds, A, m_3)$

Fig. 10. Rule ⟨*Replace Parameter with Method*⟩

Encapsulate Field. Public attributes reduces the modularity of object-oriented programs. They break data hiding, separating data from behaviour as client classes can have direct access to them. The rule ⟨*Encapsulate Field*⟩ (Figure 9) hides a public attribute and provides getting and setting methods for it. If a client of class A directly reads the attribute x, this reference to x must be replaced with a call to the method *getX*; if a client directly assigns to x, this assignment must be made using the method *setX*. To apply this rule from left to right there must be no direct access to the attribute x. Also, the methods *getX* and *setX* cannot be declared in the superclass of A, in A itself, nor in any subclass of A. To apply the rule ⟨*Encapsulate Field*⟩, from right to left, the methods *getX* and *setX* cannot be called in *cds*, c, and A.

Replace Parameter with Method. If a method m_2 can get a value that is passed as argument to it by means of a method call, this really should be done. The consequence of this change is that the parameter list of the method m_2 is reduced, thus improving readability. This is the purpose of the rule ⟨*Replace Parameter with Method*⟩ (Figure 10). The method m_1 on the left-hand side of the rule calls the method m_2 passing x, a locally declared variable, as a value argument. After application of this rule, the method m_1 still calls m_2. The major changes occur in m_2. The command in which x occurs in the original m_1 is extracted into the method m_3. Instead of passing the result of the call to m_3 as an argument in the call to m_2, the method m_1 just calls m_2 which is responsible for calling m_3. Consequently, the number of parameters is lesser than in the original m_2. The local variable x present in the method

