# Basic Laws of ROOL: an Object-Oriented Language


## Paulo Borba and Augusto Sampaio*



### Abstract

In this article 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 formalize 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.



Keywords: Refinement algebra, refinement calculus, object-oriented programming.



*Centro de Informática
Universidade Federal de Pernambuco
e-mail: {phmb,acas}@cin.ufpe.br

## 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 [H*et al*87, Mor94]. 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 optimizers [Sam97].

Contrasting, the laws of object-oriented programming are not well-established yet [Bor98]. Some laws have been informally discussed in the object-oriented literature [Lea97, Opd92, Fow99, Amb98], 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 [Lei98, MS97, Nau00b, Nau00a] has considered some 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) [CN99], which is based on Java [GJS96] but has a copy semantics rather than a reference semantics—the copy semantics significantly simplifies reasoning and still allows us to consider Java programs that do not have reference aliasing. 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 serve as a basis for deriving more elaborate laws for practical applications of program transformation. 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 [H*et al*87, RH88]. This, however, is beyond the scope of this article.

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 [CN99] 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 design law for safely introducing a design pattern, assisting the transition from anti-patterns to patterns [$G^+94$, $B^+98$]. The compilation laws will be used to compile the executable subset of ROOL into the Java Virtual Machine, ensuring correctness of the translation by construction.

This article is organized 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 summarize the results achieved so far.

## 2. ROOL

ROOL is an object-oriented language based on Java, but with a copy semantics rather than a reference semantics. It has been specially designed to allow reasoning about object-oriented programs and specifications, hence it mixes both kinds of constructs in the style of Morgan's refinement calculus [Mor94]. A program in ROOL consists of a *main* command and a set of class declarations. Classes are declared as in the following example:

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

where 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 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 "$\bullet$". Initializers are declared by the **new** clause and do not have parameters.

In addition to method calls, as illustrated in the *Client* class, the body of methods and initializers 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:

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

where a specification statement $x : [pre,\ post]$ 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 ($pre$) terminates in a state satisfying its postcondition ($post$). Like the languages adopted in other refinement calculi, ROOL is a specification language where programs appear as an executable subset of specifications.

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:

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

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

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

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 (in fact, **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 establish a condition $b$ at a given point in the program text. This can be characterised as a *coercion* to $b$, designated as $[b]$, defined as follows.

$$[b] \;=\; : [\textbf{true}, b]$$

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

$$\textbf{skip} \;=\; : [\textbf{true}, \textbf{true}]$$

The weakest possible characterisation of a program that always terminates is given by $x : [\textbf{true}, \textbf{true}]$. Note that it is similar to **skip**, but unlike **skip** it is allowed to assign to variables $x$. Any program which terminates successfully must refine such a specification pattern, instantiating $x$ with the program globlal variables. Further considerations about specification statements can be found, for example, in [Mor94].

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

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

where **self** 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 $e_2$. So, despite the 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. Expressions such as $\mathbf{null}.x$ and $(\mathbf{null};\ x : e)$ cannot be successfully evaluated; they yield **error**, a special value that can only be used in predicates.

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 based on weakest preconditions are given elsewhere [CN99]. Here we introduce the basic 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, [H*et al*87, RH88]. 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 [Dij76], 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 well known *copy rule* (for procedure call elimination), 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 behavior 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** $\langle$**if** symmetry$\rangle$ If $i$ ranges over $1..n$ and $\pi$ 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** $\langle$**if** true guard$\rangle$

$$\mathbf{if\ true} \rightarrow c\ \mathbf{fi} \ = \ c$$

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** $\langle$**if** false unity$\rangle$
$\mathbf{if\ false} \rightarrow c \; [] \; gcs \; \mathbf{fi} \;=\; \mathbf{if} \; gcs \; \mathbf{fi}$

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

**Law 2.1.4** $\langle := - \lhd\rhd$ right dist$\rangle$
$le := e; \; \mathbf{if} \; []i \bullet \psi_i \rightarrow c_i \mathbf{fi} \;=\; \mathbf{if} \; []i \bullet \psi_i[e/le] \rightarrow (le := e; \; c_i)\mathbf{fi}$

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

Some laws relating expressions are stated using coercions. So first it is important to recall that a coercion of $b$ behaves like **skip** if $b$ is **true** and like **miracle** otherwise.

