Laws of ROOL: an Object-Oriented Language

Abstract

In this paper we introduce some basic algebraic laws of ROOL, an object-oriented language similar to Java, but with a copy rather than a reference semantics. One immediate application of the basic laws is the derivation of more elaborate laws which formalise object-oriented design practices. We discuss further applications of the basic laws and the importance of proving their soundness with respect to an independent semantics.

1 Introduction

The laws of imperative programming are well-established and have been useful both for assisting software development and for providing precise axiomatic programming language semantic definitions [11, 15]. 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, axiomatic semantic definitions are an important tool for the design of correct compilers and code optimisers [18].

The laws of object-oriented programming, on the other hand, are not well-established yet [3]. Some laws have been informally described in the object-oriented literature [12, 1, 16], but most of them are still in the minds of object-oriented programmers which intuitively apply them everyday. So, although object-oriented programming is widely used nowadays, there is no comprehensive set of laws to help developers understand and use the properties of *medium grain* programming units and mechanisms such as classes, inheritance, and subtyping. Furthermore, some of the laws of imperative programming are not even directly applicable to corresponding *small grain* object-oriented units and constructs. For instance, due to dynamic binding, the laws of procedure call are not valid for method call. Recent work [13, 14] has considered those small grain constructs, but medium grain constructs have been largely neglected in the literature.

In this article we describe work towards a comprehensive set of basic laws for ROOL (Refinement Object-Oriented Language) [6], which is based on Java but has a copy rather than a reference semantics. We introduce algebraic laws for both small and medium grain constructs; the *laws of commands* consider the small grain constructs, whereas the *laws of classes* consider the medium grain constructs. Besides clarifying aspects of the semantics of ROOL, these laws should serve as a basis from which more elaborate laws for practical applications of program transformation can be derived. Furthermore, it is good practice to show that this set of laws is complete in some sense. The standard approach is to show that the basic set of laws is sufficient to transform an arbitrary program into a normal form expressed in terms of a small subset of the language operators, following the approach adopted, for example, in [11, 17].

This work is in the context of the CO-OP (Calculus of Object-Oriented Programming) project, a joint initiative funded by PROTEM-CC/NSF which aims at defining a formal

semantics for ROOL [6] and proposing and proving basic, design and compilation laws. The design laws guide, justify, and document informal object-oriented programming practices. In particular, design laws might support software evolution practices; for example, we sketch the derivation of a basic law for safely introducing a design pattern, assisting the transition from anti-patterns to patterns [9, 5]. The compilation laws will be used to compile the executable subset of ROOL into a Java Virtual Machine, ensuring correctness of the translation by construction.

This article is organised as follows. We first give an overview of ROOL, following with some of the basic laws of commands and classes. After that we illustrate how more elaborate laws can be derived from the basic laws, and discuss aspects about the soundness of the basic laws. Finally, we summarise the results achieved so far.

2 ROOL

ROOL is an object-oriented language based on Java [10], but with a copy semantics rather than a reference semantics. It 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 [15]. A program in ROOL consists of a *main* command and a set of class declarations. Classes are declared as in the following example:

$$
\begin{array}{l}
\textbf{class}\ \mathit{Client}\ \textbf{extends}\ \mathit{Object} \\
\quad \textbf{pri}\ \mathit{name} : \mathit{string}; \\
\quad \textbf{pri}\ \mathit{add} : \mathit{Address}; \\
\quad \vdots \\
\quad \textbf{meth}\ \mathit{getStreet}\ \mathrel{\widehat{=}}\ \textbf{res}\ r : \mathit{string}\ \bullet\ \mathit{add}.\mathit{getStreet}(r)\ \textbf{end} \\
\quad \textbf{meth}\ \mathit{setStreet}\ \mathrel{\widehat{=}}\ \textbf{val}\ s : \mathit{string}\ \bullet\ \mathit{add}.\mathit{setStreet}(s)\ \textbf{end} \\
\quad \textbf{new}\ \mathrel{\widehat{=}}\ \bullet\ \mathit{add} := \textbf{new}\ \mathit{Address}()\ \textbf{end} \\
\textbf{end}
\end{array}
$$