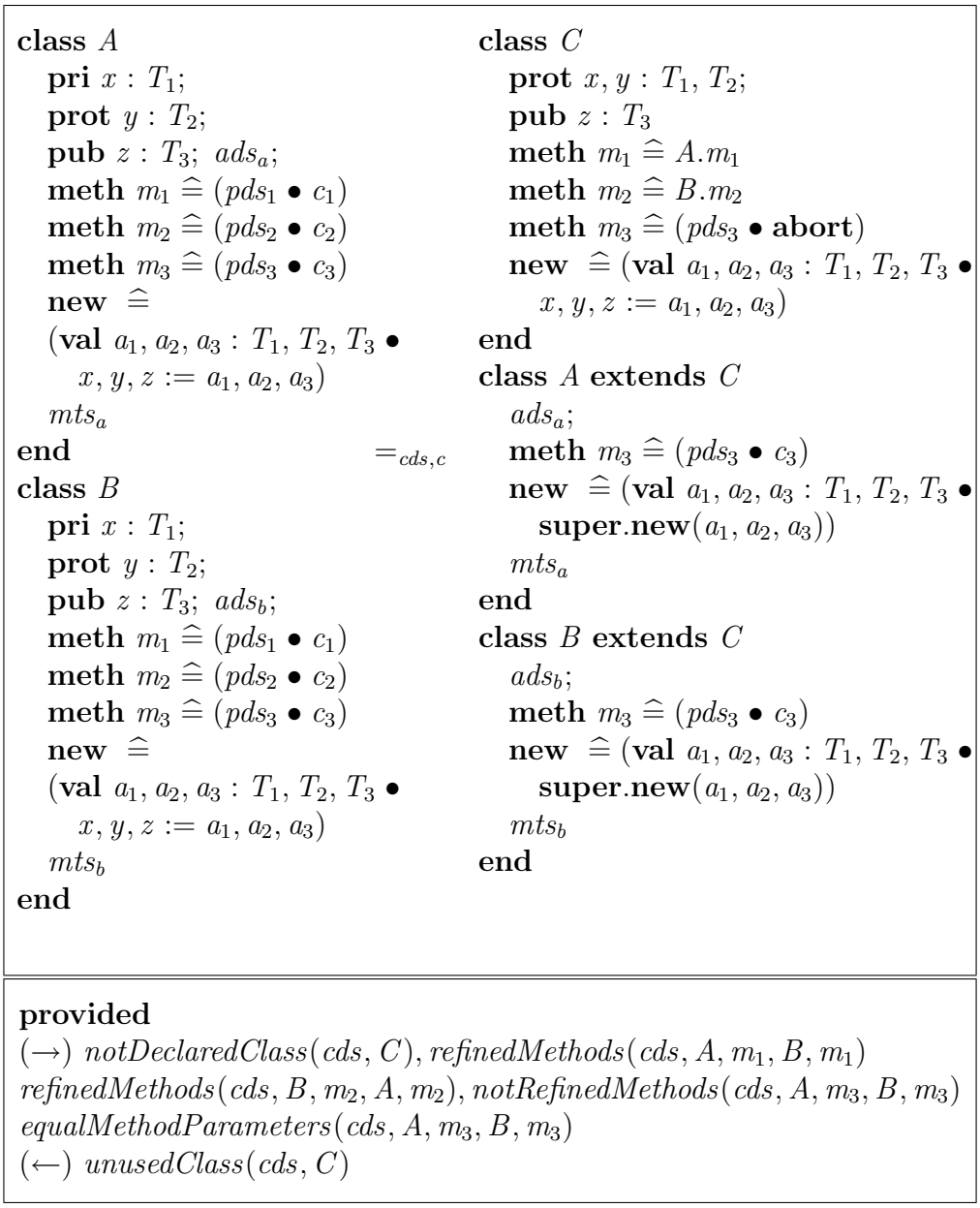

Fig. 11. Rule ⟨*Extract Superclass*⟩

m_2, on the right-hand side, replaces the occurrences of arg_1 in the original m_2. In order to apply this rule, from left to right, the method m_3 cannot be declared in the superclass of A, in A itself, nor in any of its subclasses. Also, the method m_2 cannot be called outside class A. For applying this rule from right to left, the following conditions must be satisfied: the method m_3 is not called in *cds*, c, and A.

Extract Superclass. As software systems often follow similar patterns, their development can be guided to explore what they have in common. This avoids looking for solutions that have already been encountered. On the other hand,

duplicate code is a hindrance to software maintenance. This may occur when classes have similar behaviour and indicates that an inheritance hierarchy can be extracted.

The rule ⟨*Extract Class*⟩ is adequate for delegation. Inheritance is adequate when two classes share behaviour. The rule ⟨*Extract Superclass*⟩ (Figure 11) extracts attributes and methods common to two classes A and B, declaring them in a new class C that is declared to be the superclass of A and B. Private attributes are declared as protected in C; protected and public attributes are declared with the same visibility. The class C has to be not declared. To choose which method body is moved to the superclass between two common methods of classes A and B, we have to verify which method is the most abstract. For instance, the body of method m_1 of class C is the body of method m_1 of class A because this is more abstract than method m_1 of class B. This is checked by the condition $\mathit{refinedMethods}(\mathit{cds}, A, m_1, B, m_1)$ that checks that $A.m_1$ is refined by the $B.m_1$. Recall that the notation $A.m_1$ stands for the body of method m_1 in class A. In the case of method m_2, $B.m_2$ is refined by $A.m_2$, indicating that the body of method m_2 in class C is that of $B.m_2$. If two methods have the same signature, but there is no refinement between them, we declare a method in the superclass with the same signature and define the body to be **abort**. This is the case of method m_3. The method **new** of class C initialises the common attributes of A and B that were initialised in the original classes. The method **new** of class A and B just calls the **new** of the superclass, having **super** as target of the call. To remove the class C from the sequence of class declarations *cds*, we have to assure that it is not used in *cds* and c: it is not type of any variable and it has no subclasses.

Refactoring rules as those presented in this Section can be derived from the basic laws proposed for rool [3,4], and other laws, also presented in [9]. We give an example of such derivation in the next section.

