Available online at www.sciencedirect.com
SCIENCE DIRECT®
Science of Computer Programming 52 (2004) 53–100
Science of Computer Programming
www.elsevier.com/locate/scico

Algebraic reasoning for object-oriented programming

Paulo Borba[a,*], Augusto Sampaio[a], Ana Cavalcanti[b], Márcio Cornélio[a,1]

[a] *Informatics Center, Federal University of Pernambuco, Recife 50740-540, Brazil*
[b] *Computing Laboratory, University of Kent, Canterbury CT2 7NF, UK*

Received 1 April 2003; received in revised form 20 January 2004; accepted 23 January 2004

Available online 7 June 2004

Abstract

We present algebraic laws for a language similar to a subset of sequential Java that includes inheritance, recursive classes, dynamic binding, access control, type tests and casts, assignment, but no sharing. These laws are proved sound with respect to a weakest precondition semantics. We also show that they are complete in the sense that they are sufficient to reduce an arbitrary program to a normal form substantially close to an imperative program; the remaining object-oriented constructs could be further eliminated if our language had recursive records. This suggests that our laws are expressive enough to formally derive behaviour preserving program transformations; we illustrate that through the derivation of provably-correct refactorings.

1. Introduction

Programming laws [24] state properties of programming constructs and are useful for reasoning about programs [35], designing correct compilers [22,42], and, when interpreted as program transformations, supporting informal programming practices such as refactoring [18,37]. Several paradigms have benefited from algebraic programming laws. The laws of imperative programming [24] have been useful for providing algebraic semantic definitions and for establishing a sound basis for formal software development methods. The laws of OCCAM [41] exhibit useful properties of concurrency and

* Corresponding author.
E-mail addresses: phmb@cin.ufpe.br (P. Borba), a.l.c.cavalcanti@kent.ac.uk (A. Cavalcanti).
[1] Present address: Polytechnic School, University of Pernambuco, Recife 50750-410, Brazil.

0167-6423/$ - see front matter © 2004 Elsevier B.V. All rights reserved.
doi:10.1016/j.scico.2004.03.003

communication. Algebraic properties of functional programming are elegantly addressed in [4]. An algebraic approach to reasoning about logic programming is presented in [43]. More recently, unifying theories [23] have been proposed to study different paradigms, considering a variety of semantic presentations in an integrated way: denotational, operational, and algebraic.

The laws of object-oriented programming, however, are not well established. Laws for small-grain object-oriented constructs have been considered elsewhere [27,32], but medium-grain constructs have been neglected. Some laws have been informally discussed as refactorings [18], and formalised to the degree that they can be encoded in tools [37,39], but not proved sound or complete. In summary, there is no comprehensive, provably sound, set of laws to help developers understand and use the properties of mechanisms such as classes, inheritance, and subtyping. Furthermore, some of the laws of imperative programming are not directly applicable to corresponding object-oriented constructs. For instance, due to dynamic binding, the laws of procedure call are not directly valid for method call.

In this article, we describe a comprehensive set of laws for a language that is similar to a subset of sequential Java [8]. It includes, classes, inheritance, access control, dynamic binding, type tests and casts, recursion, assignment, and many other imperative features, including specification constructs. We adopt a copy semantics, so we do not model references or sharing. This does simplify the semantics of the language, but the laws related to object-oriented features do not rely on copy semantics. There is just one exception: the law for change of data representation. Moreover, in the absence of sharing, all laws are valid.

We present laws that deal with the imperative features of our language, but we concentrate on laws for its object-oriented features. We prove them to be sound with respect to a weakest precondition semantics first presented in [8]. Furthermore, we show that our set of laws is complete in the sense that it is sufficient to reduce an arbitrary program to a normal form substantially close to an imperative program; the remaining object-oriented constructs could be further eliminated if our language had recursive records. This follows an approach adopted for imperative and concurrent languages [24,41].

Using our laws, we describe and justify a strategy for reducing programs to normal form. This does not suggest a compilation process; its sole purpose is to prove a completeness result and, therefore, suggest that our set of laws is expressive. A first version of our completeness proof is presented in [6]; here we present a generalisation in the handling of recursive methods. More importantly, we consider the soundness of our laws; this was not addressed in [6].

Besides clarifying aspects of the semantics of object-oriented constructs, the major application of our laws is to formally derive more elaborate behaviour preserving program transformations useful for optimizing or restructuring object-oriented applications. In particular, we show how they can be used to derive provably-correct refactorings. For that, we also use the law for change of representation, which generalises the traditional data refinement law for a single program module [35] to class hierarchies.

This article is organized as follows. We first give an overview of the subset of Java that we consider. After that, in Section 3, we present the algebraic laws. In Section 4, we discuss the weakest precondition semantics of our language and the proof of soundness of

our laws. Completeness is considered in Section 5, where we present the normal form and a reduction strategy. The class refinement law is presented in Section 6. In Section 7 we show how the presented laws can serve as a basis for proving refactorings. Section 8 discusses related work and Section 9 summarises our results and topics for further research. The soundness result is new; it is integrated here to slightly extended and improved versions of previously published results [6,8].

2. The language

The language that we study is, mostly, a subset of sequential Java [21] with a copy semantics. The language is adequate for reasoning about both programs and specifications since it includes constructs, such as specifications statements, of Morgan's refinement calculus [35]. The syntax of commands, in particular, is based on that of Dijkstra's language of guarded commands [14].

A program $cds \bullet c$ in our language is a set of class declarations cds followed by a main command c. Classes are declared as in the following example, where we define a class called *Client*.

class *Client* **extends object**
  pri *name: String*; *addr: Address*
  ...
  meth *getStreet* $\widehat{=}$ (**res** r : *String* • **self**.*addr*.*getStreet*(r))
  meth *setStreet* $\widehat{=}$ (**val** s : *String* • **self**.*addr*.*setStreet*(s))
  new $\widehat{=}$ **self**.*addr* := **new** *Address* **end**;
end

Subclassing and single inheritance are supported through the **extends** clause. The built-in **object** class is a superclass of all the other classes, so the **extends** clause above could have been omitted.

The class *Client* includes two private attributes: *name* and *addr*, of types *String* and *Address*. Besides the **pri** qualifier for private attributes, there are qualifiers for protected (**prot**) and public (**pub**) attributes as in Java. For simplicity, the language supports no attribute redefinition and allows 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 •. The method *getStreet* has a result parameter r, and *setStreet* has a value parameter s. Constructors are declared by the **new** clause and do not have parameters. In contrast to Java, our language adopts a simple semantics for constructors: they are syntactic sugar for methods that are called after creating objects of the corresponding class.

