Journal of Universal Computer Science, vol. 13, no. 6 (2007), 721-736
submitted: 19/1/07, accepted: 22/3/07, appeared: 28/6/07 © *J.UCS*

Logic Programming for Verification of Object-Oriented Programming Law Conditions

Leandro de Freitas, Marcel Caraciolo, Márcio Cornélio
(Departamento de Sistemas Computacionais
Escola Politécnica — Universidade de Pernambuco (UPE)
Recife, PE, Brazil
{ldf,mpc,mlc}@dsc.upe.br)

Abstract: Programming laws are a means of stating properties of programming constructs and resoning about programs. Also, they can be viewed as a program transformation tool, being useful to restructure object-oriented programs. Usually the application of a programming law is only allowed under the satisfaction of side-conditions. In this work, we present how the conditions associated to object-oriented programming laws are checked by using Prolog. This is a step towards a tool that allows user definable refactorings based on the application of programming laws.
Key Words: refactoring, logic programming, programming law conditions
Category: D.1.5, D.1.6, D.2.6

1 Introduction

Object-oriented programming has been acclaimed as a means to obtain software that is easier to modify than conventional software [Meyer 1997]. Restructuring an object-oriented program is an activity known as refactoring [Opdyke 1992, Fowler 1999], allowing us, for instance, to move attributes and methods between classes or to split a complex class into several ones. These modifications just change the internal software structure without affecting the software behaviour as perceived by users. Work on refactoring usually describes the steps used for program modification in a rather informal way [Fowler 1999, Opdyke 1992].

The refactoring practice usually relies on test and compilation cycles, based on small changes applied to a program. Works in the direction of formalising refactorings deal mainly with the identification of conditions that must be satisfied to guarantee that a change to a program is behaviour preserving [Opdyke 1992, Roberts 1999]. Opdyke [Opdyke 1992] proposes a set of conditions required for application of refactorings, whereas Roberts [Roberts 1999] introduces postconditions for refactorings, allowing the definition of refactoring chains. The possibility of defining conditions that must be satisfied to apply a chain of refactorings is a benefit of the introduction of postconditions.

In our approach, programming laws are the basis for the derivation of refactoring rules, besides laws that lead to data refinement of classes [Cornélio 2004]. These laws precisely indicate the modifications that can be done to a program, with corresponding proof obligations. Using laws, program development is justified and documented. In order to deal with the correctness of refactorings, in our approach the derivation of a refactoring rule is carried out by

the use of laws that are themselves proved [Cornélio 2004] against the semantics [Cavalcanti and Naumann 1999, Cavalcanti and Naumann 2000] of our language. Here we consider sequential programs and we do not take into account programs with space and time issues.

In order to apply an object-oriented programming law it is necessary to check side-conditions to its application or to prove some proof obligations. In this work we present an application of logic programming to verify conditions of programming laws, determining whether a law can be applied to a program or not. Program transformations accomplished by the use of programming laws preserve program behaviour [Cornélio 2004]. The use of logic programming was inspired by the JTransformer framework [Rho et al. 2003], a query and transformation engine for Java source code.

This paper is organised as follows. In Section 2 we present our study language and object-oriented programming laws. In Section 3, we introduce the structure of the program fact databases that we use for representing program syntax trees. In Section 4, we describe how we verify conditions of object-oriented programming laws. We discuss some related work in Section 5. Finally, in Section 6, we present our conclusions and suggestions for future work.

2 The Language and Laws of Programming

The language we use for our study is called ROOL (an acronym for Refinement Object-Oriented Language) and is a subset of sequential Java with classes, inheritance, visibility control for attributes, dynamic binding, and recursion [Cavalcanti and Naumann 2000]. It allows reasoning about object-oriented programs and specifications, as both kinds of constructs are mixed in the style of Morgan's refinement calculus [Morgan 1994]. The semantics of ROOL is based on weakest preconditions [Cavalcanti and Naumann 2000]. The imperative constructs of ROOL are based on the language of Morgan's refinement calculus [Morgan 1994], which is an extension of Dijkstra's language of guarded commands [Dijkstra 1976].