**Law 2.1.5** $\langle$coercion **skip**$\rangle$
$[\mathbf{true}] \;=\; \mathbf{skip}$

**Law 2.1.6** $\langle$coercion **miracle**$\rangle$
$[\mathbf{false}] \;=\; \mathbf{miracle}$

When a coercion reduces to **miracle**, the entire program will behave miraculously.

**Law 2.1.7** $\langle$miracle left zero$\rangle$
$\mathbf{miracle}; \; c \;=\; \mathbf{miracle}$

If two expressions hold the same value at a given point in the program text, it is impossible to distinghish between assigning one or the other to a left expression.

**Law 2.1.8** $\langle$expression substitution$\rangle$
$([e = f]; \; le := e) \;=\; ([e = f]; \; le := f)$

When an expression that yields **error** is assigned to a variable, the whole assignment behaves like **abort**. Therefore, evaluation of a variable can never yield **error**, since **error** is not stored.

**Law 2.1.9** $\langle$variable well defined$\rangle$
$[x \neq \mathbf{error}] \;=\; \mathbf{skip}$

An attribute update is successful provided the relevant object is not **null** and the expression to be assigned is not **error**.

**Law 2.1.10** $\langle$update well $-$ defined$\rangle$ If $o \neq \mathbf{null}$ and $e \neq \mathbf{error}$, then
$[(o; \; x : e) \neq \mathbf{error}] \;=\; \mathbf{skip}$

Object creation can never yield a **null** object.

**Law 2.1.11** $\langle$object creation$\rangle$

$[\mathbf{new}\ C \neq \mathbf{null}] \ = \ \mathbf{skip}$

Some of the laws involving object-oriented features of ROOL rely on the context in which they will be applied, unlike the previous laws which are valid in any context. We use the general form below to present the laws of these operators:

$cds, C \rhd c = c'$ provided *conditions*

meaning that the equation $c = c'$ holds in the context of class declarations *cds* provided the *conditions* are satisfied. Furthermore, the command $c$ is assumed to be inside the class with name $C$ (whose declaration is in *cds*); that is, the equation can only be applied inside $C$.

The *conditions* are always syntactic, usually related to type checking. For example, $cds, B \rhd e : C$ requires that the expression $e$ appearing in class $B$ have type $C$. Some conditions are expressed using the notation $C' \leq_{cds} C$ which holds if $C'$ is $C$ itself or a subclass of $C$ in the context of *cds*.

Furthermore, some conditions (and some of the laws of ROOL) are expressed in terms of 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 [CN99] for a formal definition of refinement for ROOL.

As an example, the following law states that **new** $C$ is not **error** provided the body of **new** terminates sucessfully.

**Law 2.1.12** $\langle$object creation well $-$ defined$\rangle$ If $c$ is the body of **new** of class $C$, $x$ is the list of global variables of $c$, and $x : [\mathbf{true}, \mathbf{true}] \sqsubseteq c$, then

$[\mathbf{new}\ C \neq \mathbf{error}] \ = \ \mathbf{skip}$

Recall that $x : [\mathbf{true}, \mathbf{true}]$ is the weakest possible specification of a program which terminates. Therefore $x : [\mathbf{true}, \mathbf{true}] \sqsubseteq c$ captures the desired condition that $c$ must terminate.

The most elaborate law for commands is the one for method call elimination, since, due to dynamic binding, the application of this law 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* in the sense of replacing the call by the body of the method; however, 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 2.1.13** $\langle$*method call* elimination$\rangle$ If $cds, A \rhd le : C$, $C' \leq_{cds} C$, $A$ is not $C$ nor $C'$, and all attributes which appear in the body of $m$ are public, then

$$cds, A \rhd le.m() \ = \ \begin{array}{ll} \mathbf{if} & (le\ \mathbf{is}\ C') \rightarrow C'.m[le/\mathbf{self}] \\ [] & (le\ \mathbf{is}\ C) \wedge \neg\,(le\ \mathbf{is}\ C') \rightarrow C.m[le/\mathbf{self}] \\ \mathbf{fi} & \end{array}$$

The notation $X.m[le/\mathbf{self}]$ stands for the body of the method $m$ in class $X$, after replacing every occurrence of **self** with $le$. Although the condition that every attribute referenced by $m$ be public might seem very strong, observe that it is necessary, otherwise the replacement of the method call by its body would cause a compilation error, since this attribute would not be visible in context $A$. We also assume that every reference to a method $m$ and to an attribute $a$, declared in the class itself, has the form $\mathbf{self}.m()$ and $\mathbf{self}.a$, respectively; this allows the replacement of **self** by $le$ in an uniform way.

