A Refinement Algebra for Object-Oriented Programming

Paulo Borba, Augusto Sampaio, and Márcio Cornélio[⋆]

Informatics Center
Federal University of Pernambuco
Recife, PE, Brazil

Abstract. In this article we introduce a comprehensive set of algebraic laws for ROOL, a language similar to sequential Java but with a copy semantics. We present a few laws of commands, but focus on the object-oriented features of the language. We show that this set of laws is complete in the sense that it is sufficient to reduce an arbitrary ROOL program to a normal form expressed in a restricted subset of the ROOL operators. We also propose a law for data refinement that generalises the technique from traditional modules to class hierarchies. Together, these laws are expressive enough to derive more elaborate rules that can be useful, for example, to formalize object-oriented design practices; this is illustrated through the systematic derivation of a refactoring from the proposed laws.

1 Introduction

The laws of imperative programming are well-established and have been useful both for assisting software development and for providing precise algebraic programming language semantic definitions [1, 2]. In fact, besides being used as guidelines to informal programming practices, programming laws establish a sound basis for formal and rigorous software development methods. Moreover, algebraic laws have proven to be an important tool for the design of correct compilers and code optimisers [3].

Other programming paradigms have also benefited from algebraic semantics which give a sound basis for program derivation and transformation. The laws of Occam [4], for example, exhibit nice and useful properties of concurrency and communication. Algebraic properties of functional programming are elegantly and deeply addressed in [5]. An algebraic approach to reasoning about logic programming is presented in [6]. Even unifying theories [7] have been proposed to classify and study different paradigms, considering a variety of semantic presentations in an integrated way: denotational, operational and algebraic.

In spite of all these efforts, the laws of object-oriented programming are not yet well-established. Some laws have been informally discussed in the object-oriented literature [8, 9]. Some others have been formalised to the degree that

[⋆] Fax: +55 81 32718438, email: {phmb,acas,mlc2}@cin.ufpe.br.

they can be encoded in refactoring tools [10, 11], but they have not been formally proved to be sound or complete in any sense. Opdyke [10] proposes a set of preconditions for application of refactorings, whereas Roberts [11] goes a step further introducing postconditions for them. The definition of correctness used by Roberts is based on test suites. There are no formal proofs that refactoring a program preserves its behavior or that it continues meeting its specification. In fact, there seems to be no comprehensive, provably sound, set of laws to help developers understand and use the properties of medium grain programming units, and of mechanisms such as classes, inheritance, and subtyping. Furthermore, some of the laws of imperative programming are not directly applicable to corresponding small grain object-oriented units and constructs. For instance, due to dynamic binding, the laws of procedure calls are not valid for method calls.

Small grain object-oriented constructs have been considered [12, 13], but medium grain constructs have been largely neglected. A great deal of work, as those presented in [14–17], has already been carried out on transformations for design models in the Unified Modelling Language (UML) [18], but those naturally do not consider programming and behavioral specification constructs. Although those UML transformations are proved sound with respect to a formal semantics, no completeness result has been reported to our knowledge.

In this article we describe a comprehensive set of laws for ROOL (Refinement Object-Oriented Language) [19, 20], which is based on the sequential subset of Java [21] but has a copy semantics. While we illustrate a few laws that deal with the imperative features of the language, we concentrate on laws for its object-oriented features. Besides clarifying aspects of the semantics of ROOL, the laws serve as a basis (interface) for deriving more elaborate rules for practical applications of program transformation. Indeed, we show how the proposed laws can be used to derive refactorings [8] which capture informal object-oriented design and maintenance practices. In addition to algorithmic refinement, we introduce a law for data refinement that allows change of data representation in class hierarchies, contrasting with traditional laws that deal with a single program module [2].

An important contribution of this work is to show that the proposed set of laws for the object-oriented features of ROOL is complete in the sense that it is sufficient to transform an arbitrary program into a normal form expressed in terms of a small subset of the language constructs, following the usual approach adopted for imperative and concurrent languages [1, 4], among others. Using the laws, we describe and justify a strategy for reducing programs to normal form. This does not suggest a compilation process for ROOL; its sole purpose is to prove a completeness result and therefore suggest that our set of laws is expressive enough to derive the more elaborate and useful transformation rules mentioned above. Programs in the normal form defined here have the same type hierarchy as the original program, but with no method declarations, and attribute declarations appearing only in **object**, ROOL's universal class.

This article is organised as follows. We first give an overview of ROOL (Section 2). After that we characterise the normal form and introduce a set of laws for the object-oriented features of ROOL, explaining and justifying how they can be used to define a strategy for reducing an arbitrary program to normal form (Section 3). Some additional laws of commands and expressions, and a law for refining class hierarchies, are presented in Section 4. Then we show how the proposed laws can serve as a basis for proving refactorings (Section 5). Finally, we summarise our results and discuss topics for further research (Section 6).

2 An Overview of ROOL

ROOL is an object-oriented language based on the sequential subset of Java [22] with a copy semantics rather than a reference semantics. The copy semantics significantly simplifies reasoning and still allows us to consider parts of Java programs that do not have reference aliasing. ROOL has been specially designed to allow reasoning about object-oriented programs and specifications, hence it mixes both kinds of constructs in the style of Morgan's refinement calculus [2].

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 as in the following example, where we define a class called *Client*.

```
class Client extends Object
   pri name : string;  pri addr : Address; ...
   meth getStreet ≙ res r : string • addr.getStreet(r) end;
   meth setStreet ≙ val s : string • addr.setStreet(s) end;
   new ≙ addr := new Address end;
end
```

Subclassing and single inheritance are supported through the **extends** clause. The built-in **object** class is a superclass of any other class in ROOL, so the **extends** clause above could have been omitted. Besides the **pri** qualifier for private attributes, there are visibility qualifiers for protected (**prot**) and public (**pub**) attributes, with similar semantics to Java. For simplicity, ROOL supports no attribute redefinition and has only public methods, which can have value and result parameters. The list of parameters of a method is separated from its body by the symbol '•'. Constructors are declared by the **new** clause and do not have parameters. In contrast to Java, ROOL adopts a simple semantics for constructors: they are syntactic sugar for methods that are called after creating objects of the corresponding class.

In addition to method calls, as illustrated in the *Client* class, the body of methods and constructors may have imperative constructs similar to those of the language of Morgan's refinement calculus. This is specified by the definition of the commands of ROOL:

$$\begin{array}{lll}
c \in Com ::= & le := e \mid c;\ c & \text{assignment, sequence} \\
 & \mid x : [\psi_1, \psi_2] & \text{specification statement} \\
 & \mid pc(e) & \text{parameterized command application} \\
 & \mid \mathbf{if}\ [\!]\, i : 1 \mathinner{\ldotp\ldotp} n \bullet \psi_i \ \rightarrow\ c_i\ \mathbf{fi} & \text{alternation} \\
 & \mid \mathbf{rec}\ X \bullet c\ \mathbf{end} \mid X & \text{recursion, recursive call} \\
 & \mid \mathbf{var}\ x : T \bullet c\ \mathbf{end} & \text{local variable block} \\
 & \mid \mathbf{avar}\ x : T \bullet c\ \mathbf{end} & \text{angelic variable block}
\end{array}$$

where *le* are the expressions allowed to appear as target of assignments and as result arguments. These expressions are defined later. A specification statement $x : [\psi_1, \psi_2]$ is useful to concisely describe a program that can change only the variables listed in the frame x, and when executed in a state that satisfies its precondition ψ_1 terminates in a state satisfying its postcondition ψ_2. The frame x lists the variables whose values may change. Like the languages adopted in other refinement calculi, ROOL is a specification language where programs appear as an executable subset of specifications.