A program $cds \bullet c$ in ROOL is a sequence of classes *cds* followed by a main command c. Classes are declared as in Fig. 1, where we define a class *Account*. Classes are related by single inheritance, which is indicated by the clause **extends**. The class **object** is the default superclass of classes. So, the **extends** clause could have been omitted in the declaration of *Account*. The class *Account* includes a private attribute named *balance*; 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 [Cavalcanti et al. 1999]. A parameterised command can have the form $\mathbf{val}\ x : T \bullet c$ or $\mathbf{res}\ x : T \bullet c$, which correspond to the call-by-value and call-by-result parameter passing mechanisms, respectively. For instance, the method *getBalance* in Fig. 1 has a result parameter r, whereas *setBalance* has a value parameter s. Initialisers are declared by the **new** clause. In the main pro-

```
class Account extends object
  pri balance : int;
  meth getBalance ≙ (res r : int • r := self.balance)
  meth setBalance ≙ (val s : int • self.balance := s)
  new ≙ self.balance := 0
end •
var acct : Account •
  acct := new Account();
  acct.setBalance(10);
end
```

Figure 1: Example of program in ROOL

gram in Fig. 1, we introduce variable *acct* of type *Account* to which we assign an object of such class.

For writing expressions, ROOL provides typical object-oriented constructs (Table 1). We assume that x stands for a variable identifier, and f for a built-in function; **self** and **super** have a similar semantics to `this` and `super` in Java, respectively. The type test e **is** N has the same meaning as in e `instanceof` N in Java: it checks whether non-null e has dynamic type N; when e is **null**, it evaluates to false. The expression $(N)e$ is a type cast; the result of evaluating such an expression is the object denoted by e if it belongs to the class N, otherwise it results in error. Attribute selection $e.x$ results in a run-time error when e denotes **null**. The update expression $(e_1;\ x : e_2)$ denotes a copy of the object denoted by e_1 with the attribute x mapped to a copy of e_2. If e_1 is **null**, the evaluation of $(e_1;\ x : e_2)$ yields **error**. Indeed, the update expression creates a new object rather than updating an existing one.

The expressions that can appear on the left-hand side of assignments, as the target of a method call, and as result arguments constitute a subset *Le* of *Exp*. They are called left-expressions.

$$le \in Le \quad ::= le1 \mid \textbf{self}.le1 \mid ((N)le).le1$$
$$le1 \in Le1 ::= x \mid le1.x$$

The predicates of ROOL (Table 1) include expressions of type **bool**, and formulas of the first-order predicate calculus.

The imperative constructs of ROOL, including the ones related to object-orientation concepts, are specified in the Table 2. In a specification statement $x : [\psi_1, \psi_2]$, x is the frame, and the predicates ψ_1 and ψ_2 are the precondition and postcondition, respectively. It concisely describes a program that, when executed in a state that satisfies the precondition, terminates in a state that satisfies the postcondition, modifying only the variables present in the frame. In a state that does not satisfy ψ_1, the program $x : [\psi_1, \psi_2]$ aborts: all behaviours are possible and nontermination too. The variable x is used to represent both a single variable and a list of variables; the context should make clear the case.

$e \in Exp$	$::=$	$\mathbf{self} \mid \mathbf{super}$	special 'references'
		$\mid \mathbf{null} \mid \mathbf{error}$	
		$\mid \mathbf{new}\ N$	object creation
		$\mid x$	variable
		$\mid f(e)$	application of built-in function
		$\mid e\ \mathbf{is}\ N$	type test
		$\mid (N)e$	type cast
		$\mid e.x$	attribute selection
		$\mid (e;\ x : e)$	update of attribute
$\psi \in Pred$	$::=$	e	boolean expression
		$\mid \psi \Rightarrow \psi$	
		$\mid (\vee\ i \bullet \psi_i)$	
		$\mid \forall\, x : T \bullet \psi$	

Table 1: Grammar for expressions and predicates

$c \in Com$	$::=$	$le := e$	multiple assignment
		$\mid x : [\psi_1, \psi_2]$	specification statement
		$\mid pc(e)$	parameterised command application
		$\mid c;\ c$	sequential composition
		$\mid \mathbf{if}\ []i \bullet \psi_i \rightarrow c_i\ \mathbf{fi}$	alternation
		$\mid \mathbf{rec}\ Y \bullet c\ \mathbf{end} \mid Y$	recursion, recursive call
		$\mid \mathbf{var}\ x : T \bullet c\ \mathbf{end}$	local variable block
		$\mid \mathbf{avar}\ x : T \bullet c\ \mathbf{end}$	angelic variable block
$pc \in PCom$	$::=$	$pds \bullet c$	parameterisation
		$\mid le.m \mid ((N)le).m$	method calls
		$\mid \mathbf{self}.m \mid \mathbf{super}.m$	
$pds \in Pds$	$::=$	$\varnothing \mid pd \mid pd;\ pds$	parameter declarations
$pd \in Pd$	$::=$	$\mathbf{val}\ x : T \mid \mathbf{res}\ x : T$	