The generalization of this law for contexts involving arbitrary inheritance hierarchy can be found in [BS00], 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

$$cds_1 =_{cds} cds_2$$

to indicate that the sets of class declarations $cds_1$ and $cds_2$ are equivalent in contexts having the class declarations in $cds$, which typically contains the definition of "auxiliary" classes for $cds_1$ and $cds_2$. In fact, the equations that we consider here hold only for valid class declarations in the context defined by $cds$; that is, both '$cds\ cds_1$' and '$cds\ cds_2$' should be valid declarations, where the juxtaposition of sets of class declarations denote their union.

In general, it might be useful to establish that the equivalence of class declarations holds just for contexts that use only a subset of the methods and attributes of the related declarations [Bor98, BG96]. 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 $cds_2$. We consider that any command $c$ built by using the methods and attributes from $v$ must be valid in the contexts defined by both '$cds\ cds_1$' and '$cds\ cds_2$'. Moreover, sometimes we omit a subscript if the equivalence holds for arbitrary values of that subscript.

#### Attributes

The basic laws of classes state simple properties about classes and their relation with attributes, methods, and invariants. We first focus on attributes. For instance, the following law states that we can introduce a "fresh" private attribute or remove a private attribute that is not used. We assume that $C.a$ refers to the name $a$ declared by class $C$.

**Law 2.2.1** ⟨introduce private attribute⟩
Let $a$ be an attribute name that is not in $ads$ and is not declared by a superclass or subclass of $C$ in $cds$. Let $T$ be a primitive data type, $C$, or a class name declared in $cds$. Then, if $C.a$ does not appear in $ops$, we have

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

The notation '**pri** $a : T;\ ads$' denotes the set of attribute declarations containing '**pri** $a : T$' and all the declarations in $ads$, whereas $ops$ stands for the declarations of operations (object initializer and methods).

Another law indicates that a protected attribute can be turned into a private one, and vice-versa, provided that the attribute is not directly used by subclasses of its associated class. This, however, is only valid for contexts that do not refer to that attribute, otherwise we could write programs that would be valid considering the class declaration on the left of the equality symbol below, but invalid considering the declaration on the right.

**Law 2.2.2** ⟨hide protected attribute⟩
If $C.a$ is not in $v$ and does not appear in any class of $cds$ then

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

There is also a similar law for relating protected and public attributes.

**Law 2.2.3** ⟨protect public attribute⟩
If $C.a$ is not in $v$ and appears only in $ops$ and in the subclasses of $C$ in $cds$ then

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

As a consequence of those two laws, by transitivity, we can derive a law that relates private and public attributes.

**Law 2.2.4** ⟨hide public attribute⟩
If $C.a$ is not in $v$ and does not appear in any class of $cds$ then

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

The semantics of attributes could be further specified by stating that an assignment to a private attribute is useless if the attribute is never read and the expression on the assignment can be successfully evaluated. In fact, instead of requiring the attribute not to be read, we can simply require it to be *auxiliary*—an attribute $a$ is auxiliary if the expression $a$ appears only in result arguments or in assignments and postconditions that change only $a$ [Mor94]. In this way, we allow the value of $a$ to be read as long as it is only used to set a new value to $a$. We express this law by an equivalence of commands valid only inside the class that declares the attribute.

**Law 2.2.5** ⟨useless assignment⟩
Let $CD$ be a class declaration such as the following:

$$\begin{array}{l} \textbf{class } C \textbf{ extends } D \\ \quad \textbf{pri } a : T;\ ads \\ \quad ops \\ \textbf{end} \end{array}$$

Then, if $a$ is auxiliary in $CD$, we have

$$cds\ CD, C \rhd [exp \neq \textbf{error}];\ a := exp \ = \ [exp \neq \textbf{error}];\ \textbf{skip}$$

for any expression $exp$.