Methods are seen as parameterized commands, which can be applied to a list of arguments to yield a command (the entry '$pc(e)$' in the description of commands). Therefore method calls are represented as the application of parameterized commands:

$$\begin{array}{lll}
pc \in PCom ::= & pds \bullet\ c & \text{parameterization} \\
 & \mid le.m \mid ((N)le).m & \text{method calls} \\
 & \mid \mathbf{self}.m \mid \mathbf{super}.m & \\
pds \in Pds \ \ ::= & \varnothing \mid pd \mid pd;\ pds & \text{parameter declarations} \\
pd \in Pd \quad\ ::= & \mathbf{val}\ x : T \mid \mathbf{res}\ x : T &
\end{array}$$

A parameterized command $pds \bullet c$ has parameters *pds* that are used in the command c. The parameterized command $le.m$ is a call to method m with target object *le*. Parameters can be passed by value (**val**) or by result (**res**).

The conditional (alternation) command is written in the same style as in Dijkstra's language. ROOL also provides recursion and variable blocks. Angelic variables are similar to local variables, but they have their initial values angelically chosen so that c succeeds. Data types T are either primitive (**bool**, **int**, and others) or class names. For simplicity, we consider that methods cannot be mutually recursive, but classes can.

For building expressions, ROOL supports typical object-oriented constructs:

$$\begin{array}{lll}
e \in Exp ::= & \mathbf{self} \mid \mathbf{super} & \text{special `references'} \\
 & \mid \mathbf{null} \mid \mathbf{new}\ N & \text{null `reference', object creation} \\
 & \mid x \mid f(e) & \text{variable, built-in application} \\
 & \mid e\ \mathbf{is}\ N \mid (N)e & \text{type test, type cast} \\
 & \mid e.x \mid (e;\ x : e) & \text{attribute selection and update}
\end{array}$$

where **self**, **super**and **is** have similar semantics to `this`, `super` and `instanceof` (which does not require exact type matching) in Java, respectively. 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 the value of e_2, in a similar way to update of arrays in the refinement calculus [2]. So, despite its name, the update expression, similarly to all ROOL expressions, has no side-effects; in fact, it creates a new object instead of updating an existing one. Variables can, however, be updated through the execution of commands, as in $o := (o;\ x : e)$, which is semantically equivalent to $o.x := e$, and updates o. Expressions such as $\mathbf{null}.x$ and $(\mathbf{null};\ x : e)$ cannot be successfully evaluated; they yield the special value **error** and lead the commands in which they appear to *abort*, as explained in the sequel.

The expressions that are allowed to appear as the target of assignments and as result arguments, define the *Le* (*left expressions*) subset of *Exp*:

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

From a theoretical point of view, ROOL can be viewed as a complete lattice whose ordering is a refinement relation on specifications. The bottom (**abort**) of this lattice is the worst possible specification:

$$\mathbf{abort} \ = \ x : [\mathbf{false}, \mathbf{true}]$$

It is never guaranteed to terminate (precondition **false**), and even when it does, its outcome is completely arbitrary (postcondition **true**). It allows any refinement; for instance, programs setting x to arbitrary values. On the other extreme we have the top (**miracle**) of the lattice; it is the best possible specification

$$\mathbf{miracle} \ = \ x : [\mathbf{true}, \mathbf{false}]$$

which can execute in any state (precondition **true**) and establishes as outcome the impossible postcondition **false**.

Although these extreme specifications are not usually deliberately written (**miracle** is not even feasible as an executable program), they are useful for reasoning. For instance, it is normally useful in program derivation or transformation to assume that a condition b holds at a given point in the program text. This can be characterized as an *assumption* of b, designated as $\{b\}$, defined as follows:

$$\{b\} \ = \ : [b, \mathbf{true}]$$

Note that, if b is **false**, an assumption reduces to **abort**. Otherwise it behaves like a program that always terminates and does nothing, denoted by **skip**:

$$\mathbf{skip} \ = \ : [\mathbf{true}, \mathbf{true}]$$

Further considerations about specification statements, ROOL, and its formal semantics based on weakest preconditions are given elsewhere [2, 19, 20]. In the next section, we define algebraic laws for the object-oriented features of ROOL, and show that these laws are complete in the sense that they can be used to define a normal form for ROOL. We give more details about the language constructs only when necessary.

3 A Normal Form for Class Hierarchies

In order to show that the proposed set of laws is comprehensive, we define a reduction strategy (based on the laws) whose target is a normal form described in terms of a restricted subset of the ROOL operators.

Definition 1 *(Subtype Normal Form) A* ROOL *program cds • c is in Subtype Normal Form if it obeys the following conditions:*

- *Each class declaration in cds, except* **object***, has an empty body; the inheritance and subtype clause* **extends** *might be included, but no declaration of methods, constructors or attributes is allowed in subclasses.*
- *The* **object** *class may include only attribute declarations, each with either a primitive type or* **object** *itself.*
- *All local declarations in the main command c are declared with either a primitive type or* **object***.*
- *No type cast is allowed in c.*

Although this normal form still preserves some object-oriented features (the subtype hierarchy, object creation and type test) it is substantially close to an imperative program. In particular, the **object** class, the only one with explicitly declared elements, takes the form of a recursive record, as only attributes are permitted. As no methods are allowed, the main command c also looks very much like an imperative program, although object creation and type test can still be used.

Further reduction could lead to the complete elimination of all object-oriented features, in which case the natural normal form would be the imperative subset of ROOL extended with recursive records. A reduction to such a form would require 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 keep the type information. We target the Subtype Normal Form, which is sufficiently close to an imperative program, and the additional laws for a reduction to a pure imperative program can be easily identified, as further discussed in Section 4.

The reduction strategy involves the following major steps:

- Move all the code (attribute and method declarations) in *cds* to the **object** class;
- Change all the declarations of object identifiers to type **object**;
- Eliminate casts;
- Eliminate method calls and declarations.

In the remainder of this section we present the reduction strategy in detail, as a sequence of simple and incremental steps.

3.1 General Assumptions

Before we start the detailed presentation of the reduction strategy, we introduce some simple and syntactic conditions that will ensure the applicability of the laws and the convergence of the overall strategy. The following conditions are assumed:

1. In the set of class declarations *cds*, two distinct classes are not allowed to declare attributes with the same name.
2. Similarly, two distinct classes in *cds* are not allowed to declare methods with the same name, except in the case where a method of a superclass is being redefined.
3. All references to an attribute or method *op* are of the form $e.op$, for any expression e, including **self**.

The first two conditions are necessary to ensure that the laws to move attributes and methods up in the inheritance hierarchy are always applicable, not generating name conflicts. The third condition allows a uniform treatment of attribute and method occurrences. The roles of these conditions are further motivated in the detailed normal form reduction steps which follow. It is important to note that they do not impose any significant constraint on our approach; they can be easily achieved by simple automatic purely syntactic transformations. Moreover, our laws are also valid for programs that do not satisfy these conditions. The conditions are only necessary for simplifying the normal form reduction process, which will not need intermediate steps for simply renaming methods and attributes.