Table 2: Grammar for commands and parameterised commands

For alternation, we use an indexed notation for finite sets of guarded commands. A method in ROOL can use its name in calls to itself in its body. This is the traditional way to define a recursive method. ROOL also includes the construct **rec** $Y \bullet c$ **end**, which defines a recursive command using the local name Y. A (recursive) call Y to such a command is also considered to be a command. The iteration command can be defined using a recursive command. Blocks **var** $x : T \bullet c$ **end** and **avar** $x : T \bullet c$ **end** introduce local variables. The former introduces variables that are demonically initialised; their initial values are arbitrary. The latter introduces variables that are angelically chosen [Back and von Wright 1998]. This kind of variable is also known as logical constant, logic variable, or abstract variable. In practice, angelic variables

only appear in specification statements.

As methods are seen as parameterised commands, which can be applied to a list of arguments, yielding a command, a method call is regarded as the application of a parameterised command. A call $le.m$ refers to a method associated to the object denoted by le. In a method call $e_1.m(e_2)$, e_1 must be a left expression.

As we intend to apply object-oriented programming laws as a means to prove more complex transformations like refactorings, we consider the application of object-oriented programming laws as two-fold: verification of conditions and program transformation itself. Below we present an example of a law. We write '$(\rightarrow)$' when some conditions must be satisfied for the application of the law from left to right. We use '$(\leftarrow)$' to indicate the conditions that are necessary for applying a law from right to left. We use '$(\leftrightarrow)$' to indicate conditions necessary in both directions. Conditions are described in the **provided** clause of laws.

To eliminate a class to which there are no references in a program, we apply Law 1 $\langle$*class elimination*$\rangle$, from left to right. This application requires that the name of the class declared in cd_1 must not be referred to in the whole program. In order to apply this law from right to left, the name of the class declared in cd_1 must be distinct from the name of all existing classes; the superclass that appears in the declaration cd_1 is **object** or is declared in *cds*. Finally, only method redefinition is allowed for the class declared in cd_1. The application of this law from right to left introduces a new class in a program.

Law 1 $\langle$*class elimination*$\rangle$
$cds\ cd_1 \bullet c = cds \bullet c$

provided

$(\rightarrow)$ The class declared in cd_1 is not referred to in *cds* or c;

$(\leftarrow)$ (1) The name of the class declared in cd_1 is distinct from those of all classes declared in *cds*; (2) the superclass appearing in cd_1 is either **object** or declared in *cds*; (3) and the attribute and method names declared by cd_1 are not declared by its superclasses in *cds*, except in the case of method redefinitions.

□

Using Law 2 $\langle$*attribute elimination*$\rangle$, from left to right, we remove an attribute from a class, since this attribute is not read or written inside such class. The notation $B.a$ refers to access to an attribute a via expressions whose static type is exactly B. Applying this law from right to left, we introduce an attribute in a class, since this attribute is new, not declared by such class, nor is declared by any superclass or subclass.

Law 2 $\langle$*attribute elimination*$\rangle$

class B **extends** A
  pri $a : T$; ads
  ops
end
	$=_{cds,c}$	**class** B **extends** A
		  ads
		  ops
		end

provided

(→) $B.a$ does not appear in *ops*;

(←) a does not appear in *ads* and is not declared as an attribute by a superclass or subclass of B in *cds*.

□

To eliminate a method from a class we use Law 3 ⟨*method elimination*⟩. We can eliminate a method from a class if it is not called by any class in *cds*, in the main command c, nor inside class C. For applying 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. We write $B \leq A$ to denote that a class B is a subclass of a class A.

Law 3 ⟨*method elimination*⟩

class C **extends** D
  ads
  meth $m \mathrel{\widehat{=}} pc$ **end**; *ops*
end
	$=_{cds,c}$	**class** C **extends** D
		  ads
		  ops
		end

provided

(→) $B.m$ does not appear in cds, c nor in *ops*, for any B such that $B \leq C$.