Recall that the command $[exp \neq \textbf{error}]$ is an abbreviation for : $[\textbf{true}, exp \neq \textbf{error}]$, which, by Laws 2.1.5 and 2.1.6, behaves as **skip** when $exp$ can be successfully evaluated, and otherwise as **miracle**. Law 2.1.7 then explains the equality above when the evaluation of $exp$ yields **error**.

Since the assignment "$a.x := exp$" is equivalent to the assignment

$$a := (a;\ x : exp),$$

note that there is no need to generalize the previous law to consider more complex left expressions on the eliminated assignment.

### Methods

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.

**Law 2.2.6** ⟨introduce method⟩
Let $m$ be a method name that is not in $v$ and is not declared by a superclass or subclass of $C$ in $cds$. Then, if $C.m$ does not appear in $mds$, $id$, and $cds$ we have

$$
\boxed{\begin{array}{l}
\textbf{class}\ C\ \textbf{extends}\ D\\
\quad ads\\
\quad mds\\
\quad id\\
\textbf{end}
\end{array}}
\quad =_{v,cds} \quad
\boxed{\begin{array}{l}
\textbf{class}\ C\ \textbf{extends}\ D\\
\quad ads\\
\quad \textbf{meth}\ m\ \mathrel{\widehat=}\ pds\ \bullet\ c\\
\quad \textbf{end};\ mds\\
\quad id\\
\textbf{end}
\end{array}}
$$

**Class Invariants**

Another 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 beginning of the execution of any method of the corresponding class.

**Law 2.2.7** ⟨introduce class invariant first⟩
Let *inv* be an invariant of class $C$. Then

$$
\boxed{\begin{array}{l}
\textbf{class}\ C\ \textbf{extends}\ D\\
\quad ads\\
\quad \textbf{meth}\ m\ \mathrel{\widehat=}\ pds\ \bullet\ c\\
\quad \textbf{end};\ mds\\
\quad id\\
\textbf{end}
\end{array}}
\quad =_{v,cds} \quad
\boxed{\begin{array}{l}
\textbf{class}\ C\ \textbf{extends}\ D\\
\quad ads\\
\quad \textbf{meth}\ m\ \mathrel{\widehat=}\ pds\ \bullet\ [inv];\ c\\
\quad \textbf{end};\ mds\\
\quad id\\
\textbf{end}
\end{array}}
$$

There is a similar law stating that a class invariant is valid at the end of the execution of methods as well.

**Law 2.2.8** ⟨class invariant after⟩
Let *inv* be an invariant of class $C$. Then

$$
\boxed{\begin{array}{l}
\textbf{class}\ C\ \textbf{extends}\ D\\
\quad ads\\
\quad \textbf{meth}\ m\ \mathrel{\widehat=}\ pds\ \bullet\\
\quad\quad c\\
\quad \textbf{end};\ mds\\
\quad id\\
\textbf{end}
\end{array}}
\quad =_{v,cds} \quad
\boxed{\begin{array}{l}
\textbf{class}\ C\ \textbf{extends}\ D\\
\quad ads\\
\quad \textbf{meth}\ m\ \mathrel{\widehat=}\ pds\ \bullet\\
\quad\quad c;\ [inv]\\
\quad \textbf{end};\ mds\\
\quad id\\
\textbf{end}
\end{array}}
$$

## 3. Applications

Algebraic laws in the style presented in the previous section have proved useful for several applications such as proving properties about programs [RH88], designing compilers [Sam97], and partitioning algorithms [SSB97], all correct by construction. Such laws are useful for program or specification transformation in general. To give a simple idea of the supported 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.

$$
\begin{array}{ll}
 & \mathbf{var}\ x : C \bullet x := \mathbf{new}\ C';\ \begin{array}{ll} \mathbf{if} & (x\ \mathbf{is}\ C') \to c_1 \\ [\,] & \neg\,(x\ \mathbf{is}\ C') \to c_2 \\ \mathbf{fi} & \end{array}\ \mathbf{end} \\
= & \\
 & \mathbf{var}\ x : C \bullet x := \mathbf{new}\ C';\ c_1\ \mathbf{end}
\end{array}
$$

Using the algebraic laws of commands presented in the previous section, we can formally derive the above equation; for simplicity, we implicitly use expression laws in order to establish that **new** $C$ **is** $C \equiv$ **true** assuming that **new** $C$ in this case is not **error**.