The body of methods and constructors are commands similar to those of Morgan's refinement calculus. Their syntax is formalised as follows:

$c \in Com$ ::=	$le := e \mid c; c$	assignment, sequence
	$\mid x : [\psi_1, \psi_2]$	specification statement
	$\mid pc(e)$	parametrised command application
	$\mid$ **if** $[]\ i \bullet \psi_i \rightarrow c_i$ **fi**	conditional

$$
\begin{array}{ll}
\mid \textbf{rec}\ X \bullet c\ \textbf{end} \mid X & \text{recursion, recursive call} \\
\mid \textbf{var}\ x : T \bullet c\ \textbf{end} & \text{local variable block} \\
\mid \textbf{avar}\ x : T \bullet c\ \textbf{end} & \text{angelic variable block}
\end{array}
$$

We allow x, e, le, and T to also denote lists of identifiers, expressions and types; this shall be clear from the context. The expressions le that are allowed to appear as the target of assignments and method calls, and as result arguments, define the subset le (*left expressions*) of valid expressions. We define this set later in this section.

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 is the list of the variables whose values may change, and ψ_1 and ψ_2 are formulas of the predicate calculus. For conciseness, we omit the standard definition of the syntax of formulas. Like the languages adopted in other refinement calculi, we have a specification language where programs appear as an executable subset of specifications.

Although not usually deliberately written, the following specification is useful for reasoning.

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

It is never guaranteed to terminate (precondition **false**). It is also useful in program derivation or transformation to assume that a condition b holds at a given point in the program text. This can be characterised as an assumption of b, written as $\{b\}$, and defined as follows.

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

If b is **false**, the assumption reduces to **abort**. Otherwise, it behaves like a command that always terminates and does nothing, denoted by **skip**.

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

The empty frame guarantees that no variables are changed.

Methods are seen as parametrised commands [1,11], 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 parametrised commands. The syntax of parametrised commands is defined as follows.

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

The parametrised command $pds \bullet c$ declares parameters pds used in a command c. The parametrised command $le.m$ is a call to a method m with target object le. Parameters can be passed by value (keyword **val**) or result (**res**). In the body of the *getStreet* method of

the class *Client*, for instance, we have a call to a method *getStreet* with target *addr*, and argument r. A call to a method m on the current object must be written as **self**.m since **self** is not optional; in the case of redefinitions, the method declared by the superclass can be called by writing **super**.m.

Data types T are either primitive (**bool**, **int**, and others) or classes. We consider that methods cannot be mutually recursive, but classes can.

The conditional (alternation) command is in the style of the guarded **if** of Dijkstra's language. In the BNF, we use an informal indexed notation for a finite set of guarded commands $\psi_i \rightarrow c_i$ separated by []. We also have recursion and variable blocks. Angelic variables, also known as logical variables or logical constants, are similar to standard local variables, except that its initial value is angelically chosen to make sure c succeeds, if possible at all. For example, in the program fragment

$$\textbf{avar}\ x : int \bullet \{x = 2\}; \ldots \textbf{end}$$

the variable x is assigned value 2 upon (an implicit) initialisation; otherwise, the assumption $\{x = 2\}$ would behave like **abort**. Angelic declarations are not code, but they are useful for reasoning.

Our language includes typical object-oriented expressions:

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

The expressions **self**, **super**, and **is** have similar semantics to `this`, `super`, and `instanceof` (which does not require exact type matching) in Java, respectively. We must write **self**.a to access the attribute a of the current class, since **self** is not optional. The update expression $(e_1; x : e_2)$ denotes a copy of the object e_1, but with the attribute x mapped to a copy of e_2; this is similar to update of arrays in Morgan's refinement calculus [35]. So, despite its name, the update expression, similarly to the other 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 **null**.x and (**null**; $x : e$) cannot be successfully evaluated; they yield the special value **error** and lead the commands in which they appear to *abort*.

The left-expressions are defined as follows:

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

These are the expressions that can appear as targets of assignments, and as result arguments; they can also appear as targets of method calls, along with **self**, **super**, and cast expressions.

3. Algebraic laws

Algebraic laws are usually presented as context-independent equations, as in the law

$$(x := x) = \textbf{skip}$$

and several other laws of imperative programming [24,41]. Such laws are compositional; they can be used, for example, as rewrite rules and program transformations, and one can even think of more than one law being applied simultaneously to different fragments of a program. Due to independence of a particular context, these laws are also applicable to *open* programs.

The laws we propose in this section focus on the object-oriented features of our language. These laws are mostly concerned with properties of class declarations and method calls, which are inherently context-dependent, especially when considering class hierarchies. Therefore, the proposed laws need to address context issues. Equivalence of sets of class declarations cds_1 and cds_2 is denoted by $cds_1 =_{cds,c} cds_2$, where cds is a context of class declarations for cds_1 and cds_2, and c is the main command. This is just an abbreviation for the program equivalence $cds_1 cds \bullet c = cds_2 cds \bullet c$, which we formalise in the next section.

These laws consider the entire context, and therefore apply to *closed* programs. Nevertheless, their associated side conditions are purely syntactic. Furthermore, although the context is captured for each particular law application, this is by no means a requirement that the context be fixed for successive transformations. The first law introduced below allows elimination and introduction of class declarations; thus its application may change the context of a development. If, eventually, a modified context no longer satisfies the conditions of a law previously applied, this does not invalidate the effected transformation; it just means that in the current context the application of the law would not be valid.

Law 1. ⟨class elimination⟩

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

provided

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

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

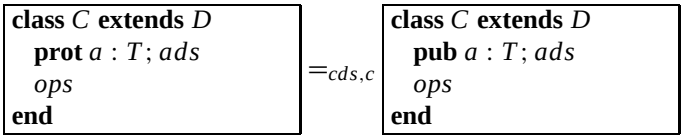
We write '($\rightarrow$)' before the first proviso since it is only required for applications of this law from left to right. We also write '($\leftarrow$)', when a proviso is necessary only for applying a law from right to left, and '($\leftrightarrow$)' when it is necessary in both directions. This also helps to interpret each law as two behaviour preserving transformations with different provisos.

We now present laws to manipulate attribute and method declarations, method calls, and commands in general.

3.1. Attribute declarations

The first laws we present in this section allow us to change the declaration of attributes. The following law relates protected and public attributes. From left to right, it establishes that a protected attribute can be made public; from right to left, it asserts that a public attribute can be made protected, provided that it is only directly used by the class in which it is declared and its subclasses. This proviso is necessary to guarantee that the law relates well-formed programs.

Law 2. ⟨change visibility: from protected to public⟩



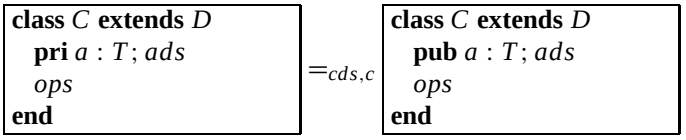
provided

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

We write **prot** $a : T; ads$ to denote the attribute declaration **prot** $a : T$ and all the declarations in ads, whereas *ops* stands for the declarations of methods and constructors. The notation $B.a$ refers to uses of the name a via expressions whose static type is exactly B, as opposed to any of its subclasses. For example, if we write that $B.a$ does not appear in *ops*, we mean that *ops* does not contain any expression such as $e.a$, for any e of type B, strictly. The subclass relation is denoted by $\leq$.