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

□

There are also laws to deal with moving attributes and methods to superclasses, changing types of attributes and parameters, eliminating method calls, for instance, and other features [Borba et al. 2004, Cornélio 2004]. Some programming laws presented in [Borba et al. 2004, Cornélio 2004] can be considered basic refactorings when compared to the classification of refactorings presented by Opdyke [Opdyke 1992].

3 Program Facts Database Structure

Our aim is to check side-conditions of programming laws relying on a Prolog factbase that represents the abstract syntax tree of a ROOL program. The main reason to use Prolog is due to its declarative nature, allowing us to concentrate in the solution, not in the process to describe a solution. We have implemented a compiler to translate a ROOL program to a fact base. In fact, the input to our

```
package(PackageId, MainId).
package(PackageId, ClassId).
class(ClassId, ClassName).
extends(ClassId, SuperClassId).
main(MainId, ‘‘Main’’).
mainCommand(MainId, MainCmdId).
```

Figure 2: Class and main command program facts

```
attribute(AttributeId, ClassId, 'AttributeName').
attributeType(AttributeId, 'Type', 'Visibility').
method(MethodId, ClassId, 'MethodName').
methodVal(MethodId, ValParamId).
methodRes(MethodId, ResParamId).
methodStat(MethodId, StatementId).
```

Figure 3: Attribute and method facts

compiler is a ROOL program that is enriched with tokens that are recognized by the rewriting system Maude [Clavel et al. 2005]. This is a consequence of a decision of using Prolog as a means to verify conditions stated by programming laws, whereas Maude would be used for the implementation of the transformations described by the programming laws.

The compilation process is constituted by two phases. As already said, the compiler receives as input a ROOL program enriched with tokens that can be read by the Maude rewriting system. In the first phase, the compiler reads such an input program and generates an abstract syntactic tree. In the second phase, the compiler scans the syntactic tree and generates Prolog clauses that represent the program syntactic tree.

For every syntactic element of the language we define a fact in Prolog. For instance, we describe facts for classes and the main command (Fig. 2). Although we do not have packages in ROOL, we decided to represent such a concept in program fact bases. We consider that a program is written inside a single package that is introduced in the fact `package` whose first element is the package identifier; the second element is a class identifier. We consider the main command as a particular case: the fact package also lists the identifier of the main command (`main`). Classes are introduced in the fact `class` whose first element is a class identifier and the second is a class name. Attributes and methods of a class have specific facts that take into account the identifier of the class in which they are declared.

For attribute declaration, the fact `attribute` introduces an identifier for an attribute, the identifier of the class in which the attribute is declared, and the

```
assign(CmdId, ParentId, MethodId)
assignExp(CmdId, LeftExpId, ExpId)
```

Figure 4: Assignment facts

```
class CLID 'Account extends CLID 'object {
pri 'balance: Int;
meth MtID 'getBalance ^ = res 'r: int .# 'r := self.'balance; # end
meth MtID 'setBalance ^ = val 's: int .# self.'balance := 's; # end
new ^ = self.'balance := 0;
}
main <
var 'acct : Account .#
    'acct := new 'Account();
    'acct.'setBalance(10);
# end >
```

Figure 5: Input example program

attribute name. The type and visibility of an attribute are introduced by the fact `attributeType`. As already discussed, methods in ROOL are declared as parameterised commands. We separate facts about parameter declarations from facts about the method body itself, the method statement (command). Method parameters are described by distinguishing them according to the parameter passing mechanism. The declaration of a result parameter is introduced by the fact `methodRes`; we use `methodVal` for a value parameter. To introduce the parameter identifier, we use fact `varDec` for variable declaration (see Appendix A). The method body is introduced by the fact `methodStat` (for method command). The first element of this fact is the identifier of the method, and a list of identifiers of commands that appear in the method body.

As an example of facts about a command, we present facts related to assignment (Fig. 4). The fact `assign` introduces an assignment identifier, the parent command in which the assignment appears (a guarded command, for instance), and the identifier of the method in which the assignment is introduced. The assignment itself is constituted by a left-expression (the assignment target), and the expression that is assigned to the left-expression, the right-hand side of the assignment. More examples of fact can be found in Appendix A.