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

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 interesting and elaborate example which illustrates the use of the laws of commands and classes.

### 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 useful for software evolution, when programmers are interested on restructuring code in order to achieve modularity and, consequently, improve software reuse and extensibility [Fow99].

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

```
class Client
  pri name : string;
  pri street : string; pri number : string; ...
  meth getStreet ≙ res r : string • r := street end;
  meth setStreet ≙ val s : string • street := s end;
  new ≙ skip end
end
```

into two other classes: the *Client* class introduced in Section 2.—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 an *Address* class containing address information and their associated operations.

In fact, this kind of restructuring is justified by the class-split law [Bor98]. 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.

Formally, we are interested in sketching the proof that the set of class declarations formed by the following classes:

$$
\begin{array}{l}
\mathbf{class}\ \mathit{Client} \\
\quad \mathbf{pri}\ \mathit{name} : \mathit{string}; \\
\quad \mathbf{pri}\ \mathit{street} : \mathit{string};\ \mathbf{pri}\ \mathit{number} : \mathit{string};\ \ldots \\
\quad \mathbf{meth}\ \mathit{getStreet}\ \widehat{=}\ \mathbf{res}\ r : \mathit{string}\ \bullet\ r := \mathit{street}\ \mathbf{end}; \\
\quad \mathbf{meth}\ \mathit{setStreet}\ \widehat{=}\ \mathbf{val}\ s : \mathit{string}\ \bullet\ \mathit{street} := s\ \mathbf{end}; \\
\quad \mathbf{new}\ \widehat{=}\ \mathbf{skip}\ \mathbf{end} \\
\mathbf{end} \\
\mathbf{class}\ \mathit{Address} \\
\quad \mathbf{pri}\ \mathit{street} : \mathit{string};\ \ \mathbf{pri}\ \mathit{number} : \mathit{string};\ \ldots \\
\quad \mathbf{meth}\ \mathit{getStreet}\ \widehat{=}\ \mathbf{res}\ r : \mathit{string}\ \bullet\ r := \mathbf{self}.\mathit{street}\ \mathbf{end}; \\
\quad \mathbf{meth}\ \mathit{setStreet}\ \widehat{=}\ \mathbf{val}\ s : \mathit{string}\ \bullet\ \mathbf{self}.\mathit{street} := s\ \mathbf{end}; \\
\quad \mathbf{new}\ \widehat{=}\ \mathbf{skip}\ \mathbf{end} \\
\mathbf{end}
\end{array}
$$

is equivalent to the set of class declarations formed by the classes bellow:

$$
\begin{array}{l}
\mathbf{class}\ \mathit{Client} \\
\quad \mathbf{pri}\ \mathit{name} : \mathit{string};\ \ \mathbf{pri}\ \mathit{add} : \mathit{Address};\ \ldots \\
\quad \mathbf{meth}\ \mathit{getStreet}\ \widehat{=}\ \mathbf{res}\ r : \mathit{string}\ \bullet\ \mathit{add}.\mathit{getStreet}(r)\ \mathbf{end}; \\
\quad \mathbf{meth}\ \mathit{setStreet}\ \widehat{=}\ \mathbf{val}\ s : \mathit{string}\ \bullet\ \mathit{add}.\mathit{setStreet}(s)\ \mathbf{end}; \\
\quad \mathbf{new}\ \widehat{=}\ \mathit{add} := \mathbf{new}\ \mathit{Address}\ \mathbf{end}; \\
\mathbf{end} \\
\mathbf{class}\ \mathit{Address} \\
\quad \mathbf{pri}\ \mathit{street} : \mathit{string};\ \ \mathbf{pri}\ \mathit{number} : \mathit{string};\ \ldots \\
\quad \mathbf{meth}\ \mathit{getStreet}\ \widehat{=}\ \mathbf{res}\ r : \mathit{string}\ \bullet\ r := \mathbf{self}.\mathit{street}\ \mathbf{end}; \\
\quad \mathbf{meth}\ \mathit{setStreet}\ \widehat{=}\ \mathbf{val}\ s : \mathit{string}\ \bullet\ \mathbf{self}.\mathit{street} := s\ \mathbf{end}; \\
\quad \mathbf{new}\ \widehat{=}\ \mathbf{skip}\ \mathbf{end} \\
\mathbf{end}
\end{array}
$$

