Automation of a Normal Form Reduction Strategy for Object-Oriented Programming

Bruno Lira, Ana Cavalcanti, and Augusto Sampaio

Centro de Informática - Universidade Federal de Pernambuco
Recife - PE, Brasil
{bol, alcc, acas}@cin.ufpe.br
http://www.cin.ufpe.br

Abstract. The work presented in this paper builds on previous results which define a normal form reduction strategy for *ROOL*, an object-oriented language similar to sequential Java. The purpose of this reduction process is to establish relative completeness of a set of algebraic laws for the object-oriented features of *ROOL*. The purpose of the present work is to show how each algebraic law can be encoded in the Maude term rewrite system, and how the reduction engine of Maude can be used as an automatic tool to mechanize the reduction process.

1 Introduction

There are several approaches for defining the semantics of a programming language: operational, denotational, algebraic, etc. In the algebraic approach, the semantics is presented as a set of laws which relate the programming constructs.

The algebraic style has several distinguishing features:

1. As algebraic laws are directly expressed in programming terms, they can be promptly used for program transformation.
2. No explicit model is built in a algebraic style, unlike in operational or denotational semantics. Therefore, extending an algebraic semantics tends to be more modular.
3. Algebraic laws are easily mechanisable as rewrite rules in a term rewrite system like *OBJ3* [10] or *Maude* [8].

These potential advantages have motivated several efforts concerning the definition of algebraic semantics of programming languages. Concerning imperative programming, there is the classical work of Hoare et al [12]. For concurrent programming, we single out the laws of *occam* [16], as proposed by Roscoe and Hoare.

Only recently, laws of object-oriented programming have been given the deserved attention. In [3], Borba and Sampaio propose a comprehensive set of laws for the object-oriented features of ROOL, a language similar to sequential Java, but with a copy rather than a reference semantics [5, 6].

Although the algebraic approach has the potential advantages mentioned above, it also presents some inherent difficulties. One is related to the completeness of the proposed algebraic laws; another concerns their consistency.

Completeness is usually addressed by showing that the laws are sufficient to reduce an arbitrary program to a normal form expressed in terms of a small subset of the language operators. This has been adopted in [12, 16, 4]. Consistency is proven by appealing to an another independent semantics, usually in the denotational or operational style.

In this paper, we show how the laws proposed in [4] can be encoded as rewrite rules in Maude. We then use Maude's rewrite engine to show how an arbitrary program can be automatically reduced to a normal form. This serves as an operational proof of the relative completeness of the proposed set of rules and as a case study of Maude as the backend for a more complex tool to suport advanced object-oriented programming techniques. It also serves as a transformation tool, in that the rules can be used for program transformation.

Existing works to validate a Java semantics using a theorem prover and other tools [15, 18] are based on an operational semantics. Validation is obtained through interpretation. Our aim in this work is different: reduction of programs to a normal form.

2 Rool

ROOL is an object-oriented language based on Java, but with a copy semantics rather than a reference semantics. It has been specially designed to allow reasoning about object-oriented programs and specifications, based on Morgans's refinement calculus [14]. The language was fully described in [5] and [3]. We give here just a general view of the language.

A program in *ROOL* is written as $cds \bullet c$, for a main command c and a set of class declarations *cds*. Classes are declared according to the following scheme:

$class\ C_1\ [\ extends\ C_2\ ]$
$\quad \{pri\ x_1 : T_1\ ;\}^*\ \{prot\ x_2 : T_2\ ;\}^*\ \{pub\ x_3 : T_3\ ;\}^*$
$\quad meth\ m = (\ pds\ .\ c\ )\ end;$
$end.$

where C_n are classes' names; x_n are valid names for identifiers, and T_n are data types of *ROOL* (basic types, such as integer or boolean, or class name).

Subclassing and single inheritance are supported by the *extends* clause, which is optional. Its absence means that the class is a direct subclass of *Object*, which is a superclass of all classes in *ROOL*. There are three visibility qualifiers for attributes in our language: *pri*, *prot* and *pub* for the private, protected and public atributes, respectively, with similar semantics to Java. For simplicity, we consider only public methods,which can have value, result, and value-result parameters. The list of parameters are separated from its body by the symbol $\bullet$.