Our second law relates private and public attributes.

Law 3. ⟨change visibility: from private to public⟩



provided

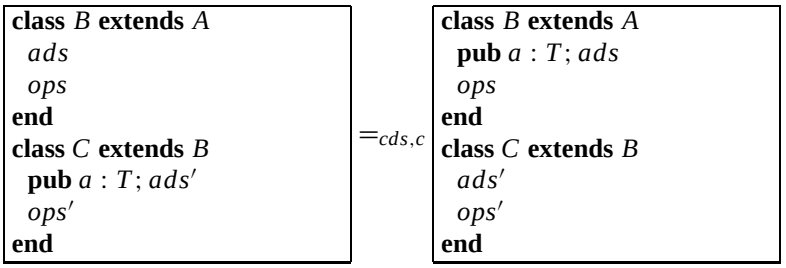
(←) $B.a$, for any $B \leq C$, does not appear in cds, c. □

When applied from right to left, this law makes a public attribute private. For that, the attribute cannot be used anywhere outside the class where it is declared; this is enforced by the proviso. The law that allows us to change attribute visibility from private to protected, and vice versa, can be derived from the above two laws.

The following law establishes that we can move a public attribute a from a class C to a superclass B, and vice versa. To move the attribute up to B, it is required that this does not generate a name conflict: no subclass of B, other than C, can declare an attribute with the same name; our language does not allow attribute redefinition or hiding as in Java. We do not need to worry about a being declared in B itself, as this is not possible: if it were, then

C would not be well-formed. We can move a from B to C provided that a is used only as if it were declared in C.

Law 4. ⟨move attribute to superclass⟩



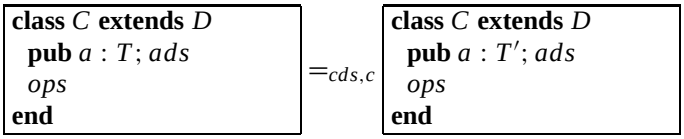
provided

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

The second proviso above, according to the special notation $D.a$ previously introduced, 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 whether a is declared in B or in C.

The following law allows us to change the class type of an attribute to a supertype, and vice versa.

Law 5. ⟨change attribute type⟩



provided

(↔) $T \leq T'$ and every non-assignable occurrence of a in expressions of ops, cds and c is cast with T or any subtype of T declared in cds.
(←) (1) every expression assigned to a, in ops, cds and c, is of type T or any subtype of T; (2) every use of a as result argument is for a corresponding formal parameter of type T or any subtype of T. □

Assignable occurrences of identifiers are result arguments and targets of assignments. For instance, in **self**.$a := e$ and $le.m$(**self**.a), the occurrences of a are assignable, if the single parameter of m is passed by result. On the other hand, in an assignment **self**.$a.x := e$, there is an assignable occurrence of x but not of a. Therefore, a is required to be cast in the proviso above. The same comment applies to a result argument **self**.$a.x$. Occurrences of identifiers as result arguments and targets of assignments are not cast anywhere; like in Java, this is not allowed in our language.

3.2. Method declarations

In this section we give laws related to the declaration of methods. The following law states that we can introduce or remove a trivial method redefinition, which amounts simply to a call to the method in the superclass.

Law 6. ⟨introduce method redefinition⟩

```
class B extends A                    =   class B extends A
  ads                                      ads
  meth m ≙ pc                              meth m ≙ pc
  ops                                      ops
end                                    end
class C extends B                      class C extends B
  ads′                                     ads′
  ops′                                     meth m ≙ super.m
end                                        ops′
                                       end
```

provided

(→) m is not declared in ops'. □

Strictly, we cannot define a method as **meth** $m \mathrel{\widehat{=}}$ **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** $m \mathrel{\widehat{=}} (pds \bullet$ **super**.$m(\alpha pds))$, where αpds denotes the list of parameter names declared in pds. For simplicity, however, we adopt the shorter notation **meth** $m \mathrel{\widehat{=}}$ **super**.m.

The next law states that we can merge a method declaration and its redefinition into a single declaration in the superclass. The resulting method uses type tests to choose the appropriate behaviour.

Law 7. ⟨move redefined method to superclass⟩

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

provided

($\leftrightarrow$) (1) **super** and private attributes do not appear in b'; (2) **super**.m does not appear in ops';
($\rightarrow$) b' does not contain uncast occurrences of **self** nor expressions in the form $((C)\textbf{self}).a$ for any private attribute a in ads';
($\leftarrow$) m is not declared in ops'. □

The provisos concerning **super** are needed because its semantics may be affected if it is moved from a subclass to a superclass, or vice versa. The other provisos ensure the validity of the programs involved. We can only move the body of m up if it does not refer to elements of the class where it is declared through uncast **self**. As mentioned in the previous section, **self** must be used for calling methods and selecting attributes of the current object.

Our third method law allows us to move up in the class hierarchy a method declaration that is not a redefinition. Our language supports method redefinition but, as opposed to Java, not overloading. Hence, we cannot have different methods in the same class, or in a class and a subclass, with the same name, but different parameters. Our 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⟩

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

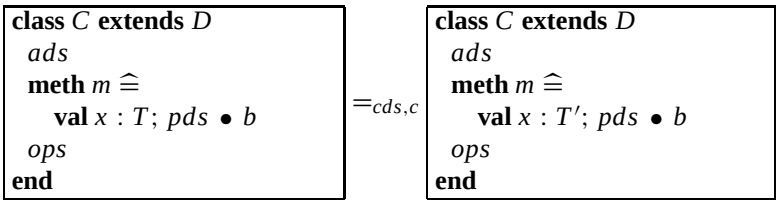
provided

($\leftrightarrow$) (1) **super** and private attributes do not appear in pc; (2) m is not declared in any superclass of B in cds;
($\rightarrow$) (1) m is not declared in ops, and can only be declared in a class D, for any $D \leq B$ and $D \not\leq C$, if it has the same parameters as pc; (2) pc does not contain uncast occurrences of **self** nor expressions in the form $((C)\textbf{self}).a$ for any private attribute a in ads';
($\leftarrow$) (1) m is not declared in ops'; (2) $D.m$, for any $D \leq B$, does not appear in cds, c, ops or ops'. □

The provisos for this law are similar to those of Laws 4 and 7. Only the first two are necessary to preserve semantics; the others guarantee that we relate syntactically valid programs. The second proviso, associated to applications of the law in both directions, precludes superclasses of B from defining m, because, otherwise, when moving it, we could affect the semantics of calls such as $b.m(e)$, for a b storing an object of B.

The next two laws allow us to change the type of a parameter; they are similar to Law 5. The first law handles value parameters.