considering a specific view $v$ and context *cds* of class declarations. In fact, the derivation follows considering an arbitrary set of class declarations *cds* not containing *Client* and *Address*, and any view $v$ not including the methods and attributes of the *Address* class. We assume that *add* is an attribute name not declared in the subclasses of *Client* that appear in *cds*. The initial derivation steps use Law 2.2.1 to justify the introduction of the *add* attribute to *Client*, followed by Law 2.2.5 to initialize this attribute. The resulting class is shown below, where the parts of the code that were modified are underlined:

$$
\begin{array}{l}
\textbf{class}\ \ \mathit{Client} \\
\quad \textbf{pri}\ \ \mathit{name} : \mathit{string};\ \underline{\textbf{pri}\ \ \mathit{add} : \mathit{Address}}; \\
\quad \textbf{pri}\ \ \mathit{street} : \mathit{string};\ \textbf{pri}\ \ \mathit{number} : \mathit{string};\ \ldots \\
\quad \textbf{meth}\ \ \mathit{getStreet}\ \ \widehat{=}\ \ \textbf{res}\ \ r : \mathit{string}\ \ \bullet\ \ r := \mathit{street}\ \ \textbf{end}; \\
\quad \textbf{meth}\ \ \mathit{setStreet}\ \ \widehat{=}\ \ \textbf{val}\ \ s : \mathit{string}\ \ \bullet\ \ \mathit{street} := s\ \ \textbf{end}; \\
\quad \textbf{new}\ \ \widehat{=}\ \ \underline{\mathit{add} := \textbf{new}\ \ \mathit{Address}}\ \ \textbf{end} \\
\textbf{end}
\end{array}
$$

For simplicity, we omit the *Address* class from the derivation steps and later assume that its attributes can be made public, and then private again, by using Law 2.2.4. Moreover, we do not always mention the use of command laws, which are, for example, necessary to establish that

$$[\textbf{new}\,\mathit{Address} \neq \textbf{error}] = \textbf{skip}$$

and then to eliminate **skip**.

From this class, by using Law 2.2.7 for establishing the invariant "$\mathit{add} \neq \textbf{null}$", and Law 2.2.5 for introducing an useless assigment, we obtain the following class:

$$
\begin{array}{l}
\textbf{class}\ \ \mathit{Client} \\
\quad \textbf{pri}\ \ \mathit{name} : \mathit{string};\ \textbf{pri}\ \ \mathit{add} : \mathit{Address}; \\
\quad \textbf{pri}\ \ \mathit{street} : \mathit{string};\ \textbf{pri}\ \ \mathit{number} : \mathit{string};\ \ldots \\
\quad \textbf{meth}\ \ \mathit{getStreet}\ \ \widehat{=}\ \ \textbf{res}\ \ r : \mathit{string}\ \ \bullet\ \ r := \mathit{street}\ \ \textbf{end}; \\
\quad \textbf{meth}\ \ \mathit{setStreet}\ \ \widehat{=}\ \ \textbf{val}\ \ s : \mathit{string}\ \ \bullet \\
\quad\quad \mathit{street} := s;\ \underline{\mathit{add}.\mathit{street} := s} \\
\quad \textbf{end}; \\
\quad \textbf{new}\ \ \widehat{=}\ \ \mathit{add} := \textbf{new}\ \ \mathit{Address}\ \ \textbf{end} \\
\textbf{end}
\end{array}
$$

Note that $\mathit{add}.\mathit{street} := s$ is actually equivalent to $\mathit{add} := (\mathit{add};\ \mathit{street} : s)$, and several command laws can be used to prove that $(\mathit{add};\ \mathit{street} : s)$ does not yield **error**. Similar steps should be carried out for updating the other attributes of *Address*; they must store the same values stored by the corresponding attributes of *Client*.