In Fig. 5, we present the program that appears in Fig. 1 in the syntax that is also read by the Maude rewriting system. After compiling the program presented in Fig. 5, we obtain program facts as those presented in Fig. 6. Here we present just a subset of the facts for the program presented in Fig. 6.

```
package(1000, 1001).
class(1001, "'Account").
extends(1001, 0000). //Object  0000
attribute(1002, 1001, "'balance").
attributeType(1002, "int", "pri").

method(1003, 1001, "'getBalance").
methodRes(1003, 1004).            varDec(1004, 1003, 1003, "'r").
methodStat(1003, 1005).           VarDecType(1004, "int").

assign(1005, 1003, 1003).         exp(1006, 1005, 1003, "'r", null).
assignExp(1005, 1006, 1007).      exp(1007, 1005, 1003, "self", 1008).
                                  exp(1008, 1007, 1003, 'balance, null).
package(1000, 1020).
main(1020, "Main").                mainCommand(1020, 1021).
varDec(1020, 1001, null, "'acct"). assign(1021, 1020, null).
varDecType(1020, "'Acoount").      assignExp(1021, 1022, 1023).
                                   exp(1022, 1021, null, "'acct", null).
new(1023, 1021, null).
newExp(1023, 1024, 1021).       exp(1024, 1023, null, "'Account", null).
```

Figure 6: Program facts for the example program of Fig. 5

```
superClass(ClassID, Ancestor) :-
   extends(ClassID, Ancestor).
superClass(ClassID, Ancestor) :-
   extends(ClassID,Y),
   superClass(Y,Ancestor).
```

Figure 7: Clause for checking superclass relation

4 Verifying Side-Conditions of Programming Laws

In this section, we present how we have implemented the verification of conditions for application of programming laws. Here we deal just with the activity of verifying conditions for the application of programming laws, not with the program transformation that is a consequence of a law application. We encoded in a Prolog program the conditions that appear in the object-oriented programming laws presented in [Borba et al. 2004, Cornélio 2004]. Our implementation can be seen as constituted by layers. The bottom layer is constituted by the facts that represent the abstract syntax tree of a program; the second layer is composed by clauses that express conditions associated to programming laws. The third layer is constituted by conjunction of clauses expressing the conditions of programming laws.

Some clauses appear in the implementation of conditions of distinct laws. For instance, the clause `superClass` (Fig. 7) checks whether a class whose identifier is `classId` has class `Ancestor` as superclass. Notice that this clause is based on the fact `extends` of program factbases. We use clause `attributeNameClass` (Fig. 8) to check whether an attribute name appears in a given class. In this

```
attributeNameClass(AttributeName,ClassName):-
    returnAttributeID(AttributeName, AttributeId),
    returnClassID(ClassName,ClassId),
    atributeIDClass(AttributeId, ClassId).
returnAttributeID(AttributeName, AttributeId):-
    attribute(AttributeId, _, AttributeName, _, _).
```

Figure 8: Checking attribute name in class and attribute identifier retrieval

```
verifyClassIsSuperclass(ClassName):-
    returnClassID(ClassName, ClassID) ,
    package(PackageId,ClassID),
    subClass(ClassID, Descendant),
    package(PackageId,Descendant).
```

Figure 9: Verifying whether a class has subclasses

clause, we use clause `returnAttributeID`, besides other clauses, that returns the attribute identifier. We consider clause `returnAttributeID` as basic.

Here we describe how we verify the conditions associated to the laws we presented in Section 2. Let us consider the conditions of Law 1 ⟨*class elimination*⟩, for its application from left to right. To eliminate a class, it cannot be referred to in the entire program. In other words, such a class cannot be type of attributes (`verifyClassAttType`), variables (`verifyClassVariableType`), and parameters (`verifyClassParamType`). Also, this class is not superclass of any class (`verifyClassIsSuperclass`), it is not used in type casts and tests (`verifyClassIsCast` and `verifyClassTypeTest`), and does not appear in **new** expressions (`verifyClassInNewExp`). The negation of these clauses define clause `verifyClassEliminationLR` (see (1)).

```
verifyClassEliminationLR(ClassName,PackageId):-
    verifyClass(ClassName,ProgramID),
    not(verifyClassAttType(ClassName)),
    not(verifyClassParamType(ClassName)),
    not(verifyClassVariableType(ClassName)),
    not(verifyClassIsSuperclass(ClassName)),
    not(verifyClassIsCast(ClassName)),
    not(verifyClassTypeTest(ClassName)),
    not(verifyClassInNewExp(ClassName)).
```
(1)