Law 9. ⟨change value parameter type⟩

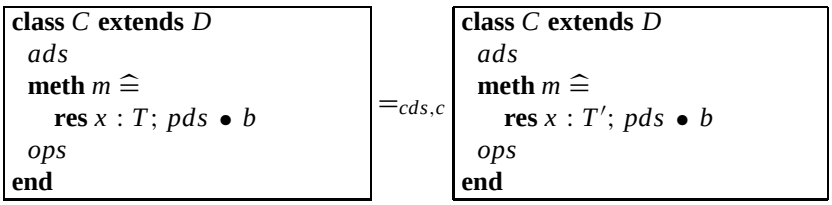


provided

(↔) $T \leq T'$ and every non-assignable occurrence of x in expressions of b are cast with T or any subtype of T;
(←) (1) every actual parameter associated with x in *ops*, *cds*, and c is of type T or any subtype of it; (2) every expression assigned to x in b, is of type T or any subtype of T; (3) every use of x as result argument in b is for a corresponding formal parameter of type T or any subtype of T. □

For a result parameter, we have the following law. As opposed to a value argument, the type of a result argument has to be that of the corresponding formal parameter or a supertype of it. We cannot change the type of a parameter to a supertype of any of the arguments used in the program.

Law 10. ⟨change result parameter type⟩



provided

(↔) $T \leq T'$ and every non-assignable occurrence of x in expressions of b are cast with T or any subtype of T;
(→) every actual parameter associated with formal parameter x in *ops*, *cds*, and c is of type T' or any supertype of it;
(←) (1) every expression assigned to x in b is of type T or any subtype of T; (2) every use of x as result argument in b is for a corresponding formal parameter of type T or any subtype of T. □

The first proviso is the same as that in the previous law: it restricts the way in which the parameter is used in the method body. The second proviso is related to the use of arguments. The third proviso is similar to that in Law 5.

A method that is not called can be eliminated. Conversely, we can always introduce a new method in a class.

Law 11. ⟨method elimination⟩

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

provided

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

The introduction and elimination of attributes is considered in Section 6.

3.3. Method calls

The laws in this and in the next section give properties of the equivalence relation for commands, instead of programs or class declarations as those in the previous sections. The following law indicates that we can replace a method call **super**.m in a class C by a copy of the body of m as declared in the immediate superclass of C, provided the body does not contain **super** nor private attributes.

Law 12. ⟨eliminate **super**⟩
Consider that *CDS* is a set of two class declarations as follows.

class B **extends** A
ads
meth $m \mathrel{\widehat{=}} pc$
ops
end

class C **extends** B
ads'
ops'
end

Then we have that

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

provided

($\rightarrow$) **super** and the private attributes in ads do not appear in pc. □

The notation *cds CDS* denotes the union of the class declarations in *cds* and *CDS*, and '$cds, N \rhd c = d$' 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 might use **main** for asserting that the equality holds inside the main program.

Law 12 is similar to the standard copy rule for procedures [35]; for calls **super**.m, dynamic binding does not apply. The arguments to which **super**.m is applied are not touched by this law; pc ends up applied to the same arguments.

In the case where a method is not redefined, and there are no visibility concerns, we can use the copy rule to characterise method calls. It might be surprising that we need only such simple laws to characterise method call elimination. The reason is that dynamic binding is handled by Law 7 as a separate issue. Hereafter, the notation $cds, N \rhd e : C$ is used to indicate that in the class N declared in cds, the expression e has static type C. Again, instead of a class name, we might use **main** for asserting that the typing holds inside the main program.

Law 13. ⟨method call elimination⟩

Consider that the following class declaration

class C **extends** D
ads
meth $m \mathrel{\widehat{=}} pc$
ops
end

is included in cds and $cds, A \rhd le : C$. Then

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

provided

($\leftrightarrow$) (1) m is not redefined in cds and pc does not contain references to **super**; (2) all attributes which appear in the body pc of m are public. □

A method call $le.m(e)$ aborts when le is **null** or **error**. Thus, we need the assumption $\{le \neq \textbf{null} \wedge le \neq \textbf{error}\}$ on the right-hand side of the above law. The law for a call **self**$.m(e)$ is similar. As already mentioned in Section 2, the assumption $\{b\}$ behaves like **skip** if b is true, and as **abort** otherwise. The notation $pc[le/\textbf{self}]$ denotes the parametrised command pc where **self** is replaced with le.

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 requires a cast, as in $((C)x).a$. If a is declared in B, then the cast is not necessary for compilation, but once it is there, it cannot simply be eliminated, because a cast also has a run time effect.

At run time, if the value of a cast expression does not have the required type, its evaluation results in **error**, and the command in which it appears aborts. In the example above, if the attribute a is in class B, although the cast could be eliminated regarding its static effect, it still has a dynamic effect when the object value of x happens to be of type C, but not of type B.

In order to capture the behaviour of casts, we use assumptions. The following law deals with the elimination of type casts in targets of method calls.

Law 14. ⟨eliminate cast of method call⟩

If $cds, A \rhd e : B, C \leq B$ and m is declared in B or in any of its superclasses in cds, then $cds, A \rhd ((C)e).m(e') = \{e \textbf{ is } C\}; e.m(e')$. □

Casts in arguments can also be eliminated, but we omit this similar law.

3.4. Commands and expressions

In the same way that the type on an attribute (Law 5) or parameter (Laws 9 and 10) can be changed if all its uses are cast, we can also change the type of a local variable in this case.

Law 15. ⟨change variable type⟩

$$cds, A \rhd \textbf{var } x : T \bullet c \textbf{ end} = \textbf{var } x : T' \bullet c \textbf{ end}$$

($\leftrightarrow$) $T \leq T'$ and every non-assignable occurrence of x in expressions of c is cast with T or any subtype of T;
($\leftarrow$) (1) every expression assigned to x in c is of type T or any subtype of T; (2) every use of x as result argument in c is for a corresponding formal parameter of type T or any subtype of T. □

The same holds for angelic variables.

The following law formalises the fact that any expression can be cast with its declared type.

Law 16. ⟨introduce trivial cast in expressions⟩
If $cds, A \rhd e : C$, then $cds, A \rhd e = (C)e$. □

For simplicity, this is formalised as a law of expressions, not commands. Nevertheless, it should 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 the same antecedent as Law 16.

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

This law is equally valid for left-expressions, which are a form of expression. However, our language, like Java, does not allow casts to appear in targets of assignments and result parameters. So Law 16 should not be considered an abbreviation for laws such as

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

which are not valid since $((C)e) := e'.x$ is not a command in our language, even if e is a left expression.

Law 14, presented in the previous section, allows us to eliminate casts in targets of method calls. We can also eliminate casts in assignments as below.

Law 17. ⟨eliminate cast of expressions⟩
If $cds, A \rhd le : B, e : B', C \leq B'$ and $B' \leq B$, then

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

Similar laws apply to expressions in conditionals and other points of a program.

Two simple laws of type test are presented below; they are laws of expressions. Law 18 asserts that the type test **self is** M is true when appearing inside a subclass of M.