In addition to method calls, the body of methods may have imperative constructors similar to the ones adopted by Morgan [14].

$c \in Com ::= \mid le := e \mid c \; ; \; c$	multiple assigment, sequence
$\mid pc(\; e\;)$	parameterized command application
$\mid if \; []_i \bullet \Psi_i \rightarrow c_i \; fi$	alternation, with condition Ψ_i
$\mid rec \; X \bullet c \; end \mid X$	recursion, recursive call
$\mid var \; x : T \bullet c \; end$	local variable block
$\mid avar \; x : T \bullet c \; end$	angelic variable block

Methods are seen as parameterized commands, which can be applied to a list of arguments to yield a command. Therefore, method calls are represented as the application of parameterized programs.

A parameterized command is composed by a parameter declaration followed by a command or a reference to a method. This can be a method m of the current class or a call $le.m$ with target object le.

$pc \in PCom ::= pds \bullet c$	parametrization
$\mid m \mid le.m \mid$	method calls
$pds \in Pds ::= \; \emptyset \mid pd \mid pd \; ; \; pds \mid$	parameter declarations
$pd \in Pd ::= \; val \; x : T \mid res \; x : T \mid vres \; x : T \mid$	

Methods in $ROOL$ cannot be mutually recursive, but classes can.

For expressions, $ROOL$ supports typical object-oriented constructors:

$e \in Exp ::= self \mid super$	special 'references'
$\mid null \mid new \; N$	null 'reference', object creation
$\mid x \mid f(\; e\;)$	variable, built-in application
$\mid e \; is \; N \mid (\; N\;) \; e$	type test, type cast
$\mid e.x \mid (\; e \; ; \; x : e\;)$	attribute selection and update

where $self$ and *super* has the same semantics to *this* and *super* in Java, respectively, and the update ($e_1 \; ; \; x : e_2$) denotes a copy of the object denoted by e_1, but with the attribute x mapped to a copy of e_2.

We identify a subset of the expressions, named Le, that can appear at the left side of an assignment or as a target of a method call: any variable (x), the access to an object's attribute ($le.x$), a type cast ($(N) \; le$) and $self$.

An expression is simple if it has one of the following forms, where *bop* and *uop* stand for binary and unary operators, respectively.

$$
\begin{array}{lll}
SimpleExp ::= & \mid self \mid null & \mid new \; N \\
 & \mid x_1 \; bop \; x_2 & \mid uop \; x \\
 & \mid x & \mid x \; is \; N \\
 & \mid (\; N\;) \; x & \mid x_1.y_2 \\
 & \mid (x_1 \; ; \; x_2 : x_3\;) &
\end{array}
$$

This notion is used in the normal form definition.

3 A Normal Form

The main objective of our work is to automatize the process of reducing a *ROOL* program to a normal form [4], therefore, demonstrating the completeness of the algebraic rules of our language. So, if we want to reach our goal, we must use a suitable normal form, which we call Subtype Normal Form.

By definition, a *ROOL* program $cds \bullet c$ is in Subtype Normal Form if it obeys the following conditions:

1. Each class declaration in *cds* includes only the clause **extends**; no explicit declaration of methods (including constructors) or attributes are allowed. The only exception is *Object* which may include declaration of attributes (each with either a primitive type or the type *Object* itself).
2. All the local variables in the main command c are also declared to have a primitive or *Object* type.
3. No type cast is allowed in c.

A program in this normal form preserves some object-oriented features, such as subtype hierarchy, creation of objects, and type tests. In spite of this, it is very close to an imperative program as in the work of Hoare [12]. *Object* takes the form of a recursive record, and the main command is practically an imperative program with the added possibility of creating objects and making type tests.

We could advance more in the reduction of *ROOL* programs, defining a normal form which would be completely imperative, eliminating all the object-oriented features. For this, we need some sort of encoding in the style of a mapping from an object to a relational model, as an extra variable (attribute or field) would be necessary to record the type information. Such advance would bring a considerable increase in its complexity to the reduction strategy, and so, is left as a topic for further research.

The reduction process involves the following major steps. First move all code (attributes and methods) in *cds* to the *Object* class. Afterwards, change all the declarations of object identifiers to type *Object* and eliminate casts. Finally, eliminate method calls and declarations.