3.2 Make Attributes Public

Aiming to move all the code up to **object**, the first step in our reduction strategy (process) would be to move attributes up. But before that we should make sure that they are either public or protected, otherwise method declarations in the subclasses might become invalid. For simplicity, we make all attributes public so that we have to deal only with this case in the remaining steps of the reduction process.

In order to make attributes public, we apply two laws. The following one indicates that a protected attribute can be turned into a public one, and vice-versa, provided that the attribute is only directly used by its class and the associated subclasses. This proviso is necessary to guarantee that the law relates valid ROOL programs. We use the notation '**prot** $a : T$; *ads*' to denote the set of attribute declarations containing '**prot** $a : T$' and all the declarations in *ads*, whereas *ops* stands for the declarations of operations (object constructor and methods). Equivalence of sets of class declarations cds_1 and cds_2 is denoted by $cds_1 =_{cds,c} cds_2$, where *cds* is the context of 'auxiliary' declarations for cds_1 and cds_2, and c is the main program. This is just an abbreviation for the corresponding behavioural program equivalence: $cds_1\, cds \bullet c = cds_2\, cds \bullet c$ [20].

Law 1 ⟨change visibility: from protected to public⟩

$$\boxed{\begin{array}{l} \textbf{class}\ C\ \textbf{extends}\ D \\ \quad \textbf{prot}\ a : T;\ ads \\ \quad ops \\ \textbf{end} \end{array}} \quad =_{cds,c} \quad \boxed{\begin{array}{l} \textbf{class}\ C\ \textbf{extends}\ D \\ \quad \textbf{pub}\ a : T;\ ads \\ \quad ops \\ \textbf{end} \end{array}}$$

provided

(←) $B.a$, for all $B \leq C$, appears only in *ops* and in the subclasses of C in *cds*.

The typed name $B.a$ refers to uses of the attribute name a via expressions of type B. So when we write that $B.a$ does not appear in *ops* we mean that *ops* does not contain any expression in the form $e.a$ for any expression e of type B, strictly (that is, the type of e cannot be a subclass of B). The subclass relation is denoted by $\leq$, or $\leq_{cds}$ when the set of auxiliary class declarations *cds* is not clear from the context.

Since we want to turn protected attributes into public ones, we should apply Law 1 from left to right. In this direction, the condition for applying the law is trivially valid since a protected attribute can only appear inside its class and the associated subclasses. In fact, the above condition is relevant only when applying the law in the other direction, for turning a public attribute into a protected one. That is why we write '(←)' before the condition. We also write '(→)', when a proviso is necessary only for applying a law from left to right, and '(↔)', when it is necessary in both directions.

For turning the private attributes into public ones, we apply the following law from left to right. The proviso is trivially valid for similar reasons to the ones discussed for the previous law.

Law 2 ⟨change visibility: from private to public⟩

$$\boxed{\begin{array}{l}\textbf{class}\ C\ \textbf{extends}\ D\\ \quad\textbf{pri}\ a : T;\ ads\\ \quad ops\\ \textbf{end}\end{array}} \quad =_{cds,c} \quad \boxed{\begin{array}{l}\textbf{class}\ C\ \textbf{extends}\ D\\ \quad\textbf{pub}\ a : T;\ ads\\ \quad ops\\ \textbf{end}\end{array}}$$

provided

(←) $B.a$, for all $B \leq C$, does not appear in *cds*, c.

Another obvious law related to the previous ones allows us to change attribute visibility from private to protected. This is omitted here since it is not necessary in our reduction strategy; it can also be trivially derived from the previous two laws.

By exhaustively applying Laws 1 and 2 to all classes in *cds*, in any order, we turn all protected and private attributes into public ones.

3.3 Move Attributes Up

After making all attributes public, we can move them up to the **object** class. This is justified by the following law. It establishes that we can move a public attribute from a subclass to a superclass as long as this does not generate a name conflict, such as when a superclass and one of its subclasses declare an attribute with the same name[1]. This law also indicates that we can move the attribute in the other direction provided that it is used only as if it were declared in the subclass, otherwise we would obtain an invalid program.

Law 3 ⟨move attribute to superclass⟩

$$\boxed{\begin{array}{l} \textbf{class}\ B\ \textbf{extends}\ A \\ \quad ads \\ \quad ops \\ \textbf{end} \\ \textbf{class}\ C\ \textbf{extends}\ B \\ \quad \textbf{pub}\ a : T;\ ads' \\ \quad ops' \\ \textbf{end} \end{array}} \quad =_{cds,c} \quad \boxed{\begin{array}{l} \textbf{class}\ B\ \textbf{extends}\ A \\ \quad \textbf{pub}\ a : T;\ ads \\ \quad ops \\ \textbf{end} \\ \textbf{class}\ C\ \textbf{extends}\ B \\ \quad ads' \\ \quad ops' \\ \textbf{end} \end{array}}$$

provided

($\rightarrow$) The attribute name a is not declared by the subclasses of B in cds;
($\leftarrow$) $D.a$, for all $D \leq B$ and $D \nleq C$, does not appear in cds, c, ops or ops'.

Notice that, according to the special notation $D.a$ previously introduced, the second condition above precludes an expression such as **self**.a from appearing in ops, but does not preclude **self**.$c.a$, for an attribute '$c : C$' declared in B. The last expression is valid in ops no matter a is declared in B or in C.

Starting from the bottom of the class hierarchy defined by cds and moving upwards in the hierarchy towards **object**, the exhaustive application of this law, from left to right, will move all attributes to **object**. The name conflicts that could be generated during this process are avoided because, as mentioned in Section 3.1, we assume that no two classes declare attributes with the same name; this can be easily achieved by renaming attributes. Therefore the condition for applying the law from left to right becomes trivially valid.

3.4 Trivial Cast Introduction

To enable and simplify the next steps, we generate a uniform program text in which occurrences of an identifier are cast wherever possible. Casts will not be introduced only for primitive type identifiers and for occurrences of identifiers as targets of assignments or result arguments, since this is not allowed in ROOL. This step is necessary, for instance, to guarantee that method bodies are valid when moved to a superclass. For example, a method body such as $x :=$ **self** would not be valid in a superclass, assuming that the type of x is a subclass C, but the corresponding body with a cast, $x := (C)$**self**, would be valid.

The following law formalizes the fact that any expression can be trivially cast with its own type. The notation $cds, N \triangleright e : T$ asserts that the expression e that can appear in the body of a method in class N has type T. Similarly, the

[1] ROOL does not allow attribute redefinition or hiding as in Java.

notation $cds, N \rhd c \ = \ d$ indicates that the equation $c \ = \ d$ holds inside class named N, in a context defined by the set of class declarations *cds*. Instead of a class name, we use **main** in the notations above for asserting that the typing or equality holds inside the main program.

Law 4 ⟨introduce trivial cast in expressions⟩
If $cds, A \rhd e : C$, then

$$cds, A \rhd e \ = \ (C)e$$

This is formalised as a law of expressions, not commands. It can be considered as an abbreviation for several laws of assignments, conditionals, and method calls that deal with each possible pattern of expressions. For example, it abbreviates the following laws all with a similar antecedent to Law 4.

$$cds, A \rhd (le := e.x) \ = \ le := ((C)e).x$$
$$cds, A \rhd le.m(e) \ = \ le.m((C)e)$$

This is equally valid for left-expressions, which are a form of expression.