Law 18. ⟨**is** test true⟩
If $N \leq_{cds} M$, then $cds, N \rhd \textbf{self is } M = \textbf{true}$ □

In complement to Law 18, Law 19 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 19. ⟨**is** test false⟩
If $N \not\leq_{cds} M$ and $M \not\leq_{cds} N$, then $cds, N \rhd \textbf{self is } M = \textbf{false}$ □

The following two laws express simple properties of the alternation command. The first law allows us to simplify an alternation whose commands are the same in all branches, assuming that the disjunction of all guards is true.

Law 20. ⟨**if** identical guarded commands⟩
If $\bigvee i : 1 \ldots n \bullet \psi_i = \textbf{true}$, then $\textbf{if } [] \ i : 1 \ldots n \bullet \psi_i \ \rightarrow \ c \ \textbf{fi} = c$. □

The other law states that the order of the guarded commands of an alternation is immaterial.

Law 21. ⟨**if** symmetry⟩
If π is any permutation of $1 \ldots n$, then

$$\textbf{if } [] \ i : 1 \ldots n \bullet \psi_i \ \rightarrow \ c_i \textbf{ fi} = \textbf{if } [] \ i : 1 \ldots n \bullet \psi_{\pi(i)} \ \rightarrow \ c_{\pi(i)} \textbf{ fi}$$ □

The soundness of our laws is considered in the next section. Examples of their use can be found in Sections 5 and 7.

4. Soundness

In order to guarantee that the laws presented in the previous section are sound, so that the associated program transformations preserve behaviour, we use a formal semantics to prove their validity. In this section, we first present the typing rules of our language and a weakest precondition semantics that is defined by induction on the typing rules [8]. Based on that, later in the section, we discuss the soundness of our laws.

4.1. Typing

In many presentations of weakest precondition semantics, the typing context is implicit, because a fixed global state is adequate for a simple imperative language. In object-oriented languages, however, this context plays a strong role in the semantics due to dynamic binding and visibility. We formalise a type system for our language and define the semantics by induction on typing judgements, as we often use typing information in semantic definitions.

We define a judgement $\Gamma, \Sigma, N \rhd c : \textbf{com}$ for when a command c is well-typed in a context defined by the class declarations recorded in the typing environment Γ, by the variables in the local signature Σ, and the class N. The typing environment records the class names, the attributes and methods available by declaration or inheritance in each of them, the attributes types and visibility, the method parameters, and the inheritance relationship. The signature records the attributes visible in N, and the parameters and local variables in scope. The main command is regarded as part of a special class called **main**. In the judgement $\Gamma, \Sigma, \textbf{main} \rhd c : \textbf{com}$, the signature Σ includes the global program variables, which represent its inputs and outputs.

Table 1
Selected typing rules

$$\frac{N \neq \textbf{main}}{\Gamma, N \rhd \textbf{self} : N} \qquad \frac{\Gamma, N \rhd e : N' \qquad \Gamma.attr\ N'\ x = T \qquad visib\ \Gamma\ N'\ N\ x}{\Gamma, N \rhd e.x : T}$$

$$\frac{\Gamma, (\Sigma; x : T) \rhd c : \textbf{com} \qquad par \in \{\textbf{val}, \textbf{res}, \textbf{vres}\}}{\Gamma, \Sigma \rhd (par\ x : T \bullet c) : \textbf{pcom}(par\ x : T)}$$

$$\frac{\Gamma \rhd le : T \qquad \Gamma \rhd e : T' \qquad T' \leq_\Gamma T}{\Gamma \rhd le := e : \textbf{com}}$$

$$\frac{\Gamma \rhd le.m : \textbf{pcom}(pds) \qquad \Gamma \rhd e : T \qquad sdisjoint(le, rargs\ pds\ e) \qquad aptype\ \Gamma\ pds\ e\ T}{\Gamma \rhd le.m(e) : \textbf{com}}$$

$$\frac{\Gamma, \textbf{main} \rhd c : \textbf{com} \qquad (\Gamma, \Sigma) = ((VDecs\ cds\ \textbf{main}); x : T) \qquad Vmeth\ \Gamma\ cds}{x : T \rhd cds \bullet c : \textbf{program}}$$

We also have judgements for expressions, predicates, parametrised commands, and programs. For expressions, the judgement $\Gamma, \Sigma, N \rhd e : T$ asserts that e is well-typed and has type T in the context characterised by Γ, Σ, N. In fact, e and T can be lists of expressions and corresponding types; the context should make clear which one is meant. In the laws, we have used $cds, N \rhd e : T$ as an abbreviation for $\Gamma, \Sigma, N \rhd e : T$, where Γ and Σ are the typing environment and signature determined by the set of class declarations cds and the class N. For parametrised commands, $\Gamma, \Sigma, N \rhd pc : \textbf{pcom}(pds)$ asserts that pc is well-typed and has parameters pds.

Table 1 presents some typing rules; there, we omit the local signature Σ and the class N when they do not change. The typing rule for **self** states that its type is that of the current class N. This class name, however, should not be **main**, since this actually denotes the main program, which cannot include occurrences of **self**.

The type of $e.x$ is that of the x attribute of the class N' of e ($\Gamma.attr\ N'\ x$), provided this attribute is visible from the current class. Visibility is considered in $visib\ \Gamma\ N'\ N\ x$, a condition stating that x is an attribute of N' visible from inside the class N.

In $(par\ x : T \bullet c)$, we use par to denote a keyword that describes the passing mechanism for the parameter x. This parametrised command is well-typed if c is well-typed in the extended context that includes x in the local signature. Even though our language does not include value-result parameters, in the semantics, we consider an extended language with constructs that simplify definitions. This extended language does include value-result parameters.

An assignment $le := e$ is well-typed if its source e and its target e are well-typed. Moreover, the type T' of e must be a subtype of the type T of le. For a typing environment Γ, the subtyping relation is denoted by $\leq_\Gamma$.

A method called $le.m(e)$ is well-typed if the parametrised command $le.m$ and the list of arguments e are well-typed. Moreover, $le.m(e)$ potentially modifies le. So, to avoid aliasing, we require that the list formed by le and the result arguments ($rargs\ pds\ e$) does not have repetitions ($sdisjoint(le, rargs\ pds\ e)$). The arguments e must also have types appropriate with respect to the mechanisms by which they are passed. For instance, the

type of a value argument must be a subtype of the parameter type. This is enforced by *aptype* Γ *pds e* T.

Finally, we have a typing judgement $x : T \rhd cds \bullet c : \textbf{program}$ for complete programs $cds \bullet c$. The context is defined simply by the global variables x. The program $cds \bullet c$ is well-typed, if c is well-typed in the context $\Gamma, \Sigma, \textbf{main}$, where Γ and Σ are the typing environment and the local signature defined by *cds* and x $((\Gamma, \Sigma) = ((VDecs\ cds\ \textbf{main}); x : T))$. Moreover, the methods in the classes in *cds* have to be well-typed ($Vmeth\ \Gamma\ cds$).