Clause `verifyClassIsSuperclass` (Fig. 9) verifies whether a class identified by its class name is superclass of any class in a program. For this, it is necessary to retrieve the class identifier from the class name. We use the identifier to check if

any other class in a program is a subclass of a class with such an identifier. On the other hand, to introduce a class in a program we have to check that the name of the class we are introducing is not already in the program—we negate clause `verifyClass` that checks if a class is already in a program. Moreover, we check if the superclass of the class we introduce is already in the program or is **object** (clause `verifySuperclasAlreadyDeclared`). We have also to guarantee that attributes of the new class are not present in the superclass by negating clause `verifyAttSuperlasses`. We have to be more careful with methods. If the new class we are introducing declares a method with a name that is already a name of a method of any superclass of the class being introduced, we require that this method has the same parameters, since in ROOL overloading is not allowed. These conditions are conjoined in the clause `verifyClassEliminationRL` (see (2)).

```
verifyClassEliminationRL(ClassName,PackageId):-
    not(verifyClass(ClassName,PackageId)),
    verifySuperclasAlreadyDeclared(ClassName),
    not(verifyAttSuperlasses(ClassName)),
    ((verifyMethNameInSuperclasses(ClassName),
    verifyMethParamSuperclasses(ClassName));
    (not(verifyMethNameInSuperclasses(ClassName))).
```
(2)

The clause `attElimLR` (see (3)) implements the condition to remove a private attribute by using Law 2 ⟨*attribute elimination*⟩, from left to right. The condition requires that the attribute is not read or written inside a class, which is expressed by clause `priAttAccess`.

```
attElimLR(AttributeName,ClassName) :-
    not(priAttAccess(AttributeName, ClassName)).
```
(3)

On the other hand, to introduce an attribute, we require that it is not already present in a class—we negate `attDecClass`—nor it is declared in any superclass or subclass—we negate `attDecHierarchy`. This is expressed by the clause `attElimRL` (see (4)).

```
attElimRL(AttributeName,ClassName) :-
    not(attDecClass(AttributeName, ClassName)),
    not(attDecHierarchy(AttributeName, ClassName)).
```
(4)

The condition for applying Law 3 ⟨*method elimination*⟩, from left to right, is expressed in condition `methElimLR` (see (5)). We require there are no calls to the method in the entire program. For this moment, we deal with the static type of attributes, variables, and parameters to check the type of method call targets to a method. We require that there are no calls on attributes whose type is the class from which we eliminate the method or any of its superclasses. This is also applied to variables and parameters. We have also to check object chains. For instance, consider a method call like $x.y.z.m()$, in which m is the method we

want to eliminate. A method call as the one given above does not necessarily implies that we cannot eliminate method m, we have to check the declared type of z. If it is the class from which we intend to eliminate m or any of its subclasses, we cannot eliminate m, otherwise we can.

```
methElimLR(MethodName, ClassName) :-
  not(methodCallOnAttribute(MethodName, ClassName)),
  not(methodCallOnVar(MethodName, ClassName)),
  not(methodCallOnParam(MethodName, SubClassName)),
  not(methodCallOnObjectChains(MethodName, ClassName)).
```
(5)

To introduce a method in a class, we require the method to be new in the hierarchy, not declared in the class, nor in any superclass or subclass (see (6)).

```
methElimRL(MethodName, ClassName) :-
 newMethodInHierarchy(MethodName, ClassName).
```
(6)

These are examples of verification of conditions of some object-oriented programming laws using logic programming. Besides these laws, we have implemented verification of conditions of other 19 programming laws. We have 25 laws altogether.

5 Related Work

Opdyke [Opdyke 1992] formally describes conditions that must be satisfied to apply a refactoring. Some "low-level" refactorings [Opdyke 1992, Chapter 5] proposed by Opdyke are, in fact, programming laws of our language. This is the case of the refactoring that deals with the introduction of attributes, for instance. In fact, some conditions of programming laws are similar to conditions of Opdyke's "low-level" refactorings.

Roberts [Roberts 1999] goes a step further than Opdyke and describes both preconditions and postcondition of refactorings, allowing support for refactoring chains. The definition of postconditions allows the elimination of program analysis that are required within a chain of refactorings. This comes from the observation that refactorings are typically applied in sequences intended to set up preconditions for later refactorings. Pre- and postconditions are all described as first-order predicates; this allows the calculation of properties of sequences of refactorings. We have not defined postconditions of programming laws; we describe the transformations of such laws as meta-programs, not by means of properties they have.