where subclassing and single code inheritance is 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 actually have been omitted. Besides the **pri** qualifier for private attributes, there are visibility qualifiers for protected and public attributes, with similar semantics to Java. For simplicity, we consider only public methods, which can have value, result and value-result parameters. The list of parameters of a method is separated from its body by the symbol "•". A similar syntax is used for initializers, declared by the **new** clause.

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

$$
\begin{array}{lcll}
c \in \mathit{Com} & ::= & le := e \mid c;\ c & \text{multiple assignment, sequence} \\
 & & \mid x : [\psi, \psi] & \text{specification statement} \\
 & & \mid le.m(e) & \text{method call} \\
 & & \mid \textbf{if}\ []i \bullet \psi_i \rightarrow c_i\ \textbf{fi} & \text{alternation} \\
 & & \mid \textbf{rec}\ X \bullet c\ \textbf{end} \mid X & \text{recursion, recursive call} \\
 & & \mid \textbf{var}\ x : T \bullet c\ \textbf{end} & \text{local variable block} \\
 & & \mid \textbf{avar}\ x : T \bullet c\ \textbf{end} & \text{angelic variable block}
\end{array}
$$

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

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

where **self** has a similar semantics to `this` in Java, and the update $(e_1;\ x : e_2)$ denotes a copy of the object denoted by e_1 but with the attribute x mapped to a copy of e_2. The expressions that are allowed to appear as the target of assignments and method calls, and as result and value-result arguments, define the Le subset of Exp:

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

The elements of Le are called *left expressions*.

Further details about ROOL and its formal semantics are given elsewhere [6]. Here we concentrate on the algebraic laws of the language, giving more details about the language constructs only when necessary. We now illustrate basic laws of commands. After that we consider basic laws of classes.

2.1 Laws of Commands

The laws of commands of ROOL are similar to the laws of imperative languages presented in, for example, [11, 17]. However, ROOL has some commands whose syntax and semantics differ from the languages explored in the cited works, as well as new commands related to its object-oriented features. Thus we need to define new laws for these commands.

For example, the body of an alternation command in ROOL is a *guarded command set* as proposed by Dijkstra [8], while the laws presented in the literature are usually for more restricted versions of alternation. Furthermore, there are the commands which support object-oriented features such as type tests and casts, and method calls whose behaviour cannot be defined by the *copy rule*, due to dynamic binding.

Each law presented in the remainder of this work has a number and a name (suggestive of its use) for further references.

To illustrate the behaviour of the alternation in ROOL, we present the following laws. The first one states that the order of the guarded commands of an alternation is immaterial.

Law 2.1.1 ⟨**if** symmetry⟩ If i ranges over $1..n$ and π is any permutation of $1..n$, then
$\mathbf{if}\ []i \bullet \psi_i\ \rightarrow\ c_i\ \mathbf{fi}\ =\ \mathbf{if}\ []i \bullet \psi_{\pi(i)}\ \rightarrow\ c_{\pi(i)}\ \mathbf{fi}$

An alternation with a single guarded command which has a true guard behaves like the command itself.

Law 2.1.2 ⟨**if** true guard⟩
$\mathbf{if}\ \mathbf{true}\ \rightarrow\ c\ \mathbf{fi}\ =\ c$

On the other hand, a command with a false guard can be eliminated from an alternation, as it would never be executed.

Law 2.1.3 ⟨**if** false unity⟩
$$\textbf{if false} \rightarrow c \ [] \ gcs \ \textbf{fi} \ = \ \textbf{if} \ gcs \ \textbf{fi}$$

Assignment distributes rightward through alternation, replacing occurrences of the assigned variables in the condition by the corresponding expressions.