4 Correctness

Here we present the derivation of the rule ⟨*Extract Class*⟩. Initially the class A contains attributes and methods that describe a specific concept (see left-hand side of Figure 7). The first step of the derivation is to introduce the class B to describe this specific concept.

To introduce a new class, we use the law ⟨*introduce class*⟩ to declare the class B with the attributes that describe the concept we want to extract from A. These attributes are declared as public because the law for introducing method calls requires attributes that appear in methods of the target object to be public. We can declare B only with these attributes and then, applying a law for method introduction, we can declare getting and setting methods for these attributes. However, as we are declaring a new class, we can do this

$$
\begin{array}{l}
\textbf{class } A \textbf{ extends } C \\
\quad \textbf{pri } x, b : T, B;\ ads_a; \\
\quad \textbf{meth } getX \mathrel{\widehat{=}} (\textbf{res } arg : T \bullet arg := b.x)\textbf{end} \\
\quad \textbf{meth } setX \mathrel{\widehat{=}} (\textbf{val } arg : T \bullet b.x := arg)\textbf{end} \\
\quad \textbf{meth } m_1 \mathrel{\widehat{=}} (pds_1 \ \bullet c[le := exp_1[b.x], b.x := exp_2])\textbf{end} \\
\quad \textbf{new} \mathrel{\widehat{=}} b := \textbf{new } B() \\
\quad mts_a[exp_1[b.x], b.x := exp_2] \\
\textbf{end}
\end{array}
$$

Fig. 12. An intermediary class A

with all attributes and necessary methods. The class B is as follows.

$$
\begin{array}{l}
\textbf{class } B \\
\quad \textbf{pub } x : T; \\
\quad \textbf{meth } getX \mathrel{\widehat{=}} (\textbf{res } arg : T \bullet arg := \textbf{self}.x)\textbf{end} \\
\quad \textbf{meth } setX \mathrel{\widehat{=}} (\textbf{val } arg : T \bullet \textbf{self}.x := arg)\textbf{end} \\
\textbf{end}
\end{array}
$$

Afterwards, we turn the class A into a client of class B by declaring an attribute of type B in A. In fact, we use the new attribute to relate the attributes of A to those declared in B. This relation is established by a coupling invariant. The law $\langle$*introduce attribute-coupling invariant*$\rangle$ allows introducing attributes and requires their types to be classes declared in *cds*. The attributes cannot be declared in the superclass of A nor in any of its subclasses. We apply this law to add the attribute b of type B. The coupling invariant CI below is used to relate the attribute x of class A with attribute x of class B.

$$CI \mathrel{\widehat{=}} x = b.x$$

It guarantees that the values or objects recorded in x are the same as those in the attribute b of class B.

The law $\langle$*introduce attribute-coupling invariant*$\rangle$ changes not only the attributes of a class, but also its methods. Every condition is extended with the coupling invariant, and assignments are extended by modifications to the new variables that maintain the coupling invariant. The modifications follow the laws for data refinement in [17]. Afterwards, we refine the commands, in the traditional way, so that original attributes do not occur in them, but only the attributes declared in B accessed through b. The assignment $\textbf{self}.x := arg$ in the method *setX* becomes $\textbf{self}.x, b.x := arg, arg$. Then, it is diminished to the assignment $b.x := arg$. Assignments like $le := c[\textbf{self}.x]$ become $le := c[b.x]$. The class A after all these changes is presented in the Figure 12.

Now we have to introduce calls, in class A, to the getting and setting methods declared in B. We proceed in the following way: we introduce a local variable that substitutes the variable we want to pass as argument. If

the argument is a result one, the last command in the local variable block is an assignment of the variable introduced in this block to the variable that was substituted.

$$\mathbf{var}\ p\_arg : T \bullet p\_arg := b.x;\ arg := p\_arg\ \mathbf{end}$$

Then, we introduce a parameterised command that corresponds to the local variable block. Based on this parameterised command we introduce a method call. All these steps are carried out by using appropriate laws.