The previous two laws are sufficient to introduce trivial casts in an arbitrary ROOL program. As a result, all non-assignable object expressions are cast, either because they were originally cast, or because casts were introduced by the current step of our reduction strategy. An additional observation, however, is that the transformations described by the above laws (as well as subsequent laws which relate commands) must be carried out not only inside classes, but also in the main program.

3.5 Eliminate super

Before moving methods up to **object**, we should also make sure that their bodies do not contain references to **super**, otherwise the program semantics could not be preserved. Indeed, when moving a method up, instead of referring to a method m of an immediate superclass C we might end up referring to the method m of a superclass of C. Furthermore, the resulting program could even be invalid, since **super** cannot appear in **object**.

Our approach for eliminating **super** relies on the following law, which indicates that we can replace 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 5 ⟨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 B
  ads′
  ops′
end
```

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

$$cds\ CDS, C \rhd \textbf{super}.m \ = \ pc$$

where ‘*cds CDS*’ denotes the union of the class declarations in *cds* and *CDS*.

However, notice that a method called via **super** is not always declared in the immediate superclass of the class where the call appears. In this situation Law 5 cannot be applied. So, in order to avoid that situation, we apply the following law before eliminating **super**. It basically says that we can introduce or remove a trivial method redefinition.

Law 6 ⟨introduce method redefinition⟩

$$\boxed{\begin{array}{l}\textbf{class}\ B\ \textbf{extends}\ A\\ \quad ads\\ \quad \textbf{meth}\ m \mathrel{\widehat{=}} pc\ \textbf{end};\ ops\\ \textbf{end}\\ \textbf{class}\ C\ \textbf{extends}\ B\\ \quad ads'\\ \quad ops'\\ \textbf{end}\end{array}} \quad = \quad \boxed{\begin{array}{l}\textbf{class}\ B\ \textbf{extends}\ A\\ \quad ads\\ \quad \textbf{meth}\ m \mathrel{\widehat{=}} pc\ \textbf{end};\ ops\\ \textbf{end}\\ \textbf{class}\ C\ \textbf{extends}\ B\\ \quad ads'\\ \quad \textbf{meth}\ m \mathrel{\widehat{=}} \textbf{super}.m\ \textbf{end};\\ \quad ops'\\ \textbf{end}\end{array}}$$

provided

(→) m is not declared in ops'.

Strictly, we cannot define a method as **meth** $m \mathrel{\widehat{=}} \textbf{super}.m$. A method declaration is an explicit parametrised command, so that, above, *pc* has the form $(pds \bullet c)$; the redefinition of m should be **meth** $m \mathrel{\widehat{=}} (pds \bullet \textbf{super}.m(\alpha pds))$, where αpds gives the list of parameter names declared in *pds*. For simplicity, however, we adopt the shorter notation **meth** $m \mathrel{\widehat{=}} \textbf{super}.m$.

In fact, for eliminating **super**, we first introduce several method redefinitions using **super** itself. This is exhaustively done by applying Law 6, from left to right, whenever the proviso is valid, for all methods of all superclasses in *cds*, starting from **object** and moving downwards in the class hierarchy.

After introducing the trivial method redefinitions, every class will redefine (trivially or not) all the methods in its superclass. So we can exhaustively apply Law 5, from left to right, for eliminating all method calls using **super**. This elimination process starts at the immediate subclasses of **object** and move downwards. As the methods of **object** cannot refer to **super**, and all attributes are already public at this point, the conditions of Law 5 are valid for the immediate subclasses of **object**. After eliminating **super** from those classes, the conditions will be valid for their immediate subclasses and so on.

3.6 Move Methods Up

After introducing trivial casts and eliminating **super**, we can safely move methods up to **object**. This is justified by two laws. We apply the first one when the method declaration that we want to move up is a redefinition of a method declared in the immediate superclass. This law states that we can merge the two method declarations into a single one in the superclass. The resulting method body uses type tests to choose the appropriate behaviour. As mentioned in Section 3.1, **self** is not omitted in calls and selections of methods and attributes of the same class where they appear.

Law 7 ⟨move redefined method to superclass⟩

```
class B extends A
  ads
  meth m ≙ pds • b end;
  ops
end
class C extends B
  ads′
  meth m ≙ pds • b′ end;
  ops′
end
```

$=_{cds,c}$