Law 2.1.4 ⟨:= − ◁▷ right dist⟩
$$le := e; \ \textbf{if} \ []i \bullet \psi_i \rightarrow c_i \textbf{fi} \ = \ \textbf{if} \ []i \bullet \psi_i[e/le] \rightarrow (le := e; \ c_i)\textbf{fi}$$

where $\psi_i[e/le]$ denotes the substitution of e for every occurrence of le in ψ_i.

Concerning the object-oriented features of ROOL, we do not need specific laws for type tests or casts. In the next section we show that type tests can be introduced or eliminated using general laws of assignment and alternation. Type casts are enforced by the typing rules of ROOL [6] and therefore there are no interesting laws concerning their inclusion or elimination from the program.

Nevertheless, we do need laws for method calls. Actually, these laws are the most elaborate ones for commands since, due to dynamic binding, the application of these laws depends on the inheritance hierarchy of the particular context. Here we present a simplified version of the law which considers a class with one subclass which redefines a method, say m. The idea is similar to the copy rule (for procedure call elimination) in the sense of replacing the call by the body of the method, but, due to dynamic binding, we have to inspect the type of the object to discover what is the correct class from which we can obtain the body of the method.

Law 2.1.5 ⟨*method call* elimination⟩ If x has type C, and C' (a subclass of C) redefines the method m of C, then

$$x.m() \ = \ \begin{array}{ll} \textbf{if} & (x \ \textbf{is} \ C') \rightarrow body(x, m, C') \\ [] & (x \ \textbf{is} \ C) \wedge \neg \, (x \ \textbf{is} \ C') \rightarrow body(x, m, C) \\ \textbf{fi} & \end{array}$$

where $body(x, m, X)$ is a function which replaces the method call $x.m()$ with the body of the method m in class X. In the main program (or in a client class) replacing the method by its body must consider the following: commands (for example, assignments) which may access and modify the attributes of the class; calls to methods of the same or of a superclass; and calls to other methods (of client classes).

The details of how these are dealt with when the method call is replaced with its body can be found in [2], together with an extensive set of laws for ROOL, including refinement laws.

2.2 Laws of Classes

Instead of relating commands, the laws of classes relate larger grain programming units: class declarations. So we write

$$cds_1 =_{cds} cds_2$$

in order to indicate that the sets of class declarations cds_1 and cds_2 are equivalent in contexts having the class declarations in cds, which can be seen as the definition of common "auxiliary" classes for both cds_1 and cds_2.

In general, it might be useful to express that the equivalence between class declarations holds just for contexts that use only a subset of the methods and attributes of the related declarations [3, 4]. In this case, we write

$$cds_1 =_{cds,v} cds_2$$

where the view v specifies the methods and attributes that can be used by the contexts where cds_1 is equivalent to cds_2. For any law in this form, we assume that cds_1 and cds_2 are valid class declarations in the context defined by cds. Moreover, we assume that any command c built by using the methods and attributes from v is either valid in the contexts defined by $cds \cup cds_1$ and $cds \cup cds_2$ or invalid in both contexts.

Attributes

The basic laws of classes state simple properties about classes and their relation with attributes, methods and invariants. We now focus on attributes. For instance, the following law states that we can insert a "fresh" private attribute or remove a private attribute that is not used.

In order to formalize the law, several *atts* functions are used to yield the attribute names that appear in declarations. For example, $atts(C, mds)$ yields the names of the attributes of class C that appear on the set of method declarations mds. We also assume that the *family* function yields the declarations of the superclasses and subclasses of a given class, whereas *attsDecs* yields the attribute declarations contained in a given set of class declarations.

Law 2.2.1 $\langle$**pri**$\rangle$
For any attribute name a not in $atts(ads)$, $atts(attDecs(family(C, cds)))$, $atts(C, mds)$ and $atts(C, id)$,

class C **extends** D ads mds id **end**	$=_{cds,v}$	**class** C **extends** D **pri** $a : T$; ads mds id **end**

where T must be either a primitive data type, C, or a class name declared in cds.