The assignment $arg := b.x$ present in method *setX* is the same as the one of method *getX* of class B, so we can introduce a method call. Applying law $\langle$*pcom elimination - res*$\rangle$ in Figure 1, from left to right, we obtain the following parameterised command.

$$(\mathbf{res}\ arg : T \bullet arg := b.x)(arg)$$

To introduce the method call, we need a program in the same format as the one on the right-hand side of law $\langle$*method call elimination*$\rangle$ in Figure 2. We have the same parameterised command as the body of method *getX* of class B. We need to introduce the assumption $\{b \neq \mathbf{null}\}$. This is carried out using a class invariant: a predicate that is valid along the whole lifetime of an object. Above, the class invariant is $b \neq \mathbf{null}$: if we begin the execution of a method in a state in which $b \neq \mathbf{null}$, its execution terminates without changing this fact. Moreover the initialiser of class A establishes $b \neq \mathbf{null}$. In other words, **null** is not assigned to b in any method of A.

With this we can introduce the assumption we need. After these steps, we have the following command

$$\{b \neq \mathbf{null}\}(\mathbf{res}\ arg : T \bullet arg := b.x)(arg)$$

which is in the format required by law $\langle$*method call elimination*$\rangle$. Then, applying this law, we introduce the method call $b.getX(arg)$ in the method *getX* of class A. Similar steps are followed to introduce calls to the method *setX* of class B.

We further refine assignments in m_1 so that they make use of the *setX* and *getX* methods. For an assignment $le := exp_1[b.x]$, we declare a variable *aux* to be used as argument for *getX*. We assign to it the expression $b.x$.

$$\mathbf{var}\ aux : T \bullet aux := b.x;\ le := exp_1[b.x]\ \mathbf{end}$$

As the expression $b.x$ is assigned to *aux* and appears in $le := exp_1[b.x]$, we can replace $b.x$ with *aux*. We obtain the following program.

$$\mathbf{var}\ aux : T \bullet aux := b.x;\ le := exp_1[aux]\ \mathbf{end}$$

All these changes are carried out using laws that we omit here due to lack of space, but that can be found elsewhere [11,9]. The assignment $aux := b.x$

```
class A extends C
  pri b : B; ads_a;
  meth getX ≙ (res arg : T • b.getX(arg))
  meth setX ≙ (val arg : T • b.setX(arg))
  meth m_1 ≙ (pds_1 • c[var aux : T • self.getX(aux);
              le := exp_1[aux]end, self.setX(exp_2)])
  new ≙ b := new B();
  mts_a[var aux : T • self.getX(aux);
        le := exp_1[aux]end, self.setX(exp_2)]
end
class B
  pri x : T;
  meth getX ≙ (res arg : T • arg := self.x)
  meth setX ≙ (val arg : T • self.x := arg)
end
```

Fig. 13. The final classes A and B

can be transformed into a call to method *getX* of class B, as we have already discussed. The resulting call $b.getX(aux)$ is the command present in the method *getX* of class A. Using the same step for introducing method call, we can introduce a call to *getX* of class A having **self** as target. The resulting program is the following.

$$\textbf{var}\ aux : T \bullet \textbf{self}.getX(aux);\ le := exp_1[aux]\ \textbf{end}$$

Similar steps refine assignments $b.x := exp_2$ to $\textbf{self}.setX(exp_2)$. At this point, we do not use the attribute x declared in A, this attribute can be removed using the law $\langle$*introduce private attribute*$\rangle$ in Figure 3, from right to left. The public attribute x of class B can hidden by using a specific law for this [3]. The resulting classes are presented in the Figure 13.

The rule $\langle$*Extract Class*$\rangle$ is applied by means of very small transformation steps in [9], a case study on integration of object-oriented programming languages and relational databases.