The application of this strategy to a *ROOL* program ($cds \bullet c$) converges to the normal form if the program satisfies the following assumptions: *cds* has no name clashing for attributes; there is no name clashing for methods in *cds*, except in the case of redefinitions; all references to an attribute a are of the form $self.a$; recursion in method declarations is expressed in terms of the *rec* operator; and, finally, the class *Object* is explicitly declared by the program in reduction. None of these assumptions diminish the expression power of the language, and can be satisfied by simple syntactical transformations.

The first two conditions are necessary to ensure that the laws to move attributes and methods up in the hierarchy are always applicable. The third condition allows a uniform treatment of attribute occurrences. Finally, the fourth allows that the elimination of method calls to be achieved by a law similar to the copy rule, since, strictly, the methods are not recursive; although their body might be a recursive command described using *rec*.

The above strategy is based on algebraic laws. In the next section, we will show the general steps that compose our reduction process.

4 The Reduction Strategy

All rules necessary for the reduction strategy are fully described in [4], where we can find conditions for their left-to-right and right-to-left use, as well as the proof that any *ROOL* program, satisfying the conditions stated in Section 3, can be reduced to the normal form.

In this subsection, we are going to give a simple example of how the strategy can be used to simplify a *ROOL* program. So we are only going to consider the application of the rules in the direction used in our tool. With that simplification, we aim at facilitating their understanding.

We are going to use the following example program (we name it *PRG*):

```
class Object end
class Account
    pri number : String ; prot saldo : double ;
    meth credit = val  valor : double  .
            self.saldo := self.saldo + valor end
    meth withdraw = val  valor : double  .
            self.saldo := self.saldo - valor end
    meth Const = val n  : String .
            self.number := n; self.saldo := 0 end
end
class Special_Account extends Account
    pri  points : double ;
    meth credit = val  valor : double  .
        self.points := self.points + 1; super.credit(valor) end
end
. var conta1 : Account .
    conta1 := new Account ("21.342-7");
    conta1.credit (500.00); conta1.withdraw (45.00);
end
```

This represents a simple bank account with the operations of credit and debit and a special account that gives one point for each deposit made in the account.

Make Attributes Public The first step of our strategy is to change the visibility of the attributes of the classes from protected and private to public ones. This is specified by the following algebraic rule which used the notation $cds_1 =_{cds,c} cds_2$ corresponding to the equivalence of sets of class declarations cds_1 and cds_2, where cds is the context 'auxiliary' declarations for cds_1 and cds_2, and c is the main program.

Law <change visibility of the attributes>

`class C extends D`
prot a **:** T**;**
pri a_1 **:** T_1**;**
pub a_2 **:** T_2**;** `ads`
`ops`
`end`
	$=_{cds,c}$	`class C extends D`
		pub a **: T;**
		pub a_1 **:** T_1**;**
		pub a_2 **:** T_2**;** `ads`
		`ops`
		`end`

So, the classes of our simple program could be rewritten to:

```
class Object end
class Account
    pub number : String ; saldo : double ;
    ...
end
class Special_Account extends Account
    pub points : double ;
    ...
end
```

Move Attributes Up After changing the visibility of the attributes to public, we can easily move them up through the class hierarchy tree until *Object*.

```
class Object
    pub number : String ; pub saldo, points : double ;
end
class Account
    meth credit = val  valor : double  .
            self.saldo := self.saldo + valor end
    ...
end class Special_Account extends Account
    meth credit = val  valor : double  .
       self.points := self.points + 1; super.credit(valor) end
end
```

Eliminating Super To move safely the methods of the classes to *Object* we should make sure that their bodies do not contain ***super***. To eliminate such calls, we can replace them by a copy of the method's body declared in the superclass, unless it contains ***super*** or any private attributes.