The notation "**pri** $a : T$; ads" denotes the set of attribute declarations containing "**pri** $a : T$" and all the declarations in ads, whereas id stands for an initializer declaration.

Another law indicates that a private attribute can be turned into a protected one, and vice-versa, provided that the attribute is not used by subclasses of its associated class. This law, however, is only valid for contexts that do not refer to that attribute.

Law 2.2.2 $\langle$**prot**$\rangle$
If a is not in v and $atts(C, subclasses(C, cds))$ then

class C **extends** D **pri** $a : T$; ads mds id **end**	$=_{v,cds}$	**class** C **extends** D **prot** $a : T$; ads mds id **end**

Although we omit the details here, we also have a similar law for relating protected and public attributes. As a consequence, by transitivity, we can derive a law that relates private and public attributes, under the conditions stated on the previous law.

The semantics of private attributes is further specified by the following law, which states that assignments to private attributes are useless if these are never read.

Law 2.2.3 ⟨useless assignment⟩
For any command context $c[_]$ and expression exp,

class C **extends** D
  pri $a : T$; ads
  meth $m \mathrel{\widehat{=}} pds \bullet$
  c[**skip**]
  end; mds
  id
end
	$=_{cds,v}$	**class** C **extends** D
		  pri $a : T$; ads
		  meth $m \mathrel{\widehat{=}} pds \bullet$
		  $c[a := exp]$
		  end; mds
		  id
		end

if the evaluation of exp—with any of the values allowed for its free variables—does not yield $error$, and if the value of a is not read in mds, c[**skip**] and id; formally, a may only appear as the rightmost variable in the left expressions of assignments and result or value-result arguments.

A similar law is valid for method calls on objects stored on private attributes that are not read—these calls are useless as well. Moreover, there are similar laws that work on initializer bodies instead of method bodies.

Methods

Instead of illustrating properties of attributes, we now consider a basic property of method declarations. The following law states that we can introduce a "fresh" new method or remove a method that is not invoked, but that is only valid in contexts that do not refer to the method. Similarly to the $atts$ function discussed in the previous section, the $meths$ function yields the names of the methods that appear in declarations.

Law 2.2.4 ⟨method declaration⟩
If m is not in v, $meths(C, mds)$, $meths(C, id)$, $meths(methDecs(family(C, cds)))$ and $meths(C, cds)$ then

class C **extends** D
  ads
  mds
  id
end
	$=_{v,cds}$	**class** C **extends** D
		  ads
		  meth $m \mathrel{\widehat{=}} pds \bullet$
		  c
		  end; mds
		  id
		end

where each free variable in c is declared by pds, ads or any superclass of C.

Class Invariants

The last kind of basic law that we consider here establishes the semantics of class invariants. For instance, the following law states that a class invariant is valid at the end of the execution of any method of the corresponding class.

Law 2.2.5 ⟨class invariant after⟩
Let *inv* be an invariant of class C. Then

class C **extends** D
  ads
  meth $m \;\widehat{=}\; pds \bullet$
  c
  end; *mds*
  id
end
	$=_{v,cds}$	**class** C **extends** D
		  ads
		  meth $m \;\widehat{=}\; pds \bullet$
		  c; $\{inv\}$
		  end; *mds*
		  id
		end

where $\{inv\}$ is a command that aborts when *inv* is not valid, and behaves as **skip** when *inv* is valid.

There is also a similar law stating that a class invariant is valid at the beginning of the execution of methods as well, but we omit the details here.

3 Applications

Algebraic laws in the style presented in the previous section have proved useful for several applications, such as proving properties about programs [17], designing compilers [18], and partitioning algorithms [19], all correct by construction.