5 Conclusions

This paper has presented some refactoring rules for object-oriented programs and illustrated their correctness by deriving one of the rules. This derivation is based on a set of algebraic laws [3,4] for our language. The proposed strategy involves classical data refinement [17] and algorithmic refinement with new features, such as the introduction of classes and method calls. New laws have been proposed for rool as a consequence of this effort.

In [13], 68 refactorings are presented, organised as 6 groups according to the kind of refactoring. The first group is related to the composition of methods;

the second to moving features between classes; the third to data organisation; the fourth is related to the simplification of conditional expressions; the fifth is intended to make method calls simpler; and the sixth deals with generalisation. Of these, we have captured 42 as transformation rules. The others cannot be captured because rool currently does not support references and static methods.

Our general aim is formalising object-oriented design practices. This is important for the practice of refactoring of object-oriented programs and also for justifying the validity of changes accomplished by the use of design patterns. Notwithstanding the fact that we work with a language with a copy semantics, our experience until now reveals that this is not a hindrance to many refactorings. A distinguishing feature of our research is the justification of design practices using a simple, uniform, and modular reasoning mechanism: a set of basic algebraic laws of rool.

The literature related to refactoring of object-oriented programs includes work such as that of Opdyke [18], which proposes a set of seven properties that must be satisfied in order to guarantee behaviour preservation. However, there is no proof in that work that satisfying these properties preserves program semantics. Our approach to refactoring is based on rules. Each rule establishes the restrictions that must be satisfied allowing its application. The application of a rule modifies a program guaranteedly leaving its behaviour unchanged, since each rule is proved from more basic laws or against a weakest precondition semantics of rool [11] and a refinement relation defined in [6,7,5].

Tokuda [20,21] uses the properties proposed by Opdyke for behaviour preservation. The insufficiency of Opdyke's properties for preserving behaviour is recognised, and enabling conditions are added to guarantee that these properties are satisfied. Tokuda also takes the position that refactorings are behaviour-preserving due to good engineering and not to any mathematical guarantee. He argues that given a mature refactoring implementation, refactorings should be treated as trusted tools in the same way as compilers transform source code to assembly even without mathematical proof to guarantee correctness. In our project, we are also concerned with a mathematical proof of compiler correctness [12].

Roberts [19] gives a definition of refactoring that focuses on precondition and postconditions of the refactorings. He also examines techniques for using runtime analysis to assist refactoring. He takes the position that a refactoring is correct if a program that meets its specification continues to meet its specification after the refactoring. A suite of tests is understood as a form of specification. The definition of correctness is based on test suites. A refactoring is correct if a program that passes a test suite continues to pass the test suite after the refactoring. There is no semantic-based proof that refactoring preserves the behaviour of a program or continues meeting its specification.

Fowler [13] suggests that before starting refactoring one should have a solid suite of tests that must be self-checking. Every change must be followed

by program compilation and test. There are no conditions to be satisfied in order to guarantee behaviour preservation. The use of algebraic rules for refactoring eliminates the need of compiling the program as the result of a law application is correct by construction, both from the syntactic and from the semantic points of view. The use of a suite of tests is optional.

Back [2] studies a method for software construction that is based on incrementally extending the system with a new feature at a time. A layered software architecture is proposed to support this method. He also takes into account correctness conditions and reason about their satisfaction in the refinement calculus. Although the approach seems similar to ours, no concrete laws or case studies have been presented in [2].

The proof of the validity of some rules is based on the application of others. More elaborate refactorings than those presented here in fact constitute case studies. Refactoring a poorly structured program into a program structured according to a layered architecture is an already developed case study [9]. Using refactoring rules like those we present here to develop well-established design patterns is the next activity in which we are going to be involved. We have already started on this and considered 4 of the design patterns in [14]. Adapting this approach to deal with pointers is also a topic for further research. rool is already being extended in this direction.