```
class B extends A
  ads
  meth m ≙ pds •
    if ¬(self is C) → b
    [] self is C → b′
    fi
  end; ops
end
class C extends B
  ads′
  ops′
end
```

provided

(↔) **super** and private attributes do not appear in b';
super.m does not appear in ops';
(→) b' does not contain uncast occurrences of **self** nor expressions in the form $((C)$**self**$).a$ for any private or protected attribute a in ads';
(←) m is not declared in ops'.

Most of these provisos are necessary to guarantee that the application of the law to a valid ROOL program yields a valid program as well. The exceptions are those concerning **super**; the semantics of expressions and commands that use this construct might be affected by moving code from a subclass to a superclass, or vice-versa, as discussed in Section 3.5.

The other law that justifies moving methods up should be applied when the method that we want to move is not a redefinition. In this case we can only move a method up if this does not introduce a name conflict[2]. The law indicates that we can move a method down too, if this method is used only as if it were defined in the subclass.

Law 8 ⟨move original method to superclass⟩

[2] ROOL supports method overriding but not method overloading in general. Hence we cannot have different methods in the same class, or in a class and a subclass, with the same name but different parameters.

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

$=_{cds,c}$

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

provided

($\leftrightarrow$) **super** and private attributes do not appear in *pc*;
m is not declared in any superclass of B in *cds*;

($\rightarrow$) *m* is not declared in *ops*, and can only be declared in a class D, for all $D \leq B$ and $D \nleq C$, if it has the same parameters as *pc*;
pc does not contain uncast occurrences of **self** nor expressions in the form $((C)\mathbf{self}).a$ for any private or protected attribute a in ads';

($\leftarrow$) *m* is not declared in ops';
$D.m$, for all $D \leq B$, does not appear in *cds*, *c*, *ops* or ops'.

The conditions for applying this law are similar to the ones of Laws 7 and 3. Only the first two are necessary to preserve semantics. The others guarantee that we relate only syntactically valid programs. We preclude superclasses of B from defining m because this could affect the semantics of commands such as '$b.m(\ldots)$', for a b storing an object of B, when moving the declaration of m.

Using Laws 7, 8, and 4 we can move all method declarations to **object**. We start from the bottom of the class hierarchy defined by *cds* and move upwards to **object**. In a given class C, it does not matter if we move first the methods that do not call other methods declared in C; even the calls for methods of C not declared in its superclass will be valid in the superclass since at this point the occurrences of **self** are all cast. In order to maintain that, we need to apply Law 4 immediately after applying Law 7, since the last one introduces two uncast references to **self**.

Following this process, the conditions for applying Law 7 from left to right are valid because, at this phase of the reduction to normal form, all attributes are already public and declared in **object** (so ads' is empty), and all method bodies do not use the **super** construct nor have uncast references to **self** (which is a non-assignable expression). This also explains why some of the conditions for applying Law 8 from left to right are valid too. In order to understand why the other conditions are valid, recall from Section 3.1 that method names cannot be reused and notice that, after eliminating **super** (see Section 3.5), every class redefines the methods in its superclass. So if a method m is declared in C but not in B, we can then assume that it is not declared in any superclass of B.

3.7 Change Type to object

After moving all the code up, we eliminate type casts from the entire program. This is the purpose of this and the following step. One immediate condition to eliminate type casts in general is that all the identifiers (attributes, local variables and parameters names) that have a class type[3] must have the same type. In order to illustrate that, consider a class A, a subclass B of A, and the following declarations: '$a : A$' and '$b : B$'. Then, in ROOL (as well as in Java) the assignment of a to b must be cast, as in '$b := (B)a$'. Therefore, unless we replace the types of those identifiers with a single type, we will not be able to get rid of arbitrary type casts. The natural candidate type to play this role is the **object** class.

Surprisingly, perhaps, changing the type of an identifier to a superclass of its declared type can be justified provided all the occurrences of such an identifier in non-assignable expressions—that is, when not used as target of assignments and result arguments—are cast with its original type. The reason is that in this case one cannot distinguish between an identifier being declared with its original type or with a supertype of this type. The apparent paradox comes from the fact that eliminating casts requires changing the types of identifiers, and that may require the introduction of type casts. Nevertheless, as seen in Section 3.4, the introduction of such casts is trivial, because the casts are always to the declared types of the identifiers.

Here we need some laws to formalize the fact that the types of identifiers for attributes, variables, and value parameters can be replaced with a supertype if all occurrences of these identifiers in non-assignable expressions are cast. In such cases, the change of type will cause no interference, as already explained. As an example, we present a law for attributes. The laws for variables and parameters are similar. For a result parameter, we require, in addition, the types of the corresponding arguments to be greater than or equal to the new parameter type.

Law 9 ⟨change attribute type⟩

$$\boxed{\begin{array}{l}\textbf{class}\ C\ \textbf{extends}\ D\\ \quad \textbf{pub}\ a : T;\ ads\\ \quad ops\\ \textbf{end}\end{array}} \quad =_{cds,c} \quad \boxed{\begin{array}{l}\textbf{class}\ C\ \textbf{extends}\ D\\ \quad \textbf{pub}\ a : T';\ ads\\ \quad ops\\ \textbf{end}\end{array}}$$

provided

($\leftrightarrow$) $T \leq T'$ and every occurrence of a in non-assignable expressions of *ops*, *cds* and c is cast with T or any subtype of T declared in *cds*;

($\leftarrow$) every expression assigned to a, in *ops*, *cds* and c, is of type T or any subtype of T; every use of a as result argument is for a corresponding formal parameter of type T or any subtype of T.

[3] Recall that primitive type expressions cannot be cast.

The exhaustive application of this law (together with those for variables and parameters), from left to right, instantiating the type T' with **object**, allows the replacement of the types of all identifiers with the **object** class. Recall that at this point all identifiers having a class type are trivially cast, so the conditions of the law are valid.

3.8 Cast Elimination

As the types of the identifiers were changed to **object**, the trivial casts introduced by the step described in Section 3.4 are not trivial anymore. Furthermore, the program might include arbitrary casts previously introduced by the programmer. Therefore, we need new laws for eliminating nontrivial casts.

It is also worth observing that a type cast plays two major roles. At compilation time, casting is necessary when using an expression in contexts where an object value of a given type is expected, and this type is a strict subtype of the expression type. For example, if '$x : B$', $C \leq B$ and a is an attribute which is in C but not in B, then the selection of this attribute using x must be cast, as in $((C)x).a$. If a were declared in B, then the cast would not be necessary concerning compilation purposes, but once it is there it cannot simply be eliminated, because a cast also has a run time effect.

At run time, if the object value of a cast expression does not have the required type, then the expression evaluation results in **error** and the command in which this expression appears behaves like **abort**. Therefore, in the example discussed above (and assuming the attribute a is in class B) although the cast could be directly eliminated regarding its static effect, it still has a dynamic effect when the object value of x happens to be of type B but not of type C.

In order to be able to eliminate a cast while still preserving its dynamic behaviour, we use assumptions. Recall, from Section 2, that the assumption $\{b\}$ behaves like **skip** if b is true, and as **abort** otherwise. The following laws deal with the elimination of type casts in expressions. However, unlike Law 4 which deals with casts regardless of their contexts, here we need to consider the particular context in which the cast is used, in order to determine whether it can be eliminated. The static requirements of type casts are captured by the antecedent of each law, while the dynamic behaviour of the cast is replaced by an assumption. For example, the following law removes casts in assignment statements, while the next law eliminates casts in method calls.

Law 10 $\langle$eliminate cast of expressions$\rangle$
If $cds, A \rhd le : B$, $e : B'$, $C \leq B'$ and $B' \leq B$, then

$$cds, A \rhd le := (C)e \;=\; \{e\ \textbf{is}\ C\};\ le := e$$

Law 11 $\langle$eliminate cast of method call$\rangle$
If $cds, A \rhd e : B$, $C \leq B$ and m is declared in B or in any of its superclasses in cds, then

$$cds, A \rhd ((C)e).m(e') \;=\; \{e\ \textbf{is}\ C\};\ e.m(e')$$

These are just two possible cases. Observe that a type cast might occur arbitrarily nested in an expression. Therefore, in order to deal with cast elimination in general, it is convenient to reduce expressions to a simple form, so that we can then consider only a fixed number of patterns. This form is basically as defined in the BNF for expressions (see Section 2), with the replacement of arbitrary expressions (denoted by e) with variables. The reduction of an arbitrary expression to this form is a reduction strategy in itself. Nevertheless, it is a very standard one, and is not presented here; this kind of reduction strategy can be found elsewhere [3].

The laws to deal with the elimination of casts in the remaining expression patterns are very similar to the previous two laws, so are omitted here. Observe that, at this stage of our reduction strategy, all casts can be eliminated. The static role of each cast is trivially fulfilled as a consequence of the fact that the type of each identifier is **object**, and that all methods and attributes of the entire program have been moved to the **object** class. Therefore, the provisos of each law are always satisfied. Complementarily, the dynamic effect of each cast is preserved by the assumption on the right-hand side of each law. As a result, the exhaustive application of these laws eliminates all casts in the program.

3.9 Method Elimination

The purpose of this step is to eliminate all method calls and then all method declarations, keeping in the **object** class only attribute declarations. For method call elimination, observe that we need only a simple law which can be regarded as a version of the the *copy rule*. The reason is that we have already dealt with dynamic binding in Section 3.6, when moving methods all the way up to **object**. Therefore, in general the body of each method is a large conditional whose guards resolve the possible dynamic binding aspects. In fact, there are no method redefinitions at this point, since all methods are already in **object**.

Law 12 ⟨method call elimination⟩
Consider that the following class declaration

$$\begin{array}{l}\textbf{class}\ C\ \textbf{extends}\ D\\ \quad ads\\ \quad \textbf{meth}\ m \mathrel{\widehat{=}} pc\ \textbf{end};\ ops\\ \textbf{end}\end{array}$$

is included in *cds*. Then

$$cds, A \rhd e.m(e') \;=\; \{e \neq \textbf{null} \wedge e \neq \textbf{error}\};\ pc[e/\textbf{self}](e')$$

provided

($\leftrightarrow$) $cds, A \rhd e : C$; m is not redefined in *cds* and does not contain references to **super**; and all attributes which appear in the body of m (i.e., pc) are public.

Note that a method call $e.m(e')$ aborts when e is **null** or **error**. Thus, we need the assumption $\{e \neq \textbf{null} \land e \neq \textbf{error}\}$ on the right-hand side of the above law which also aborts when e is **null** or **error**; otherwise, the assumption behaves like **skip** and the method body behaves the same as its invocation.

After all calls of a method are replaced with its body, the method definition itself can be eliminated. Similarly to the notation introduced in Section 3.2, in the following law $B.m$ stands for calls of the method m via expressions of type B, strictly.

Law 13 ⟨method elimination⟩

$$\boxed{\begin{array}{l} \textbf{class}\ C\ \textbf{extends}\ D \\ \quad ads \\ \quad \textbf{meth}\ m \mathrel{\widehat{=}} pc\ \textbf{end};\ ops \\ \textbf{end} \end{array}} =_{cds,c} \boxed{\begin{array}{l} \textbf{class}\ C\ \textbf{extends}\ D \\ \quad ads \\ \quad ops \\ \textbf{end} \end{array}}$$

provided

(→) $B.m$ does not appear in cds, c nor in ops, for all B such that $B \leq C$.
(←) m is not declared in ops nor in any superclass or subclass of C in cds.

In this step, we should apply Law 12 exhaustively for eliminating all method calls. Before doing so, however, we need to eliminate all recursive calls. This should be done by defining the recursive methods using the recursive command $\textbf{rec}\ X \bullet c\ \textbf{end}$, in such a way that recursive calls become references to X. The law that can be used to perform this change is standard and omitted.

After eliminating all method calls, Law 13 is then applied to remove method definitions. There is no particular order to be followed; methods can be eliminated in any order. Even in the case where a method m invokes a method n, it is possible to eliminate m first, as in every place where m is invoked we can replace this invocation by the body which includes an invocation to n; but this is no problem since n is still in scope. Remember that at this point there are no private attributes, method redefinitions or references to **super**.

3.10 Summary of the Strategy

The main result of this work is captured by the following theorem which summarizes the overall reduction strategy.

Theorem 1 *(Reduction strategy) An arbitrary* ROOL *program satisfying the conditions stated in Section 3.1 can be reduced to Subtype Normal Form.*

Proof*: From the application of the steps described in sections 3.2–3.9, in that order.*

The proof of the above theorem is straightforward because the details of the strategy concerning its convergence in terms of the applicability of the laws is discussed in each individual step.

4 Additional Laws and Data Refinement

Although our normal form reduction strategy has hopefully uncovered an interesting and expressive set of laws, it might be surprising that some additional laws were not necessary in our reduction process. Note that this is a consequence of the fact that our subtype normal form preserves classes, attributes, type tests, and object creation. As explained in Section 3, we decided to aim at this normal form because its reduction process is entirely algebraic, whereas reduction to a pure imperative form requires some sort of encoding of the object model data into, for instance, a relational one. Nevertheless, some additional laws necessary for reduction to a pure imperative form can be easily identified.

The next two laws deal with classes and attributes. Law 14 removes a class declaration provided it is not used.

Law 14 ⟨class elimination⟩

$$cds\ cd_1 \bullet c = cds \bullet c$$

provided

(→) The class declared cd_1 does not appear in cds and c;

(←) The name of the class declared in cd_1 is distinct from those of all classes declared in cds; the superclass appearing in cd_1 is either **object** or declared in cds; 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.

If a private attribute is not read or written inside the class in which it is declared, we can remove it by using Law 15.

Law 15 ⟨attribute elimination⟩

$$\boxed{\begin{array}{l}\textbf{class}\ B\ \textbf{extends}\ A\\ \quad \textbf{pri}\ a : T;\quad ads\\ \quad ops\\ \textbf{end}\end{array}} =_{cds,c} \boxed{\begin{array}{l}\textbf{class}\ B\ \textbf{extends}\ A\\ \quad ads\\ \quad ops\\ \textbf{end}\end{array}}$$

provided

(→) $B.a$ does not appear in ops;

(←) a does not appear in ads, and it is not declared by a superclass or subclass of B in cds.

Another construct preserved by our normal form is the type test (**is**). Two simple laws of type test are presented below. These are actually laws of expressions, which usually refer to the context in which the expression occurs. Law 16 asserts that the type test **self is** N is true when appearing inside the class named N.

Law 16 ⟨**is** test true⟩

$$cds, N \rhd \mathbf{self\ is}\ N = \mathbf{true}$$

Complementary to Law 16, Law 17 asserts that the test **self is** M is false inside a class N, provided N is not a subclass of M, and vice-versa.

Law 17 ⟨**is** test false⟩
If $N \nleq M$ and $M \nleq N$, then

$$cds, N \rhd \mathbf{self\ is}\ M = \mathbf{false}$$

Apart from the laws of the object-oriented features of ROOL, laws of imperative commands are also necessary in practical applications of program transformation. These laws are simple adaptations of those found in the literature [1, 4] and are not the focus of this work. However, as an example of an imperative law of ROOL, we present Law 18 that allows us to simplify an alternation whose guarded commands are the same in all branches of the alternation, assuming that the disjunction of all guards of the alternation is true.

Law 18 ⟨**if** identical guarded commands⟩
If $\vee\ i : 1 \mathinner{..} n \bullet \psi_i = \mathbf{true}$, then

$$\mathbf{if}\ []i : 1 \mathinner{..} n \bullet \psi_i \ \rightarrow\ c\ \mathbf{fi}\ =\ c$$

As another example of law for an imperative command, Law 19 states that the order of the guarded commands of an alternation is immaterial.

Law 19 ⟨**if** symmetry⟩
If π is any permutation of $1 \mathinner{..} n$, then

$$\mathbf{if}\ []i : 1 \mathinner{..} n \bullet \psi_i \ \rightarrow\ c_i\ \mathbf{fi}\ =\ \mathbf{if}\ []i : 1 \mathinner{..} n \bullet \psi_{\pi(i)} \ \rightarrow\ c_{\pi(i)}\ \mathbf{fi}$$

For the transformation of programs, in general, we also need to apply class refinement. The traditional techniques of data refinement deal with modules that encapsulate the variables whose representations are being changed. In our approach, this is extended to consider hierarchies of classes whose attributes are not necessarily private: they can be protected or public. Law 20 allows us to introduce new attributes in a class, relating them with already existing attributes by means of a coupling invariant CI. The application of this law changes the bodies of the methods declared in the class and in its subclasses. The changes follow the traditional laws for data refinement [2].

Law 20 ⟨superclass attribute-coupling invariant⟩

$$\boxed{\begin{array}{l} \textbf{class}\ A\ \textbf{extends}\ C \\ \quad adsa; \\ \quad ops \\ \textbf{end} \\ cds' \end{array}} \preceq_{cds,c} \boxed{\begin{array}{l} \textbf{class}\ A\ \textbf{extends}\ C \\ \quad adsc; \\ \quad CI(ops) \\ \textbf{end} \\ CI(cds') \end{array}}$$

where

CI distributes over language constructs modifying commands according to traditional data refinement laws;

provided

$B.a$, for all $B \leq A$ and public attribute a in $adsa$, does not appear in cds or c;

CI refers only to public and protected attributes in $adsa$;

cds' only contains subclasses of A, and cds contains no subclasses of A.

By convention, the attributes denoted by $adsa$ are abstract, whereas those denoted by $adsc$ are concrete. The coupling invariant CI relates abstract and concrete states. The notation $CI(cds')$ indicates that CI acts on the class declarations of cds'. The application of CI to a class declaration changes the methods of such a class according to the laws of data refinement [2]: every guard may assume the coupling invariant; every command is extended by modifications to the new variables so that the coupling invariant is maintained. These transformations are also done in the class A; this is indicated by the notation $CI(ops)$. In order to apply this law, the public attributes in $adsa$ must not be accessed in the command c nor in any method of the classes in cds. Also, the coupling invariant CI must refer only to public and protected attributes in $adsa$, since these must be visible in the subclasses of A.

This law for data refining class hierarchies together with laws of commands and laws of the object-oriented features of ROOL form a solid basis for proving more elaborate transformations of object-oriented programs, as illustrated in the next section.

5 Formal Refactoring

In this section we present how the laws used for the reduction of ROOL programs to the normal form serve as a basis for justifying program transformations. We are particularly interested in the application of these laws to prove refactorings. Here we present the proof of the refactoring ⟨Pull Up/Push Down Field⟩. In fact, we represent these refactorings by a single law. Applying this law from left to right corresponds to the first refactoring; the reverse direction corresponds to the other one. Here we prove the derivation of this refactoring from left to right, which allows us to move attributes from subclasses to their common superclass. The attributes might have different names, but their types have to be the same. Here we consider public attributes as this is the most general case; private and protected attributes can be turned into public ones using Laws 2 and 1, respectively.

Refactoring 1 ⟨Pull Up/Push Down Field⟩