Such laws are useful for program or specification transformation in general. To give a simple idea of the reasoning style, we state that we can eliminate a type test on an identifier to which we have just assigned an object of the required type.

$$
\begin{array}{l}
\quad \textbf{var}\ x : C \bullet x := \textbf{new}\ C';\ \ \textbf{if}\ \ (x\ \textbf{is}\ C') \rightarrow c_1 \\
\qquad\qquad\qquad\qquad\qquad\quad [] \ \neg\,(x\ \textbf{is}\ C') \rightarrow c_2\ \textbf{end} \\
\qquad\qquad\qquad\qquad\qquad\quad \textbf{fi} \\
= \\
\quad \textbf{var}\ x : C \bullet x := \textbf{new}\ C';\ c_1\ \textbf{end}
\end{array}
$$

Using the algebraic laws of commands presented in the previous section, we can formally

derive the above equation.

$$
\begin{array}{ll}
 & \qquad\qquad\qquad\qquad\qquad\qquad\quad \textbf{if}\ \ (x\ \textbf{is}\ C') \rightarrow c_1 \\
 & \textbf{var}\ x : C \bullet x := \textbf{new}\ C';\ \ [\!]\ \neg\,(x\ \textbf{is}\ C') \rightarrow c_2\ \textbf{end} \\
 & \qquad\qquad\qquad\qquad\qquad\qquad\quad \textbf{fi} \\
= & \ \{\text{Law } \langle := - \lhd\rhd \text{ right dist}\rangle(2.1.4)\} \\
 & \qquad\qquad\quad \textbf{if}\ \ (\textbf{new}\ C'\ \textbf{is}\ C') \rightarrow (x := \textbf{new}\ C';\ c_1) \\
 & \textbf{var}\ x : C \bullet\ [\!]\ \neg\,(\textbf{new}\ C'\ \textbf{is}\ C') \rightarrow (x := \textbf{new}\ C';\ c_2)\ \textbf{end} \\
 & \qquad\qquad\quad \textbf{fi} \\
= & \ \{\text{Laws } \langle\textbf{if}\text{ false unity}\rangle(2.1.3) \text{ and } \langle\textbf{if}\text{ symmetry}\rangle(2.1.1)\} \\
 & \textbf{var}\ x : C \bullet \textbf{if}\ (\textbf{new}\ C'\ \textbf{is}\ C') \rightarrow (x := \textbf{new}\ C';\ c_1)\ \textbf{fi}\ \textbf{end} \\
= & \ \{\text{Law } \langle\textbf{if}\text{ true guard}\rangle(2.1.2)\} \\
 & \textbf{var}\ x : C \bullet x := \textbf{new}\ C';\ c_1\ \textbf{end}
\end{array}
$$

Surprisingly, perhaps, we have not introduced any specific law of type test in the previous section. General laws of assignment and alternation were sufficient to prove the above statement. Although the example is very simple, it illustrates the program transformation style using algebraic laws. In the reminder of this section we present a more elaborate example which illustrates the use of the laws of classes and method call.

3.1 Class Restructuring Laws

Here we show how the basic laws can be used for deriving design laws that justify class restructuring practices. This shall be extremely useful for software evolution, when programmers are interested on restructuring code in order to achieve modularity and, consequently, improve software reuse and extensibility.

For instance, by using the basic laws we can derive design laws that justify informal and common restructuring practices such as splitting a class like

$$
\begin{array}{l}
\textbf{class}\ \mathit{Client}\ \textbf{extends}\ \mathit{Object} \\
\quad \textbf{pri}\ \mathit{name} : \mathit{string};\ \ \textbf{pri}\ \mathit{street} : \mathit{string}; \\
\quad \textbf{pri}\ \mathit{number} : \mathit{string}; \\
\quad \vdots \\
\quad \textbf{meth}\ \mathit{getStreet}\ \widehat{=}\ \textbf{res}\ r : \mathit{string} \bullet r := \mathit{street}\ \textbf{end} \\
\quad \textbf{meth}\ \mathit{setStreet}\ \widehat{=}\ \textbf{val}\ s : \mathit{string} \bullet \mathit{street} := s\ \textbf{end} \\
\quad \textbf{new}\ \widehat{=}\ \bullet\ \textbf{skip}\ \textbf{end} \\
\textbf{end}
\end{array}
$$