However, notice that a method called via ***super*** is not always declared in the immediate superclass of the class where the call appears. In order to cope with this problem, we must first complete the declarations of the classes with trivial methods redefinitions. `Special_Account` would be rewritten to the following class after the completing process:

```
class Special_Account extends Account
    meth credit = val valor : double  .
       self.points := self.points + 1; super.credit(valor) end
    meth withdraw = val valor : double  .
            super.withdraw(valor) end
    meth Const = val n  : String .
            super.Const(n) end
end
```

Finally, eliminating the method calls via ***super*** in the same class:

```
class Special_Account extends Account
    meth credit = val valor : double  .
            self.points := self.points + 1;
            self.saldo := self.saldo + valor end
    meth withdraw = val valor : double  .
            self.saldo := self.saldo - valor end
    meth Const = val n  : String .
            self.number := n; self.saldo := 0 end
end
```

Move Methods Up Now we can move up the methods in the same way that we have made with the attributes. In case of a redefinition, we merge the two method declarations into a single one in the superclass, and if it is an original method, we can safely move it up. *Object* is rewritten to:

```
class Object
    pub number : String ; saldo, points : double ;
    meth credit = val  valor : double  .
        if (self is Account) and not(self is Special_Account) ->
              self.saldo := self.saldo + valor
        [] (self is Special_Account) ->
              self.points := self.points + 1;
              self.saldo := self.saldo + valor
        fi end
    meth withdraw = val  valor : double  .
         self .saldo := self.saldo - valor end
    meth Const = val n  : String .
         self.number := n; self.saldo := 0 end
end
```

Cast Introduction Before we change the types of the identifiers to *Object*, we must introduce trivial casts in the expressions. The reason for this is to avoid the introduction of compilations errors in a valid *ROOL* program. For example, let a be a variable of the type C, and x a method only defined in the same class C. If we change the type of a to *Object*, all method calls $a.x$ will lead to a compilation error introduced by the strategy. The solution to this problem is the introduction of trivial casts in all expressions, for example, *(C) a.x*. So, in our example program, *Object* and the main command will be rewritten to:

```
class Object
    pub number : String ; saldo, points : double ;
    meth credit = val  valor : double  .
        if (self is Account) and not(self is Special_Account) ->
              self.saldo := ((Object) self).saldo + valor
        [] (self is Special_Account) ->
              self.points := ((Object) self).points + 1;
              self.saldo :=  ((Object) self).saldo + valor
        fi end
    meth withdraw = val  valor : double .
        self.saldo := ((Object) self).saldo - valor end
    meth Const = val n  : String .
        self.number := (String) n;
        self.saldo := 0 end
end
. var conta1 : Account .
    conta1 := new Account ("21.342-7");
    ((Account) conta1).credit (500.00);
    ((Account) conta1).withdraw (45.00);
end
```

Change Type to *Object* As all expressions have a cast, introduced by the user or by the strategy itself, we can change the types of the identifiers to *Object*.

```
class Object
    pub number : String ; saldo, points : double ;
    ...
    meth Const = val n  : Object .
       self.number := (String) n; self.saldo := 0 end
end
.var conta1 : Object .
    conta1 := new Account ("21.342-7");
    ...
end
```

Cast Elimination We can, at this moment, start the elimination of all casts of our example program. For such, we rewrite casts as assertion commands, which can be defined as conditions that is checked at a given point in the program text. When an assertion $\{b\}$ is executed, if b holds, nothing happens; otherwise the program aborts.

Such rewrite is necessary because of the dynamic behaviour of the cast, which works as *abort* when the expression under its guard does not have the specified type, and *skip* otherwise.

```
class Object
    pub number : String ; pub saldo, points : double ;
    meth credit = val  valor : double  .
        if (self is Account) and not(self is Special_Account) ->
                {self is Object};
                self.saldo := self.saldo + valor
        [] (self is Special_Account) ->
                {self is Object};
                 self.points := self.points + 1;
                 self.saldo := self.saldo + valor
        fi end
    meth withdraw = val  valor : double  .
                 {self is Object};
                 self.saldo := self.saldo - valor end
    meth Const = val n  : Object .
                {self is Object}; {n is String};
                self.number := n; self.saldo := 0 end
end
.var conta1 : Object .
    conta1 := new Account ("21.342-7");
    { conta1 is Account };
    conta1.credit (500.00);
    { conta1 is Account };
    conta1.withdraw (45.00);
end
```