```
class A extends D                    class A extends D
   adsa                                 pub z : T;  adsa
   opsa                                 opsa
end                                  end
class B extends A                    class B extends A
   pub x : T;  adsb                     adsb
   opsb                    =cds,c       opsb[z/x]
end                                  end
class C extends A                    class C extends A
   pub y : T;  adsc                     adsc
   opsc                                 opsc[z/y]
end                                  end
cds'                                 cds'[z, z/x, y]
```

provided

(↔) *cds'* contains only the subclasses of B and C in which there are occurrences of x and y;

(→) The attribute name z is not declared in *adsa*, *adsb*, *adsc*, nor in any subclass or superclass of A in *cds* and *cds'*; and the attribute names x and y are not declared by *adsb*, *adsc*, nor by any subclass of A in *cds*;

$N.x$, for any $N \leq B$, does not appear in *cds* or c, and $N.y$, for any $N \leq C$, does not appear in *cds* or c;

(←) $N.z$, for any $N \leq A$, $N \nleq B$, and $N \nleq C$, does not appear in *cds* or c;

x (y) is not declared in *adsa*, *adsb* (*adsc*), nor in any subclass or superclass of B (C) in *cds* and *cds'*.

The first step of the proof is to apply Law 3 twice. Each application of this law moves the attributes x and y of classes B and C, respectively, to their common superclass A. Notice that, the attributes are public as required by Law 3. For simplicity, we omit *cds'* in the derivation because modifications to the operations of classes in *cds'* are similar to those done to *opsb* and *opsc*. We use *cds'*$[z, z/x, y]$ to denote that occurrences of x and y in operations of classes in *cds'* are replaced with z.