We can now use Law 2.2.7 again for establishing the invariant "$\mathit{add}.\mathit{street} = \mathit{street}$" and then change the assignment to $r$:

$$
\begin{array}{l}
\mathbf{class}\ \mathit{Client} \\
\quad \mathbf{pri}\ \mathit{name} : \mathit{string};\ \mathbf{pri}\ \mathit{add} : \mathit{Address}; \\
\quad \mathbf{pri}\ \mathit{street} : \mathit{string};\ \mathbf{pri}\ \mathit{number} : \mathit{string};\ \ldots \\
\quad \mathbf{meth}\ \mathit{getStreet}\ \widehat{=}\ \mathbf{res}\ r : \mathit{string} \bullet \\
\qquad \underline{r := \mathit{add.street}} \\
\quad \mathbf{end}; \\
\quad \mathbf{meth}\ \mathit{setStreet}\ \widehat{=}\ \mathbf{val}\ s : \mathit{string} \bullet \\
\qquad \mathit{street} := s;\ \mathit{add.street} := s \\
\quad \mathbf{end}; \\
\quad \mathbf{new}\ \widehat{=}\ \mathit{add} := \mathbf{new}\ \mathit{Address}\ \mathbf{end} \\
\mathbf{end}
\end{array}
$$

By using twice a variation of the law for method calls (Law 2.1.13) from right to left, we introduce two method calls to *add*:

$$
\begin{array}{l}
\mathbf{class}\ \mathit{Client} \\
\quad \mathbf{pri}\ \mathit{name} : \mathit{string};\ \mathbf{pri}\ \mathit{add} : \mathit{Address}; \\
\quad \mathbf{pri}\ \mathit{street} : \mathit{string};\ \mathbf{pri}\ \mathit{number} : \mathit{string};\ \ldots \\
\quad \mathbf{meth}\ \mathit{getStreet}\ \widehat{=}\ \mathbf{res}\ r : \mathit{string} \bullet \underline{\mathit{add.getStreet}(r)}\ \mathbf{end}; \\
\quad \mathbf{meth}\ \mathit{setStreet}\ \widehat{=}\ \mathbf{val}\ s : \mathit{string} \bullet \\
\qquad \mathit{street} := s;\ \underline{\mathit{add.setStreet}(s)} \\
\quad \mathbf{end}; \\
\quad \mathbf{new}\ \widehat{=}\ \mathit{add} := \mathbf{new}\ \mathit{Address}\ \mathbf{end} \\
\mathbf{end}
\end{array}
$$

Finally, by using Law 2.2.5 to eliminate the assignment to *street* and then Law 2.2.1 to eliminate this attribute, we arrive at

$$
\begin{array}{l}
\mathbf{class}\ \mathit{Client} \\
\quad \mathbf{pri}\ \mathit{name} : \mathit{string};\ \mathbf{pri}\ \mathit{add} : \mathit{Address}; \\
\quad \underline{\mathbf{pri}\ \mathit{number} : \mathit{string};\ \ldots} \\
\quad \mathbf{meth}\ \mathit{getStreet}\ \widehat{=}\ \mathbf{res}\ r : \mathit{string} \bullet \mathit{add.getStreet}(r)\ \mathbf{end}; \\
\quad \mathbf{meth}\ \mathit{setStreet}\ \widehat{=}\ \mathbf{val}\ s : \mathit{string} \bullet \underline{\mathit{add.setStreet}(s)}\ \mathbf{end}; \\
\quad \mathbf{new}\ \widehat{=}\ \mathit{add} := \mathbf{new}\ \mathit{Address}\ \mathbf{end} \\
\mathbf{end}
\end{array}
$$

The desired result could be obtained by repeating the steps above for eliminating the other attributes of *Client* that have corresponding ones in *Address*.

## 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 [CN99]. This has actually been carried out [CC00] for all the basic command laws presented in [BS00]. 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.

Our first attempt is captured by the following proposition.

**Proposition 4..1** $x := \textbf{new}\ C \ \sqsubseteq\ x := \textbf{new}\ C'$

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 $x\ \textbf{is}\ C'$ 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 casted in this scope.

**Proposition 4..2** If $p$ has no type tests or casts on $x$, then
$\textbf{var}\ x : C \bullet x := \textbf{new}\ C;\ \ p\ \textbf{end} \ \sqsubseteq\ \textbf{var}\ x : C \bullet x := \textbf{new}\ C';\ \ p\ \textbf{end}$

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 casted 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 [CN99].