Method Elimination The last step of our strategy is to fully eliminate the methods of *Object*. This is done by a copy rule that substitutes all method calls with its respective body, which has already dealt with dynamic bind when we moved the methods to *Object*. Afterwards we can eliminate the method definition as it is no long necessary. So, the normal form of our example program PRG is:

```
class Object
    pub number : String ; saldo, points : double ;
end
class Account end
class Special_Account extends Account end
.var conta1 : Object .
    conta1 := new Account ;
    conta1 := (val n  : Object .
               {conta1 is Object}; {n is String};
               conta1.number := n; conta1.saldo := 0 )("21.342-7");
    {conta1 is Account };
    (val  valor : double ; .
      if (conta1 is Account) and not(conta1 is Special_Account) ->
         {conta1 is Object}; conta1.saldo := conta1.saldo + valor
      [] (conta1 is Special_Account) ->
         {conta1 is Object}; conta1.points := conta1.points + 1;
         conta1.saldo := conta1.saldo + valor fi) (500.00);
    { conta1 is Account };
    (val valor : double ; . {conta1 is Object};
      conta1.saldo := conta1.saldo - valor) (45.00);
end
```

With the application of these steps, we can simplify any *ROOL* program that satisfies the provisos stated earlier on this section. More details about the reduction strategy can be found in [4].

5 A Reduction Tool

One of the major problems with our strategy is the fact that it can lead to a great number of steps to reduce a single program to normal form. In [13], we have an example where a more complex program is fully reduced according to the strategy. Besides, during the reduction process, a great number of rules can be fired at the same moment, and choosing which one of them must be applied ahead of the others is a crucial task. Therefore, we decided to make an automated tool that releases the users from the tiresome task of making the reduction by hand and optimizes the process as far as possible.

Our first challenge was to decide in which platform to develop our tool. First we selected some criteria: rewriting adaptability, facilities to build an incremental specification, full documentation for the end user, a simple reduction mechanism, and the possibility of defining different reduction strategies. According to the chosen criteria, Maude [8] was selected as our development platform. More details about the choice of the platform can be found in [13].

As mentioned in [8], Maude is a high-performance language and system, supporting both the equational and rewriting logic. It was influenced by OBJ3 [10] in many features and has a much better performance; it also supports a richer equational logic, namely, membership equational logic, that extends OBJ3's order-sorted equational logic.

In Maude, we can build two kinds of modules: the functional modules, which define data types and functions on them, by means of the membership equational logic whose equations are Church-Rosser and terminating, and the system modules, which are based on the rewriting rules. The rewrite rules need not be terminating, and need not be Church-Rosser. In our tool, we use both of them.

Defined the platform, we started to tackle our main objective. We initiated by constructing the specification of the abstract syntax of *ROOL*. For that we implemented seven functional modules which are shown in Fig. 1.

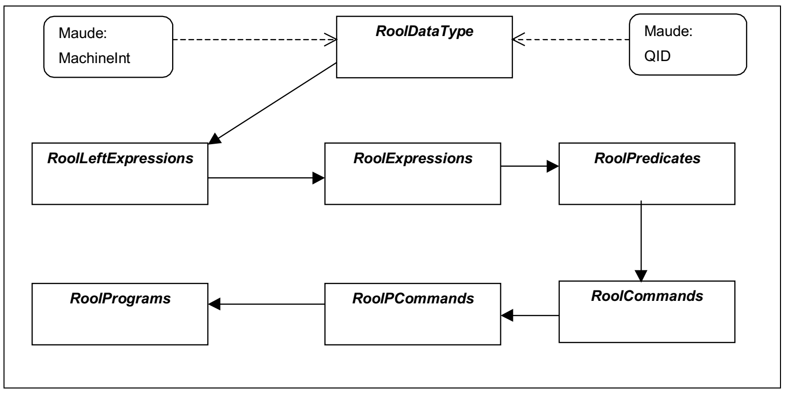

Fig. 1. Modules that implement the specification of *ROOL* in Maude.

ROOLDATATYPE specifies the data types of the language: the basic ones and the names of the classes. In our tool, all class names, with the exception of *Object*, is preceded by the prefix *CLID*. So *CLID Account* is a valid class identifier. Such abstraction was used so that we could distinguish the function of the identifier by the prefix. See table 1 for the prefixes used in our tool.