into two other classes: the *Client* class introduced in the previous section—which is similar to the class above but where direct access to the instance variables *street* and *number* are replaced by corresponding *set* and *get* method calls through the *add* attribute—and the following *Address* class:

$$
\begin{array}{l}
\textbf{class}\ \mathit{Address}\ \textbf{extends}\ \mathit{Object} \\
\quad \textbf{pri}\ \mathit{street} : \mathit{string};\ \ \textbf{pri}\ \mathit{number} : \mathit{string}; \\
\quad \vdots \\
\quad \textbf{meth}\ \mathit{getStreet}\ \widehat{=}\ \textbf{res}\ r : \mathit{string} \bullet r := \mathit{street}\ \textbf{end} \\
\quad \textbf{meth}\ \mathit{setStreet}\ \widehat{=}\ \textbf{val}\ s : \mathit{string} \bullet \mathit{street} := s\ \textbf{end} \\
\quad \textbf{new}\ \widehat{=}\ \bullet\ \textbf{skip}\ \textbf{end} \\
\textbf{end}
\end{array}
$$

In fact, this kind of restructuring is justified by the class-split law [3]. Here, instead of formalizing and proving that law, we simply sketch how it could be derived from the basic laws. We do that by illustrating how the suggested restructuring of the *Client* class above can be justified. We hope this gives an idea of how the law could be formally derived.

Considering an arbitrary view and set of class declarations, the first step in the derivation uses Law 2.2.1 to justify the introduction of the *add* attribute to *Client*, followed by a variant of Law 2.2.3 to initialize this attribute:

class *Client* **extends** *Object*
  pri *name* : *string*; **pri** *street* : *string*;
  pri *number* : *string*; **pri** *add* : *Address*;
  ⋮
  meth *getStreet* $\widehat{=}$ **res** *r* : *string* • *r* := *street* **end**
  meth *setStreet* $\widehat{=}$ **val** *s* : *string* • *street* := *s* **end**
  new $\widehat{=}$ • *add* := *new Address*() **end**
end

Then, by using a variant of Law 2.2.3 for method calls, and Law 2.2.5 for establishing the invariant $add \neq null \wedge add.street = street$, we obtain the following class:

class *Client* **extends** *Object*
  pri *name* : *string*; **pri** *street* : *string*;
  pri *number* : *string*; **pri** *add* : *Address*;
  ⋮
  meth *getStreet* $\widehat{=}$ **res** *r* : *string* •
    {*add.street* = *street*} *r* := *street*
  end
  meth *setStreet* $\widehat{=}$ **val** *s* : *string* •
    street := *s*; *add.setStreet*(*s*)
  end
  new $\widehat{=}$ • *add* := *new Address*() **end**
end

Finally, by using the law for method calls (Law 2.1.5) from right to left, followed by laws 2.2.3 and 2.2.1, we have

class *Client* **extends** *Object*
  pri *name* : *string*;
  pri *number* : *string*; **pri** *add* : *Address*;
  ⋮
  meth *getStreet* $\widehat{=}$ **res** *r* : *string* • *add.getStreet*(*r*) **end**
  meth *setStreet* $\widehat{=}$ **val** *s* : *string* • *add.setStreet*(*s*) **end**
  new $\widehat{=}$ • *add* := *new Address*() **end**
end

The desired result could be obtained by repeating the steps above for eliminating the attribute *number* and modifying its associated methods.

4 Soundness of the Laws

We have argued so far that algebraic laws can act like an interface for several practical applications. In the previous section we have briefly illustrated how they can be used to derive more elaborate laws which can be useful, for example, for developing (or restructuring) object-oriented programs. But what about the soundness of this interface (the very basic laws which cannot be derived from others).