```
class A extends D            class B extends A      class C extends A
  pub x : T, y : T;  adsa      adsb                   adsc
  opsa                         opsb                   opsc
end                          end                    end
```

The next step is to prepare A and its subclasses for data refinement. This preparation consists of the exhaustive application of a law that we omit here since it is well known. This law [2] transforms assignments of the form $t :=$ **self**$.x$ into its corresponding specification statement $t : [\mathbf{true}, t = \mathbf{self}.x]$. This transformation occurs in all subclasses of A in which there are occurrences of the abstract variables x and y in assignments. After these changes the operations of classes A, B, and C are denoted by $opsa'$, $opsb'$, and $opsc'$, respectively.

We then apply Law 20, introducing the attribute z (the concrete representation of both x and y) into A. The coupling invariant CI, relating z with x and y, is given by the predicate $((\mathbf{self\ is}\ B) \Rightarrow z = x) \wedge ((\mathbf{self\ is}\ C) \Rightarrow z = y)$.

```
class $A$ extends $D$               class $B$ extends $A$      class $C$ extends $A$
  pub $z : T$;                        $adsb$                     $adsc$
  pub $x : T, y : T$;  $adsa$         $CI(opsb')$                $CI(opsc')$
  $CI(opsa')$                       end                        end
end
```

The application of CI changes guards and commands of classes A, B, and C according to the laws of data refinement presented by Morgan [2]. Guards are augmented so that they assume the coupling invariant. The new guard may be just a conjunction of the old guard with the coupling invariant. We augment specifications so that the concrete variable appears in the frame of the specification and the coupling invariant is conjoined with the pre and post conditions. In this way, the specification statement $t : [\mathbf{true}, t = \mathbf{self}.x]$ becomes $t, z : [CI, t = \mathbf{self}.x \wedge CI]$. An assignment to an abstract variable of the form $\mathbf{self}.x := exp$ is augmented to $\mathbf{self}.x, \mathbf{self}.z := exp, exp$. These changes are also applied to classes in cds' that, for simplicity, we omit here. Since the attributes x and y are new in class A, there are no occurrences of them in $opsa'$. Consequently, we can reduce $CI(opsa')$ just to $opsa$ by using refinement laws [2].

The next step is the elimination of occurrences of abstract variables in the subclasses of A. We proceed with diminishing assignments of the form $\mathbf{self}.x, \mathbf{self}.z := exp, exp$ to $\mathbf{self}.z := exp$, as we are replacing the variables that constitute the abstract state with the variables that compose the concrete state. For specification statements of the form $t, z : [CI, t = \mathbf{self}.x \wedge CI]$ we apply Laws 16 and 17 that simplify the conjunction of the coupling invariant. Inside B, Law 16 states that the test $\mathbf{self\ is}\ B$ is true because we are refining the class B. On the other hand, Law 17 states that the test $\mathbf{self\ is}\ N$, for a class N that is not a superclass or a subclass of B is false inside B. This simplifies the coupling invariant to the predicate $(\mathbf{true} \Rightarrow z = x) \wedge (\mathbf{false} \Rightarrow z = y)$ which is trivially $z = x$. The specification statement, at this moment, is $t, z : [z = x, t = \mathbf{self}.x \wedge z = x]$ which is refined by the assignment $t := \mathbf{self}.z$, which is actually $t := \mathbf{self}.x[z/x]$, a renaming of the original code in $opsb$. Guards must be algorithmically refined.

We proceed in the same way with the commands of C that are augmented with concrete variables and that assume the coupling invariant. As the coupling invariant relates abstract and concrete variables via an equality between attribute names, the classes B and C that we obtain after the elimination of abstract variables are the same as the original except that all occurrences of x and y in the commands are replaced with z.

Since the abstract attributes are no longer read or written in B or C and their subclasses, where they were originally declared, we can remove them from A.

First we apply Law 2, from right to left, in order to change the visibility of these attributes to private, since they are are not read or written outside the class in which they are declared. Then we apply Law 15 that allows us to remove a private attribute that is not read or written inside the class it is declared. We proceed in the same way for C.