Table 1. Prefix used in our tool for distinguishing the function of the identifiers

Constructor	Used Prefix	Example
Attribute, parameter and variables	-	‘a , ‘b, ‘temp_account
Class Name	CLID	CLID ‘Account, CLID ‘Savings
Method Name	MtID	MtID ‘debit MtID ‘credit
Command Name	CID	CID ‘factorial, CID ‘Fib

ROOLLEFTEXPRESSIONS represents the left expressions of *ROOL*: the subset of the expressions which can appear at the left side of an assignment and as target of method calls. In addition, *ROOLEXPRESSIONS* and *ROOLPREDICATES* specify the others expressions and the predicates of *ROOL*, respectively. The predicates of our language can be any formula of the predicate calculus.

The imperative constructors of *ROOL* are defined in the modules *ROOLCOMMANDS* and *ROOLPCOMMANDS*. The last one builds the parameterized commands. The module *ROOLPROGRAMS* specifies the object-oriented class constructors of *ROOL*, as well as the definition of a program in our language.

With these seven modules, we can describe any *ROOL* program in our tool. For example, the program used in Section 4 can be expressed in our tool as the following specification:

```
class Object nullAtribute  nullMethod end
class CLID 'Account
    pri 'number : str :: saldo : double;
    meth MtID 'Const ^= val 'n  : String .#
        self.'numero := 'n ; self.'saldo := 0 # end
    meth MtID 'credit ^= val 'valor : double ; .#
        self.'saldo := Sum < self.'saldo :: 'valor > # end
    meth MtID 'withdraw ^= val 'valor : double ; .#
        self.'saldo := Sum < self.'saldo :: Minus < 'valor > > #
    end
end
class CLID 'Special_Account extends CLID 'Account
    pri 'points : double ;
    meth MtID 'credit ^= val 'valor : double ; .#
        self.'points := Sum< self.'poits :: 1 > ;
        super.MtID 'credit < 'valor > end
end
.main< var 'conta1 :: Lnull : CLID 'Account .
    'conta1 := new CLID 'Account ;
    'conta1 . MtID 'Const < "21.342-7" > ;
    'conta1 . MtID 'credit < 500.00 > ;
    'conta1 . MtID 'withdraw < 45.00 > ;
end
>
```

There are few differences between the actual *ROOL* concrete syntax and and our abstract syntax, specified in Maude. The absence of attributes or methods must be explicitly declared by the keywords *nullAtribute* and *nullMethod*. The main command is separated from the class declarations by *.main<...>*, as in C. We made this change to improve the visualization of the main command. Other crucial matter is the fact that the tool does not give any special treatment for the constructor of a class. It is considered a mere method as the others, and it is up to the user to call it at the appropriate time.

As for the rules, we specified them in one system module - *ROOLRULES*. The main guideline in their implementation was full automation. So, we control the rules' order of fire by functions on their conditions. As an example, the first step of the strategy, which specifies the change in visibility of the attributes, is implemented as follows:

ceq `class B attr1 meth1 end` =
    `class B` (*TransPub*(`attr1`)) `meth1 end`
    `if not` *allPub*(`attr1`) .

```
ceq class B extends C attr1 meth1 end =
    class B extends C ( TransPub( attr1 ) ) meth1 end
    if not allPub( attr1 ) .
```

The function *TransPub* is responsible for the transformation of the private and protected attributes to public ones, while *allPub* returns true if all attributes are public, and false otherwise.

Specifically, we implement this step as equations and not as a rewrite rules. This is due to the greater efficiency, in Maude, of the reduction mechanism. As for the order of firing, Maude guarantees that the equations are evaluated before the rewriting rules, which gives us the right order of firing.

Not all rules, however, can be specified in this style. Some of them have to be written as pure functions, due to the great amount of instances that we would have to write to guarantee the full automation. For example, as expressions can appear everywhere in a *ROOL* program, we would have to write a countless number of instances of rules to guarantee our main guideline in the step of introducing trivial casts. Instead, we specified this rule in the form presented bellow. In [13], we give a detailed explanation of the explosion of rules.

```
crl [law-simplify-exp] : cds1 .main< c > =>
                         simplifyExpPRG( ( cds1 .main< c > ) )

if not isAllSimpleExp( ( cds1 .main< c > ) )
and noMethod( cds1 ) .
```

The function *isAllSimpleExp* receives a program and checks if all expressions in it are simple; *noMethod* verifies if all code in the classes (attributes and methods) are only defined in *Object*. So they represent the stop point and the precondition of the rule.