4.2. Semantics

As already mentioned, we define the semantics by induction on typing rules. Nevertheless, our semantics is a function of typings, as proved in [8]. We present first the semantics of commands, parametrised commands, and programs. The semantics of method calls is discussed last.

The semantics of commands and parametrised commands rely on an environment that, for each method, records a parametrised command obtained by adding to the declaration of the method an extra parameter, *me*, passed by value-result. It provides the target object of method calls, so that we can interpret a method body in the context of its calls.

The typing and visibility restrictions imposed on user programs are too strong for the semantics. We actually base our definitions in what we call an extended typing system. Its main difference from the typing system presented here is that it does not enforce the visibility restrictions. These are natural for user programs, but not for the semantics, where, for example, we evaluate the body of a method in the context of its calls.

4.2.1. Commands and parametrised commands

For a command typing $\Gamma, \Sigma, N \rhd c : \textbf{com}$, and an environment η, the weakest precondition semantics $[\![\Gamma, \Sigma, N \rhd c : \textbf{com}]\!]\eta$ is a total function on predicates. Many of our definitions are very similar to those of imperative languages [36]. We explore a few more interesting examples.

The semantics of an assignment to a variable is very much standard; we present the definitions in a form similar to the typing rules.

$$\frac{\Gamma \rhd x : T \quad \Gamma \rhd e : T' \quad T' \leq_\Gamma T}{[\![\Gamma \rhd x := e : \textbf{com}]\!]\eta\ \psi = (e \neq \textbf{error} \wedge \psi[e/x])}$$

Since expressions like $e.x$ are partial, as they may have value **error** if e is **null**, above we have to require that e does not evaluate to **error**. The weakest precondition that guarantees that $x := e$ establishes a postcondition ψ is that e does not evaluate to **error**, and ψ holds when x takes the value e.

Assignments can be rewritten to assignments of update expressions to variables and to **self**. For example, $le.x := e$ can be written as $le := (le; x : e)$. Therefore, we need to give a weakest precondition semantics only to assignments of the form $x := e$ and $\textbf{self} := e$. Assignments to **self** are not supposed to occur in user programs, but they arise as part of the definition of the semantics of method calls and assigments such as $\textbf{self}.x := e$, which are written as $\textbf{self} := (\textbf{self}; x : e)$.

The semantics of a method call of the form **self**.$m(e)$ is given in terms of a parametrised command associated to m in the environment. This command is of the form (**vres** me : T; $pds \bullet c$), where the parameter me is used to access the target of the call, pds declares the parameters of m, and c is a version of the body of the invoked version of m that makes use of me. In the semantics of **self**.$m(e)$, this parametrised command is applied to **self** and e. A parametrised command application (**vres** $x : T \bullet c$)(le) is defined by the variable block **var** $l : T \bullet l := le$; $c[l/x]$; $le := l$ **end**, where l is a fresh variable used to hold and update the argument le. If le turns out to be **self**, as is the case in the semantics of **self**.$m(e)$, we have an assignment of l to **self**.

In these assignments, **self** is always assigned an object of the current class, so that the target and the source of the assignment have the same type. Our semantics definition, however, does not depend on this assumption.

$$\frac{\Gamma, N \rhd e : N' \quad N' \leq_\Gamma N}{[\![\Gamma, N \rhd \textbf{self} := e : \textbf{com}]\!]\eta\ \psi =}$$

$$(\vee_{N'' \leq_\Gamma N'} \bullet e\ \textbf{isExactly}\ N'' \wedge \psi[e, e.x/\textbf{self}, x]) \text{with } x = \text{dom}(\Gamma.attr\ N'')$$

The weakest precondition that guarantees that **self** $:= e$ establishes ψ is a disjunction over the subclasses N'' of the type N' of e. The test e **isExactly** N holds when the value of e is an object of class N, but not of any of its subclasses. Each disjunct requires ψ to hold when **self** takes the value e, and the attributes x of N'' take the value $e.x$. There is no need to check that e is not **error** because **error isExactly** N'' is false, for all N''. If **self** is assigned an object of the current class, the semantics simplifies to

$$e \neq \textbf{error} \wedge \psi[e, e.x/\textbf{self}, x].$$

4.2.2. Programs

The semantics of a program is that of the main command, in the context defined by the class declarations and the global variables.

$$\frac{[\![\Gamma, \Sigma, \textbf{main} \rhd c : \textbf{com}]\!]\eta = f \qquad (\Gamma, \Sigma) = ((VDecs\ cds\ \textbf{main}); x : T) \quad \eta = Meths\ \Gamma\ cds}{[\![x : T \rhd cds \bullet c : \textbf{program}]\!] = f}$$

The environment η is determined from *cds* by the function *Meths*. It is constructed using fixed points to handle recursive methods. To handle mutual recursion, a simultaneous fixed point could be taken for all methods. This, however, would complicate proofs of laws; we, therefore, disallow mutual recursion of methods, including forms that arise through dynamic binding.

Fixed points are taken for each method: if a class N declares a method $m \mathrel{\widehat{=}} (pds \bullet c)$, we take a least fixed point in the lattice of parametrised commands with parameters **vres** me : N; pds. In this respect, ($pds \bullet c$) is regarded as a context: a function of parametrised commands to parametrised commands. Its application to a parametrised command pc, with the same extended parameter declaration, yields the result of substituting pc for the occurrences of m in c. Its fixed point is associated in the environment with all classes N' that inherit m from N. This approach is similar to that in [1,10].

4.2.3. Method calls

To give the semantics of a call $le.m(e)$, we use the definitions of m in the environment: there is one for the static class type N'' of le and one for each of its subclasses. If, for example, $le.m$ is typed as $\Gamma, N \rhd le.m : \mathbf{pcom}(\mathbf{val}\ x : T)$, the parametrised commands associated with m in each of the subclasses N' of N'', including N'' itself, takes the form $(\mathbf{vres}\ me : N'; \mathbf{val}\ x : T \bullet c)$. If we define $f_{N'}$ as $[\![\Gamma, N \rhd (\mathbf{vres}\ me : N'; \mathbf{val}\ x : T \bullet c)((N')le, e) : \mathbf{com}]\!]\eta$, then the semantics of $le.m(e)$ is $f_{N'}\ \psi$, provided N' is the dynamic type of le.

In our definition below, the semantics is the disjunction, over the possible classes N', of le **isExactly** $N' \wedge f_{N'}\psi$. Thus the semantics $f_{N'}$ is used just when it should be. The possible classes N' are the subclasses of N''.

$$\frac{[\![\Gamma, N \rhd (\eta\ N'\ m)((N')le, e) : \mathbf{com}]\!]\eta = f_{N'} \text{ with } N' \leq_\Gamma N'', \text{the type of } le}{[\![\Gamma, N \rhd le.m(e) : \mathbf{com}]\!]\eta\ \psi = (\vee_{N' \leq_\Gamma N''} \bullet le\ \mathbf{isExactly}\ N' \wedge f_{N'}\psi)}$$

The parametrised command $\eta\ N'\ m$ has to be well-typed in the context of the call. It refers to attributes through the me parameter. Those attributes are not necessarily visible in the context of the call, so references to them are only valid in the extended system, where visibility constraints do not apply. The cast $(N')le$ is needed because me is a value-result parameter of $\eta\ N'\ m$ and has type N'; the argument has to have type N'.