```
class A extends D          class B extends A       class C extends A
  pub z : T; adsa            adsb                    adsc
  opsa                       opsb[z/x]               opsc[z/y]
end                        end                     end
```

This finishes the proof of the refactoring ⟨Pull Up Field⟩. The reverse direction corresponds to the refactoring ⟨Push Down Field⟩, whose proof is similar to the one presented here. As both sides are refinement of each other, we conclude that they are equal.

6 Conclusions

Defining a rich set of algebraic laws for object-oriented programming seems a desirable and original contribution. Although the laws presented here are for a particular language (ROOL), they will hopefully be of more general utility. In particular, although ROOL is based on copy semantics, whereas most practical object-oriented programming languages are based on reference semantics, the laws for the object-oriented features of ROOL, presented here, do not rely on this design decision. In contrast, some command laws (like those for combining assignments) do rely on copy semantics.

Strategies for normal form reduction such as the one introduced here are usually adopted as a measure of *completeness* of a set of proposed laws, not as the final aim for a practical programmer. In fact, our strategy aims to make a program less object-oriented and does not suggest good development practices or compilation strategies. However, when applied in the opposite direction, the laws used to define the strategy serve as a tool for carrying out practical applications of program transformation. Our completeness result, together with the proposed law for data refinement of class hierarchies, suggests that the proposed set of laws is expressive enough to derive transformation rules that capture informal design practices, such as refactorings, as illustrated in the previous section and in Cornélio's thesis [23].

One aspect which became evident when defining the laws presented here is that, associated with most of them, there are very subtle side conditions which require much attention. Uncovering the complete side conditions has certainly been one of the difficult tasks of our research. This can be contrasted with more practically-oriented work in the literature which focus on the transformations without paying must attention to correctness or completeness issues, as already discussed in Section 1.

Perhaps another interesting issue of this research is the particular approach taken for the reduction strategy, by moving all the code (attributes and methods) all the way up to the **object** class. An alternative would be to move the code down, to the classes in the leaves of the inheritance hierarchy. This has, nevertheless, been shown unsuitable for a systematisation based on algebraic laws. The reason is that moving a single attribute or method from a superclass to a subclass might have great contextual impact, whereas moving declarations up is more controllable, as it causes less effects on other code. The particular approach adopted has allowed us to separate concerns to a great extent, helping to tackle the overall task. For example, the elimination of method invocation (Law 12) has been dissociated from dynamic binding (Law 7), as well as from the behaviour of **super** (Laws 5 and 6).

A common criticism to the algebraic style is that merely postulating algebraic laws can give rise to complex and unexpected interactions between programming constructions; this can be avoided by linking the algebraic semantics with a mathematical model in which the laws can be verified. The command laws of ROOL have already been proved correct with respect to a weakest precondition semantics for the language [20]. The complete link between the algebraic semantics presented here and the weakest precondition semantics of ROOL is the subject of a complementary work [23].

Another complementary work [24], recently completed, to our research is the mechanisation of the reduction strategy, as well as the mechanical proofs of some refactorings, using the Maude [25] term rewriting system.

Acknowledgements

We thank our collaborators Ana Cavalcanti and David Naumann for many discussions that contributed to the research reported in this article. We also thank the anonymous referees for the detailed comments that helped to improve this article. Part of the work reported here was carried out when the first two authors were visiting the Stevens Institute of Technology. The authors are partly supported by CNPq, grants 521994/96–9 (Paulo Borba), 521039/95–9 (Augusto Sampaio), and 680032/99-1 (Dare CO-OP and CO-OP projects, jointly funded by PROTEM-CC and the National Science Foundation).

References

1. Hoare *et al*, C.A.R.: Laws of programming. Communications of the ACM **30** (1987) 672–686
2. Morgan, C.: Programming from Specifications. second edn. Prentice Hall (1994)
3. Sampaio, A.: An Algebraic Approach to Compiler Design. Volume 4 of Algebraic Methodology and Software Technology. World Scientific (1997)
4. Roscoe, A., Hoare, C.A.R.: The laws of occam programming. Theoretical Computer Science **60** (1988) 177–229
5. Bird, R., de Moor, O.: Algebra of Programming. Prentice Hall (1997)
6. Seres, S., Spivey, M., Hoare, T.: Algebra of logic programming. In: ICPL'99, New Mexico, USA (1999)
7. Hoare, C., Jifeng, H.: Unifying Theories of Programming. Prentice Hall (1998)

8. Fowler, M.: Refactoring—Improving the design of existing code. Addison Wesley (1999)
9. Lea, D.: Concurrent Programming in Java. Addison-Wesley (1997)
10. Opdyke, W.: Refactoring Object-Oriented Frameworks. PhD thesis, University of Illinois at Urbana-Champaign (1992)
11. Roberts, D.: Practical Analysis for Refactoring. PhD thesis, University of Illinois at Urbana Champaign (1999)
12. Mikhajlova, A., Sekerinsk, E.: Class refinement and interface refinement in object-oriented programs. In: Proceedings of FME'97. Volume 1313 of Lecture Notes in Computer Science., Springer-Verlag (1997) 82–101
13. Leino, K.R.M.: Recursive Object Types in a Logic of Object-oriented Programming. In Hankin, C., ed.: 7th European Symposium on Programming. Volume 1381 of Lecture Notes in Computer Science., Springer-Verlag (1998)
14. Evans, A.: Reasoning with UML class diagrams. In: Workshop on Industrial Strength Formal Methods, WIFT'98, Florida, USA, IEEE Press (1998)
15. Evans, A., France, R., Lano, K., Rumpe, B.: The UML as a formal modeling notation. In Bézivin, J., Muller, P.A., eds.: The Unified Modeling Language, UML'98 - Beyond the Notation. First International Workshop, Mulhouse, France, June 1998, Selected Papers. Volume 1618 of LNCS., Springer (1999) 336–348
16. Lano, K., Bicarregui, J.: Semantics and transformations for UML models. In Bézivin, J., Muller, P.A., eds.: The Unified Modeling Language, UML'98 - Beyond the Notation. First International Workshop, Mulhouse, France, June 1998, Selected Papers. Volume 1618 of LNCS., Springer (1999) 107–119
17. Gogolla, M., Richters, M.: Transformation rules for UML class diagrams. In Bézivin, J., Muller, P.A., eds.: The Unified Modeling Language, UML'98 - Beyond the Notation. First International Workshop, Mulhouse, France, June 1998, Selected Papers. Volume 1618 of LNCS., Springer (1999) 92–106
18. Booch, G., Jacobson, I., Rumbaugh, J.: The Unified Modelling Language User Guide. Addison-Wesley (1999)
19. Cavalcanti, A., Naumann, D.: A weakest precondition semantics for an object-oriented language of refinement. In: FM'99 - Formal Methods. Volume 1709 of Lecture Notes in Computer Science. Springer-Verlag (1999) 1439–1459
20. Cavalcanti, A., Naumann, D.: A weakest precondition semantics for refinement of object-oriented programs. IEEE Transactions on Software Enginnering **26** (2000) 713–728
21. Arnold, K., Gosling, J.: The Java Programming Language. Addison Wesley (1996)
22. Gosling, J., Joy, B., Steele, G.: The Java Language Specification. Addison-Wesley (1996)
23. Cornélio, M.L.: Applying Object-oriented Refactoring and Patterns as Formal Refinements. PhD thesis, Informatics Center, Federal University of Pernambuco, Brazil (To appear in 2003)
24. Lira, B.O.: Automação de Regras para Programação Orientada a Objetos. Master's thesis, Centro de Informática, Universidade Federal de Pernambuco, Brazil (2002)
25. Meseguer, J.: A logical theory of concurrent objects and its realization in the Maude language. In Agha, G., Wegner, P., Yonezawa, A., eds.: Object-Oriented Programming. MIT Press (1993) 314–390