The function *simplifyExpPRG* receives a *ROOL* program and reduces its complex expressions to a simple form. Below we can see how *simplifyExpCOM*, which is analogous to *simplifyExpPRG*, deals with the simplification of a method call:

```
ceq simplifyExpCOM( le . x < expL > , prog ) = le . x < expL >
if isSimple( expL ) and isSimple( le ) .

ceq simplifyExpCOM( le . x < expL > , prog ) =
           var createVars( le , prog , 1 ) .
               createAtribs( le , prog , 1 ) ;
               createListVars( le , prog , 1 ) . x < expL >
           end
if not isSimle( le ) .

ceq simplifyExpCOM( le . x < expL > , prog ) =
           var createVars( expL , prog , 1 ) .
```

```
                createAtribs( expL , prog , 1 ) ;
                le . x < createListVars( expL , prog , 1 ) >
            end
if not isSimple( expL ) and isSimple( le ) .
```

If the left expression `le` and the expression list `expL`, which represents the parameters are already simplified, the function *simplifyExpCOM* returns them without changes. Otherwise, it creates a fresh variable using the *createVars* auxiliary function, assigns to it the value of `le` using *createAtribs* and rewrites the method call, changing the left expression `le` to the variable created before (function *createListVars*). The same procedure is applied in the simplification of the others expressions. With only this rule, we can simplify all expressions in a *ROOL* program. The complete specification of our tool can be found in [13].

Another key issue in our implementation is the optimization of the process. During our design, we developed a "Method Jump" and the local completion mechanism. These optimizations are used during the step of eliminating method calls with target ***super***. As described earlier, before we can safely eliminate this kind of call, we have to complete all class definitions with trivial method redefinitions using ***super*** itself. This is specified as the following rule:

Law <introduce method redefinition>

```
class B extends A                     class B extends A
  ads                                   ads
  meth m = pc end ; ops                 meth m = pc end ; ops
 end                             =     end
class C extends B                     class C extends B
  ads'                                  ads'
  ops'                                  meth m = super.m end; ops'
 end                                   end
```

***Provided* -** `m` is not declared in `ops'`.

The above rule says that we can introduce, in the subclass, trivial redefinitions of parents methods via ***super*** provided it is not already redefined. The result of the exhaustive application of this rule in a *ROOL* program results in classes that have redefinitions for all parents methods.

With this completion, we can just apply a copy rule that replaces a method call using ***super*** by a copy of the method's body declared in the superclass, provided the body does not contain ***super*** nor private attributes, which would not be visible in the subclass.

Law <eliminate super>

Consider that CDS is a set of two class declarations such as the following:

```
class B extends A
   ads
   meth m = pc end; ops
 end
class C extends D
   ads'
   ops'
 end
```

If ***super*** and the private attributes in `ads` do not appear in `pc`, we have

cds CDS , *C* $\triangle$ **super** . `m = pc`

The notation *cds CDS* , *C* $\triangle$ ***super***.*m = pc* indicates that **super** . `m = pc` holds inside class *C*, in a context defined by the set of class declarations *cds CDS*.

Our optimization is that, instead of inserting redefinitions from *Object* until the classes which are the leaves of the hierarchy tree, we just get the body of the method in the nearest superclass that has a suitable definition for it. In this way, we eliminate a series of definitions that would never be used.

Another problem arises when we use such optimization: in the body of the method we can have method calls of the kind ***self.m***, where *m* is defined only in the superclass. The solution to this is the local completion method. We complete classes with trivial redefinitions of methods if it already has a method call with **super**. With these optimizations, the method jump and the local completion method, we improved the efficiency of our tool in 15

The complete tool is implemented with a total of 70 rules and is fully automated. In [13] one can find not only the complete specification, but a case study, where we demonstrate its convergency, and a the reduction of a more complex program being reduced by it.