Even though many other studies of the semantics of method calls are available in the literature, we are not aware of another weakest precondition semantics for an object-oriented language. The results summarised here extend standard work for imperative languages, and provide a suitable basis for the study of refinement involving both programs and specifications, which we discuss in the next section.

4.3. Refinement

Besides the definition of the semantics, in order to prove that our laws are sound we need a suitable notion of refinement that allows us to compare programs. Equivalence is defined as refinement in both ways.

The refinement relationship between programs is defined in a rather standard way.

Definition 1. For sets of class declarations cds and cds', commands c and c' with free variables $x : T$, $(cds \bullet c) \sqsubseteq (cds' \bullet c')$, if and only if, for all ψ, $[\![x : T \rhd (cds \bullet c) : \mathbf{program}]\!]\ \psi \Rightarrow [\![x : T \rhd (cds' \bullet c') : \mathbf{program}]\!]\ \psi$ □

The free variables of a program represent its inputs and outputs; therefore, it makes sense to compare only programs with the same set of free variables.

There are two ways in which a program can be refined: refinement of its command part and refinement of its class declarations. Refinement of commands can also be defined in a standard way.

Definition 2. For a typing environment Γ, a signature Σ, a class N, and an environment η, we have $\Gamma, \Sigma, N, \eta \rhd c \sqsubseteq c'$ if and only if, for all predicates ψ, $[\![\Gamma, \Sigma, N \rhd c : \mathbf{com}]\!]\eta\ \psi \Rightarrow [\![\Gamma, \Sigma, N \rhd c' : \mathbf{com}]\!]\eta\ \psi$. □

In the laws, we use the notation $cds, N \rhd c = c'$ as an abbreviation; the corresponding environments are those determined by *cds* and N.

A class defines a data type, so refinement of classes is related to data refinement. Class refinement requires that any complete program that uses the abstract classes *cdsa* is refined when these classes are replaced with the alternative concrete classes $cdsc$. Program refinement, however, compares programs that act on the global variables. Therefore, these variables cannot hold values of the refined classes.

Definition 3. For sets of class declarations *cds*, *cdsa*, and *cdsc*, we define $cds \rhd cdsa \preccurlyeq cdsc$ if and only if (a) $(cds\ cdsa)$ and $(cds\ cdsc)$ are both well-formed; (b) for all commands c that use only methods in *cds* and *cdsa*, and whose global variables have types that are N-free, and for all classes N declared in *cdsa*, if c is well-typed for $cds\ cdsa$, **main**, then c is also well-typed in $cds\ cdsc$, **main**; and $(cds\ cdsa \bullet c) \sqsubseteq (cds\ cdsc \bullet c)$. □

A variable of an N-free type cannot have as value or as component an object of N or of its subclasses. For a value of a class type, we use the term component for its attributes, the attributes of its object-valued attributes, and so on.

The semantics of a main command c with a global variable of some class with an attribute of an abstract type N in the context $cds\ cdsc$ is different from that in the context $cds\ cdsa$, because the values handled by c in the different contexts are different. Therefore, it does not make sense to compare them by algorithmic refinement. The restriction to N-free global variables, however, does not preclude the use of N as the type of (components of) local variables of c, and also of parameters and local variables of methods called by c. The typing requirement in the above definition ensures that the methods provided by $cdsc$ include those provided by *cdsa*, with the same signatures.

In [9], we prove that forwards simulation is a sound technique for class refinement. Simulation is defined for commands in the standard way; the main point is that the coupling invariant has to be lifted to take into account the uses of the abstract classes in the definitions of other classes. As an example, we consider that the coupling invariant *ci* establishes a relation between objects of an abstract class *Abs* and a concrete class *Conc*. If *Abs* is the type of an attribute of a third class *Client*, then a generalised coupling invariant has to be defined to relate objects of *Client* in the context of *Abs* with objects of *Conc* in the context of *Conc*. For parametrised commands, simulation requires that, when applied to related arguments, they lead to related commands.

For classes, we write $ci \rhd cdsa \preceq cdsc$, or $cdsa \preceq_{ci} cdsc$, when the class declarations *cdsa* are simulated by those in $cdsb$, with coupling invariant *ci*; we define this relation as follows.

Definition 4. For sets of class declarations *cdsa* and $cdsc$, and a coupling invariant *ci*, we have that $ci \rhd cdsa \preceq cdsc$ if and only if, for all methods m of all classes in Ns in *cdsa* and $cdsc$, we have that $ci, Ns \rhd (\eta\ Ns\ m) \preceq (\eta'\ Ns\ m)$. The environments η and η' are those determined by *cdsa* and $cdsc$.

We write $ci, Ns \rhd (\eta\ Ns\ m) \preceq (\eta'\ Ns\ m)$ to denote that the parametrised commands $(\eta\ Ns\ m)$ and $(\eta'\ Ns\ m)$ of class Ns are related by simulation with coupling invariant *ci*. The proof of simulation laws is the subject of current work.

4.4. Example proofs

In this section, we present soundness proofs of some laws presented in Section 3 in order to exemplify how proofs are worked out in the weakest precondition semantics of our language. These proofs assure that program transformations accomplished by the use of the algebraic laws preserve semantics.

4.4.1. Visibility change

The proof of Law 3 is based on Lemma 5 below; it states that the weakest precondition of a command is the same after changing the visibility of an attribute from private to public. In the typing environment Γ, the attribute a is private ($\Gamma.vis\ N\ a = \mathbf{pri}$), whereas in Γ' its visibility is public: $\Gamma'.vis\ N = \Gamma.vis\ N \oplus \{a \mapsto \mathbf{pub}\}$.

Lemma 5. *Let Γ and Γ' be typing environments such that $\Gamma.vis\ N\ a = \mathbf{pri}$ and $\Gamma'.vis\ N = \Gamma.vis\ N \oplus \{a \mapsto \mathbf{pub}\}$, but are otherwise identical.*

$$[\![\Gamma, \Sigma, N \rhd c : \mathbf{com}]\!]\eta\ \psi = [\![\Gamma', \Sigma, N \rhd c : \mathbf{com}]\!]\eta\ \psi$$

Proof. By induction. Since the semantics is defined with basis on the extended typing system, which does not enforce the visibility constraints, the difference between Γ and Γ' is irrelevant. □

The proof of Law 3 is a direct consequence of Lemma 5 and the definition of the semantics of programs. The proviso of the law guarantees that the programs involved are well-formed, so that the construction of the typing environments for them is well-defined, and gives rise to environments that satisfy the hypothesis of Lemma 5.