Kniesel [Kniesel 2005] enables conditional transformations to be specified from a *minimal* set of built-in conditions and transformations. In our approach, conditions of a refactoring, for instance, are defined with basis on object-oriented programming law conditions. In this way, applications of programming laws serve to derive more complex transformations (refactorings) that can be applied to programs. We have not defined a minimal set of conditions for law

application, but defined layers that are composed by Prolog clauses. The bottom layer is the program factbase itself. Upon the factabse, we define conditions that may be common to different laws—like condition `verifyClass` that can be considered basic—or specific to a law. Notice that these conditions are clearly stated by programming laws. Kniesel [Kniesel 2005, Kniesel and Koch 2004] defines *conditional transformation (CT)* to be a program transformation guarded by a precondition, such that the transformation is performed only if its precondition is true. We can also view a programming law as a transformation with condition that can be checked by the existence of elements in a program. The JTransformer program transformation engine [Rho et al. 2003], which is used as a backend for conditional transformations, has inspired us in the definition of our logic factbase. The conditions we implemented, as already said, are based on programming law conditions.

Li [Li 2006] has defined refactoring for Haskell programs along with a refactorer called HaRe. Some functional refactorings have object-oriented counterparts like renaming. However, there are refatorings that are specific to functional programs like the one that deals with monadic computation of expressions. The Haskell refactorer (HaRe) deals with structural, module, and data-oriented refactorings. HaRe is based on Strafunski [Lammëll and Visser 2003] which is a Haskell-centered software bundle for implementing language processing components and can be instantiated to different programming languages. Since programming languages of different paradigms have distinct program structures, they have their own program collection of refactorings. In our case, conditions and transformations are based on programming laws that were described and proved against the semantics of an object-oriented language [Cornélio 2004].

Tools that implement refactorings, like Eclipse [ecl], usually have a larger set of refactoring than ours, as we deal with a subset of sequential Java. Since our object-oriented programming laws deal with a language with a copy semantics, we can define refactorings that deal mainly with program structures, not involving sharing. On the other hand, we have identified some limitations of Eclipse when dealing with casts and moving methods to a superclass [Cornélio 2004]. Also, differently from Eclipse, we intend to build a tool that allows programmers to define their own refactorings.

6 Conclusions

Changes in software are usually consequence of evolution or correction. However, some changes are performed to improve program structure, leading to a program that is easier to understand and to maintain. These changes modify the program structure without affecting the software behaviour as perceived by users.

A disciplined way to change a program without affecting its behaviour is to apply programming laws that guarantee correctness of program transformation by construction. This is based on proof of soundness of programming laws; in our case of laws that deal with imperative and object-oriented constructs [Borba et al. 2004, Cornélio 2004]. To apply an object-oriented program-

ming law, conditions have to be satisfied. In this work, we presented an application of logic programming to check if the conditions of programming laws are satisfied, allowing us to strictly apply a law. Refactoring developers can take programming laws as a toolkit for the development of new refactorings. By using a tool that encodes programming laws, refactorings obtained are correct by construction.

We took advantage of using a declarative language like Prolog that facilitates the description of conditions. This can also be used to define preconditions for the application of a sequence composition of programming laws. In fact, this goes in the direction of the ConTraCT refactoring editor [Kniesel 2005]. It should also be necessary to describe the transformation defined by programming laws in a logic program. In fact, we have to deal with program transformations, we are considering the use of Constraint Handling Rules [Frühwirth 1998] or Transaction Logic [Bonner and Kifer 1994] in order to implement the transformations expressed by programming laws as direct changes in program factbases.

We have already used rewriting systems for the mechanical proof of refactoring rules [Júnior et al. 2005]. However, we have not verified conditions for the application of programming laws. Our work here and the one presented in [Júnior et al. 2005] can be viewed as complementary. The conditions for a transformation would be checked by the implementation in logic programming, whereas the transformation would be realised by the rewriting system in which programming laws are encoded.

Acknowledgements

The authors are supported by the Brazilian Research Agency, CNPq, grant 506483/2004-5.

References

[ecl] *Eclipse*. Eclipse.org. Available on-line http://www.eclipse.org/. Last accessed in March, 2007.

[Back and von Wright 1998] Back, R. J. R. and von Wright, J. (1998). *Refinement Calculus: A Systematic Introduction*. Springer-Verlag.

[Bonner and Kifer 1994] Bonner, A. J. and Kifer, M. (1994). An Overview of Transaction Logic. *Theoretical Computer Science*, 133(2):205–265.