## 5. Conclusions and Future Work

This article has illustrated how an object-oriented language (ROOL) can be defined as an algebraic structure whose axioms are basic laws which characterize 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 formalized and applied. Formalizing 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 [BS00] for ROOL, which have been proved [CC00] based on a weakest pre-condition semantics for ROOL [CN99], 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 formalizing object-oriented development methods, one can find several approachs, for example [Lei98, MS97], 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 command 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 also consider laws of both commands and classes. We are not aware of any other work in this direction.

## Acknowledgements

We thank our collaborators Ana Cavalcanti and David Naumann for many discussions which contributed to the research reported in this article. In particular, the further conditions imposed on Proposition 2 to obtain Proposition 3 have been pointed out by them. We would also like to thank the anonymous referees for the careful evaluation of this article.

The authors are partly supported by CNPq, grants 521994/96–9 (Paulo Borba), 521039/95–9 (Augusto Sampaio), and 680032/99-1 (CO-OP project, jointly funded by PROTEM-CC and the National Science Foundation).

## References

[Amb98] Scott Ambler. *Building Object Applications that Work.* Cambridge University Press and Sigs Books, 1998.

[B+98] William Brown et al. *Anti Patterns: Refactoring Software, Architectures and Projects in Crisis.* Wiley Computer Publishing, 1998.

[BG96] 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.

[Bor98] 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.

[BS00] Paulo Borba and Augusto Sampaio. The basic laws of ROOL. Technical report, Centro de Informática, Universidade Federal de Pernambuco, Brazil, to appear in `http://www.cin.ufpe.br/~lmf/coop/papers`, 2000.

[CC00] Márcio Cornélio and Ana Cavalcanti. Proving the basic laws of ROOL in a weakest precondition semantics. Technical report, Centro de Informática, Universidade Federal de Pernambuco, Brazil, to appear in `http://www.cin.ufpe.br/~lmf/coop/papers`, 2000.

[CN99] Ana Cavalcanti and David Naumann. A weakest precondition semantics for an object-oriented language of refinement. In *FM'99 - Formal Methods*, volume 1709 of *Lecture Notes in Computer Science*, pages 1439–1459. Springer-Verlag, 1999.

[Dij76] E. W. Dijkstra. *A Discipline of Programming.* Prentice-Hall, Englewood Cliffs, 1976.

[Fow99] Martin Fowler. *Refactoring—Improving the design of existing code.* Addison Wesley, 1999.

[G+94] Erich Gamma et al. *Design Patterns: Elements of Reusable Object-Oriented Software.* Addison-Wesley, 1994.

[GJS96] James Gosling, Bill Joy, and Guy Steele. *The Java Language Specification.* Addison-Wesley, 1996.

[H*et al*87] C. A. R. Hoare *et al*. Laws of programming. *Communications of the ACM*, 30(8):672–686, August 1987.

[Lea97] Doug Lea. *Concurrent Programming in Java.* Addison-Wesley, 1997.

[Lei98] 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.

[Mor94] Carroll Morgan. *Programming from Specifications.* Prentice Hall, second edition, 1994.

[MS97] A. Mikhajlova and E. Sekerinsk. Class refinement and interface refinement in object-oriented programs. In *Proceedings of FME'97*, volume 1313 of *Lecture Notes in Computer Science*, pages 82–101. Springer-Verlag, 1997.

[Nau00a] David Naumann. Predicate transformer semantics of a higher order imperative language with record subtypes. *Science of Computer Programming*, 2000. To appear.

[Nau00b] David Naumann. Soundness of data refinement for a higher order imperative language. *Theoretical Computer Science*, 2000. To appear.

[Opd92] William Opdyke. *Refactoring Object-Oriented Frameworks.* PhD thesis, University of Illinois at Urbana-Champaign, 1992.

[RH88] A. Roscoe and C. A. R. Hoare. The laws of occam programming. *Theoretical Computer Science*, 60:177–229, 1988.

[Sam97] Augusto Sampaio. *An Algebraic Approach to Compiler Design*, volume 4 of *Algebraic Methodology and Software Technology.* World Scientific, 1997.

[SSB97] 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.