The approach followed in the CO-OP project is to verify the basic algebraic laws of ROOL against an independent predicate transformer semantics for the language [6]. This has actually been carried out [7] for all the basic laws presented in [2]. Rather than presenting such proofs, our purpose here is to single out the importance of linking the algebraic laws to a mathematical model in which they can be verified. This is essential to avoid postulating invalid laws, especially when they are intuitively believed to be true, as exemplified in the remainder of this section.

4.1 A Surprising Result

Proving laws might bring surprises. The well-known statement

> One can replace an object of a class C with an object of a (behavioral) subclass C' of C

(which is widely taken as a general law of the object-oriented paradigm) is not true in general. In particular, it is not generally valid in languages such as Java.

In the several attempts to formalise the above statement (given below) we use a refinement relation. For commands c_1 and c_2, $c_1 \sqsubseteq c_2$ has the usual meaning that c_2 satisfies every specification satisfied by c_1. Therefore, substitution of c_2 for c_1 in any context is an improvement (or at least will leave things unchanged when $c_1 = c_2$). See [6] for a formal definition of refinement for ROOL.

Our first attempt is captured by the following proposition.

Proposition 4.1 $x := \mathbf{new}\ C \ \sqsubseteq\ x : \mathbf{new}\ C'$

At first glance this captures the statement quoted above. After more detailed inspection, however, we realized that this is not valid in contexts which include type tests or casts on x, since the condition x **is** C' is satisfied after the assignment on the right-hand side of the above inequation, but is false after the assignment on the left-hand side.

This has led us to suggest a more specific context where the substitution could be safely carried out, making x local to the context of the transformation, and requiring that x could not be type tested or cast in this scope.

Proposition 4.2 If p has no type tests or casts on x, *then*
$\mathbf{var}\ x : C \bullet x := \mathbf{new}\, C;\ \ p\ \mathbf{end}\ \sqsubseteq\ \mathbf{var}\ x : C \bullet x := \mathbf{new}\, C';\ \ p\ \mathbf{end}$

Nevertheless, the above condition is still not strong enough to ensure a correct transformation in general. The problem is that x could be assigned to a global variable in the scope p (or passed as parameter in a method call) and therefore this would allow the object assigned to it to be type tested or cast outside the scope of the local block.

The proposition below incorporates the necessary conditions to ensure a safe transformation.

Proposition 4.3 If p has no type tests or casts on x, and x is not passed as parameter in method calls nor is assigned to variables of p, then
$\mathbf{var}\ x : C \bullet x := \mathbf{new}\, C;\ \ p\ \mathbf{end}\ \sqsubseteq\ \mathbf{var}\ x : C \bullet x := \mathbf{new}\, C';\ \ p\ \mathbf{end}$

This is now consistent and follows from the definition of refinement presented in [6].

5 Conclusions and Future Work

This paper has illustrated how an object-oriented language (ROOL) can be defined as an algebraic structure whose axioms are basic laws which characterise the semantics of the language. But 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, as discussed in the previous section.

Once the laws have been proved, in whatever model, they should serve as tools for carrying out program transformation. One immediate application of the basic laws is to serve as an interface from which one can derive more elaborate transformation strategies which justify programming practices. This was illustrated in Section 3, where we have shown how a design transformation for object-oriented development can be formalised and applied. Formalising further transformations, and applying them to realistic case studies, is one of the topics for further research, as well as developing rules to compile the executable subset of ROOL into a Java Virtual Machine.

Although we have already discovered a comprehensive set of basic laws [2] for ROOL, which have been proved [7] based on a weakest pre-condition semantics for ROOL [6], we still need to prove a reduction theorem capturing that our set of laws is complete in the sense of allowing reduction of arbitrary ROOL programs to a normal form expressed in a small subset of the language operators.