[Borba et al. 2004] Borba, P., Sampaio, A., Cavalcanti, A., and Cornélio, M. (2004). Algebraic Reasoning for Object-Oriented Programming. *Science of Computer Programming*, (52):53–100.

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

[Cavalcanti and Naumann 2000] Cavalcanti, A. L. C. and Naumann, D. A. (2000). A Weakest Precondition Semantics for Refinement of Object-oriented Programs. *IEEE Transactions on Software Engineering*, 26(8):713–728.

[Cavalcanti et al. 1999] Cavalcanti, A. L. C., Sampaio, A., and Woodcock, J. C. P. (1999). An Inconsistency in Procedures, Parameters, and Substitution in the Refinement Calculus. *Science of Computer Programming*, 33(1):87–96.

[Clavel et al. 2005] Clavel, M. et al. (2005). *Maude Manual*. SRI International.

[Cornélio 2004] Cornélio, M. L. (2004). *Refactorings as Formal Refinements*. PhD thesis, Centro de Informática, Universidade Federal de Pernambuco. Also available at `http://www.dsc.upe.br/~mlc`http://www.dsc.upe.br/∼mlc.
[Dijkstra 1976] Dijkstra, E. W. (1976). *A Discipline of Programming*. Prentice-Hall.
[Fowler 1999] Fowler, M. (1999). *Refactoring: Improving the Design of Existing Code*. Addison-Wesley.
[Frühwirth 1998] Frühwirth, T. (1998). Theory and Practice of Constraint Handling Rules. *Journal of Logic Programming, Special Issue on Constraint Logic Programming*, 37(1-3):95–138.
[Júnior et al. 2005] Júnior, A. C., Silva, L., and Cornélio, M. (2005). Using CafeOBJ to Mechanise Refactoring Proofs and Application. In Sampaio, A., Moreira, A. F., and Ribeiro, L., editors, *Brazilian Symposium on Formal Methods*, pages 32—46.
[Kniesel 2005] Kniesel, G. (2005). ConTraCT - A Refactoring Editor Based on Composable Conditional Program Transformations. In Lämmel, R., Saraiva, J., and Visser, J., editors, *Generative and Transformational Techniques in Software Engineering*. Pre-proceedings of the International Summer School, GTTSE 2005.
[Kniesel and Koch 2004] Kniesel, G. and Koch, H. (2004). Static Composition of Refactorings. *Science of Computer Programming*, (52):9—51.
[Lammëll and Visser 2003] Lammëll, R. and Visser, J. (2003). A Strafunski Application Letter. In Dahl, V. and Wadler, P., editors, *Proc. of Practical Aspects of Declarative Programming*, volume 2562 of *LNCS*, pages 357–375. Springer-Verlag.
[Li 2006] Li, H. (2006). *Refactoring Haskell Programs*. PhD thesis, University of Kent.
[Meyer 1997] Meyer, B. (1997). *Object-Oriented Software Construction*. Prentide-Hall, second edition.
[Morgan 1994] Morgan, C. C. (1994). *Programming from Specifications*. Prentice Hall, second edition.
[Opdyke 1992] Opdyke, W. (1992). *Refactoring Object-Oriented Frameworks*. PhD thesis, University of Illinois at Urbana-Champaign.
[Rho et al. 2003] Rho, T. et al. (2003). *JTransformer Framework*. Computer Science Department III–University of Bonn. http://roots.iai.uni-bonn.de/research/jtransformer/.
[Roberts 1999] Roberts, D. B. (1999). *Practical Analysis for Refactoring*. PhD thesis, University of Illinois an Urbana-Champaign.

A Facts for Commands and Expressions

A.1 Variable Declaration

```
varDec(VarDecId, ParentId, MethodId, 'VariableName')
varDecType(VarDecId, 'Type')
avarDec(AVarDecId, ParentId, MethodId, 'VariableName')
avarDecType(AVarDecId, 'Type')
```

A.2 Parameterized Command and Method Call

```
pCommand(CmdId, ParentId, MethodId)
methodCall(MethodCallId, ParentId, MethodId)
methodCallMeth(MethodCallId, ExpId, MethodName)
methodCallExp(MethodCallId, ExpCallId)
```

A.3 Types Test and Cast

```
is(IsId, ParentId, MethodId)
isExp(IsId, ExpId, 'ClassName')
cast(CastId, ParentId, MethodId)
castExp(CastId, ClassName, ExpId)
```