6 Conclusions and Future Work

Our main contribution is an operational proof that the reduction strategy specified in [4] is feasible for automation and the defined set of rules is relatively complete. Our tool can, in fact, reduce *ROOL* programs that satisfy the provisos mentioned in Section 3 of this paper.

The use of Maude has shown to be invaluable for our work. We had no great problems to give an abstract syntax specification for *ROOL* as well as to specify and use the laws for reducing programs. In fact, all laws were literaly translated to rewrite rules in the system module of Maude and we used the default rewrite/reduce engine of the system. This work is also a contribution to the Maude community as it proves a good case study of program transformation in the refered theorem prover.

Although our implementation is a specific tool, it will serve as the basis for the implementation of a more general one that will help in the justification of widespread object-oriented programming techniques. In that new tool, we will use the laws for other tasks of program transformation such as refactoring and compilation [7].

Another topic for the future is the extension of the reduction strategy for the imperative subset of *ROOL*. We will also consider the proposition of rules for a language based on a reference semantics instead of a copy one.

References

1. Borba, P.: *Where are the laws of object-oriented programming?* In: I Brasilian Workshop on Formal Methods, pages 59-70, Porto Alegre Brasil (1998)
2. Borba, P. and Sampaio, A.: *Regras de Comando de ROOL* (draft)
3. Borba, P. and Sampaio, A.: *Basic Laws of ROOL: an Object-Oriented Language.* In: 3rd Brazilian Workshop of Formal Methods. (2000) 33-44
4. Borba, P. and Sampaio, A.: *A Normal Form Reduction Strategy for ROOL: an Object-Oriented Language.* In: Technical Report, Centro de Informática, UFPE, (2002)
5. Cavalcanti, A.L.C. and Naumann, D.: *A Weakest Precondition Semantics for an Object-oriented Language of Refinement.* In: Jeannette M. Wing, Jim Woodcock and Jim Davies, editors, FM'99 - Formal Methods, LNCS, 1709:1439-1459. Springer-Verlag, (1999)
6. Cavalcanti, A.L.C. and Naumann, D.: *A Weakest Precondition Semantics for refinement of object-oriented programs.* In: IEEE Transactions on Software Engineering, 26(8): 713 - 728, (2000)
7. Duran, Adolfo and Sampaio, Augusto and Cavalcanti, Ana.: *Formal Bytecode Generation for a ROOL Virtual Machine.* In: 4th Brazilian Workshop of Formal Methods, 2001. Revista de Informtica Terica e Aplicada, 7(1): 49-68, 2000.
8. Clavel, M. and Durán, F. *et al.*: *Maude: Specification and Programming in Rewriting Logic.* In: Computer Science Laboratory - SRI International, (1999)
9. Conelio, M. and Cavalcanti, A.L.C.: *Proving the basic laws of ROOL in a weakest precondition semantics.* In: Technical Report, Centro de Informática, UFPE, 2000
10. Goguen, J. Winkler, T. *et al.*: *Introducing OBJ.* In: Technical Report SRI-CSL-92-03, SRI International, Computer Science Laboratory, (1992)
11. Gosling, J.; Joy, B. and Steele, G.: *The Java Language Specification.* In: Addison-Wesley (1996)
12. Hoare, C. A. R. *et al.*: *Laws of Programming.* In: Communications of the ACM, 30 (8): 672-686, (1987)
13. Lira, Bruno: *Automatização de uma estratégia de redução a forma normal para programas orientados a objetos.* In: MSc Thesis - to appear.
14. Morgan, C.: *Programming from Specifications.* In: Prentice Hall, second edition (1994)
15. Oheimb,D. and Nipkow, T.: *Machine-checking the Java specification: Proving Type-Safety.* Chapter of Springer LNCS Vol. 1523: Formal Syntax and Semantics of Java, 1999
16. Roscoe, A. and Hoare, C.A.R.: *The laws of occam programming.* In: Theoretical Computer Science, 60:177-229 (1988)
17. Sampaio, A.: *An Algebraic Approach to Compiler Design Volume 4 of Algebraic Methodology and Software Technology.* World Scientific, (1997)
18. Syme, D.: *Proving Java Type Soundness.* University of Cambridge Computer Laboratory Technical Report 427, (1997)