In the literature related to formalising object-oriented development methods, one can find several approachs, for example [13, 14], to extend refinement calculi (originally conceived for imperative languages) to object-oriented languages. In terms of object-oriented features, these works impose strong restrictions to the language, not dealing, for example, with classes or restricting inheritance. Furthermore, these approaches concentrate only on laws for deriving programs from specifications. Our purpose here (and in the CO-OP project at large) is to develop a refinement algebra for ROOL which must be complete in the sense explained above. We are not aware of any other work in this direction.

Acknowledgements

We thank our collaborators Ana Cavalcanti and David Naumann for the so many discussions which contributed to the research reported in this paper. In particular, the further conditions imposed on Proposition 2 to obtain Proposition 3 have been pointed out by them.

References

[1] Scott Ambler. *Building Object Applications that Work.* Cambridge University Press and Sigs Books, 1998.

[2] P. Borba and A. Sampaio. Basic laws of rool. Technical report, Centro de Informática, Universidade Federal de Pernambuco, Brazil, 2000.

[3] Paulo Borba. Where are the laws of object-oriented programming? In *I Brazilian Workshop on Formal Methods*, pages 59–70, Porto Alegre, Brazil, 19th–21st October 1998.

[4] Paulo Borba and Joseph Goguen. Refinement of concurrent object oriented programs. In Stephen Goldsack and Stuart Kent, editors, *Formal Methods and Object Technology,* Chapter 11. Springer-Verlag, 1996. Also appeared as Technical Report TR-18-95, Oxford University, Computing Laboratory, Programming Research Group, November 1995.

[5] William Brown et al. *Anti Patterns: Refactoring Software, Architectures and Projects in Crisis.* Wiley Computer Publishing, 1998.

[6] A. Cavalcanti and D. Naumann. A weakest precondition semantics for refinement of object-oriented programs. *IEEE Transactions on Software Engineering*, 2000. To appear.

[7] M. Cornelio and A. Cavalcanti. Proving the basic laws of rool in a weakest precondition semantics. Technical report, Centro de Informática, Universidade Federal de Pernambuco, Brazil, 2000.

[8] E. W. Dijkstra. *A Discipline of Programming.* Prentice-Hall, Englewood Cliffs, 1976.

[9] Erich Gamma et al. *Design Patterns: Elements of Reusable Object-Oriented Software.* Addison-Wesley, 1994.

[10] James Gosling, Bill Joy, and Guy Steele. *The Java Language Specification.* Addison-Wesley, 1996.

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

[12] Doug Lea. *Concurrent Programming in Java.* Addison-Wesley, 1997.

[13] K. R. M. Leino. Recursive Object Types in a Logic of Object-oriented Programming. In C. Hankin, editor, *7th European Symposium on Programming*, volume 1381 of *Lecture Notes in Computer Science*. Springer-Verlag, 1998.

[14] A. Mikhajlova and E. Sekerinsk. Fme'97: Industrial applications and strengthened foundations of formal methods. In *Proceedings of FME'97*, volume 1313 of *Lecture Notes in Computer Science*, pages 82–101. Springer-Verlag, 1997.

[15] Carroll Morgan. *Programming from Specifications.* Prentice Hall, second edition, 1994.

[16] William Opdyke. *Refactoring Object-Oriented Frameworks.* PhD thesis, University of Illinois at Urbana-Champaign, 1992.

[17] A. Roscoe and C. A. R. Hoare. The laws of occam programming. *Theoretical Computer Science*, 60:177–229, 1988.

[18] Augusto Sampaio. *An Algebraic Approach to Compiler Design*, volume 4 of *Algebraic Methodology and Software Technology.* World Scientific, 1997.

[19] L. Silva, A. Sampaio, and E. Barros. A normal form reduction strategy for hardware/software partitioning. In *Proceedings of Formal Methods Europe (FME) 97*, volume 1313 of *Lecture Notes in Computer Science*, pages 624–643. Springer-Verlag, 1997.