References

[1] Back, R. J. R., "Procedural Abstraction in the Refinement Calculus," Ser. A No. 55. Department of computer Science, Åbo - Finland, 1987.

[2] Back, R. J. R., *Software Construction by Stepwise Feature Introduction*, ZB 2002: Formal Specification and Development in Z and B, Didier Bert *et. al.*, Lecture Notes in Computer Science **2272** (2002), 162-183.

[3] Borba, P., and A. Sampaio, *Basic Laws of ROOL: an Object-Oriented Language*, Revista de INFORMÁTICA TEÓRICA e APLICADA, Instituto de Informática-UFRGS, **7** (2000) 1:49-68.

[4] Borba, P., and A. Sampaio, "The basic laws of ROOL," Technical report, Centre of Informatics-UFPE, 2000, URL: `http://www.cin.ufpe.br/~lmf/coop/papers`.

[5] Cavalcanti, A. L. C., and D. Naumann, *A weakest precondition semantics for an object-oriented language of refinement*, FM'99 - Formal Methods, Lecture Notes in Computer Science **1709** (1999), 1439-1459.

[6] Cavalcanti, A. L. C., and D. A. Naumann, *A Weakest Precondition Semantics for Refinement of Object-oriented Programs*, IEEE Transactions on Software Engineering **8** (2000), 713-728.

[7] Cavalcanti, A. L. C., and D. Naumann, "A Weakest Precondition Semantics for an Object-oriented Language of Refinement - Extended Version," CS 9903, Stevens Institute of Technology, 1999.

[8] 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), 1:87-96.

[9] Cornélio, M., A. Cavalcanti, and A. Sampaio, "Program Development in ROOL," Technical report, Centre of Informatics - UFPE, 2001, URL: `http://www.cin.ufpe.br/~mlc2/papers/progDevelopROOL.ps`.

[10] Cornélio, M., A. Cavalcanti, and A. Sampaio, "Refactoring Rules in ROOL," Technical report, Centre of Informatics - UFPE, 2002, URL: `http://www.cin.ufpe.br/~mlc2/papers/refactRulesROOL.ps`.

[11] Cornélio, M., A. Cavalcanti, A. Sampaio, and P. Borba, "Proving the basic laws of ROOL in a weakest precondition semantics," Technical report, Centre of Informatics - UFPE, 2000, URL: `http://www.cin.ufpe.br/~lmf/coop/papers`.

[12] Duran, A., A. C. A. Sampaio, and A. L. C. Cavalcanti, "Formal Bytecode Generation for a ROOL Virtual Machine," 4th Brazilian Workshop on Formal Methods, 2001.

[13] Fowler, M., "Refactoring: Improving the Design of Existing Code," Addison-Wesley, 1999.

[14] Gamma, E. *et al.*, "Design Patterns: elements of reusable object-oriented software," Addison-Wesley Professional Computing Series, Addison-Wesley, 1994.

[15] Hoare, C.A.R. *et al.*, *Laws of programming*, Communications of the ACM **30** (1987), 8:672-686.

[16] Meyer, B., "Object-Oriented Software Construction," 2nd Ed., Prentide-Hall, 1997.

[17] Morgan, C. C., "Programming from Specifications," 2nd Ed., Prentice Hall, 1994.

[18] Opdyke, W., "Refactoring Object-Oriented Frameworks," Ph.D. thesis, University of Illinois at Urbana-Champaign, 1992.

[19] Roberts, D. B., "Practical Analysis for Refactoring," Ph.D. thesis, University of Illinois an Urbana-Champaign, 1999.

[20] Tokuda, L. A., "Evolving Object-Oriented Designs with Refactoring," Ph.D. thesis, The Department of Computer Sciences, The University of Texas at Austin, 1999.

[21] Tokuda, L., and D. Batory, *Evolving Object-Oriented Designs with Refactoring*, Journal of Automated Software Engineering, **8** (2001), 89-120.