4.4.2. Move redefined method to superclass

The core of the proof of Law 7 is the lemma below. The syntactic function *meI* transforms a method body (command) to make use of the extra *me* parameter used in the environment. Apart from the command itself, it takes as parameters the typing environment, and the class and name of the method.

Lemma 6. *Consider a typing environment Γ, and environments η and η' such that*

$$\eta\ B\ m = (\mathbf{vres}\ me : B; pds \bullet meI\ \Gamma\ B\ m\ b)$$

and

$$\eta\ C\ m = (\mathbf{vres}\ me : C; pds \bullet meI\ \Gamma\ C\ m\ b')$$

Moreover,

$$\eta'\ B\ m = (\mathbf{vres}\ me : B; pds \bullet \begin{array}{l}\mathbf{if}\ \neg(me\ \mathbf{is}\ C)\ \rightarrow\ meI\ \Gamma\ B\ m\ b \\ [\!]\ me\ \mathbf{is}\ C\ \rightarrow\ meI\ \Gamma\ B\ m\ b' \\ \mathbf{fi})\end{array}$$

and

$$\eta'\ C\ m = (\mathbf{vres}\ me : C; pds \bullet \begin{array}{l}\mathbf{if}\ \neg(me\ \mathbf{is}\ C)\ \rightarrow\ meI\ \Gamma\ C\ m\ b \\ [\!]\ me\ \mathbf{is}\ C\ \rightarrow\ meI\ \Gamma\ C\ m\ b' \\ \mathbf{fi})\end{array}$$

For all other classes and methods, η and η' are equal. If B, C, b, and b' are as in Law 7, then, for all classes N,

$$[\![\Gamma, N \rhd c : \mathbf{com}]\!]\eta\ \psi = [\![\Gamma, N \rhd c : \mathbf{com}]\!]\eta'\ \psi$$

Proof. By induction. The different environments potentially affect the semantics of method calls. If the method called is not m, then the semantics recorded in η and η' are the same, so the result is trivial. The case we consider below is that of a call to m.

Case. $le.m(e)$

$$\begin{array}{ll}
& [\![\Gamma, N \rhd le.m(e) : \mathbf{com}]\!]\eta\ \psi \\
= & \vee_{N' \leq_\Gamma N''} \bullet le\ \mathbf{isExactly}\ N' \wedge \\
& \qquad [\![\Gamma, N \rhd (\eta\ N'\ m)\ ((N')le, e) : \mathbf{com}]\!]\eta\ \psi & \text{[semantics]}
\end{array}$$

In this definition, if le is not **null** or **error**, then le **isExactly** N'' holds for exactly one subclass N'' of the type N of le. If N'' is B or C, then η and η' record different semantics for m. Otherwise, the semantics are the same, and the result is trivial.

If the exact type of le is B, we can proceed as follows.

$$\begin{array}{ll}
& \vee_{N' \leq_\Gamma N''} \bullet le\ \mathbf{isExactly}\ N' \wedge \\
& \qquad [\![\Gamma, N \rhd (\eta\ N'\ m')((N')le, e) : \mathbf{com}]\!]\eta\ \psi \\
= & [\![\Gamma, N \rhd (\eta\ B\ m)((B)le, e) : \mathbf{com}]\!]\eta'\ \psi & \text{[assumption]} \\
= & [\![\Gamma, N \rhd (\mathbf{vres}\ me : B;\ pds \bullet meI\ \Gamma\ B\ m\ b)((B)le, e) : \mathbf{com}]\!]\eta'\ \psi & \text{[hypothesis]} \\
= & [\![\Gamma, N \rhd \mathbf{var}\ me : B \bullet & \text{[semantics]} \\
& \qquad me := (B)le;\ (pds \bullet meI\ \Gamma\ B\ m\ b)(e);\ (B)le := me \\
& \quad \mathbf{end}\ \mathbf{com}]\!]\eta'\ \psi \\
= & [\![\Gamma, N \rhd \mathbf{var}\ me : B \bullet & [\neg(me\ \mathbf{is}\ C) \text{ holds after } me := (B)le] \\
& \qquad me := (B)le; \\
& \qquad \mathbf{if}\ \neg(me\ \mathbf{is}\ C) \rightarrow (pds \bullet meI\ \Gamma\ B\ m\ b)(e) \\
& \qquad [\!]\ me\ \mathbf{is}\ C \rightarrow (pds \bullet meI\ \Gamma\ B\ m\ b')(e) \\
& \qquad \mathbf{fi}; \\
& \qquad (B)le := me \\
& \quad \mathbf{end} : \mathbf{com}]\!]\eta'\ \psi \\
= & [\![\Gamma, N \rhd \mathbf{var}\ me : B \bullet & [me \text{ is not declared in } pds] \\
& \qquad me := (B)le; \\
& \qquad (pds \bullet \mathbf{if}\ \neg(me\ \mathbf{is}\ C) \rightarrow meI\ \Gamma\ B\ m\ b \\
& \qquad\qquad [\!]\ me\ \mathbf{is}\ C \rightarrow meI\ \Gamma\ B\ m\ b' \\
& \qquad\qquad \mathbf{fi})(e); \\
& \qquad (B)le := me \\
& \quad \mathbf{end} : \mathbf{com}]\!]\eta'\ \psi \\
= & [\![\Gamma, N \rhd (\mathbf{vres}\ me : B;\ pds \bullet & \text{[semantics]} \\
& \qquad \mathbf{if}\ \neg(me\ \mathbf{is}\ C) \rightarrow meI\ \Gamma\ B\ m\ b \\
& \qquad [\!]\ me\ \mathbf{is}\ C \rightarrow meI\ \Gamma\ B\ m\ b' \\
& \qquad \mathbf{fi})(e) : \mathbf{com}]\!]\eta'\ \psi
\end{array}$$

$$= [\![\Gamma, N \rhd (\eta'\ B\ m)((B)le, e) : \mathbf{com}]\!]\eta'\ \psi \qquad \text{[hypothesis]}$$
$$= [\![\Gamma, N \rhd le.m(e) : \mathbf{com}]\!]\eta'\ \psi \qquad \text{[assumption and semantics]}$$

Some of the above steps are justified by properties of commands. They are formalised by standard command laws, which we omit.

If the exact type of le is C, the proof is similar. □

The proof of Law 7 is based on the semantics of programs, and on the lemma above. The semantics of **super** is given by a copy-rule, but, since **super** is not present in b' and ops', the different definitions of m do not affect the result of applying such a rule. The typing environments defined by the programs are the same, since the methods available in the classes B and C are the same in both of them. The provisos guarantee that they are well-typed.

The environments η and η' defined by the programs in Law 7 are as in Lemma 6. The only final detail is that, in Lemma 6, we did not consider the fact that, if a subclass of C does not redefine m, then its semantics in the environment is also affected by the change. This generalisation of Lemma 6 is rather lengthier, but its proof is similar to that presented above.

4.4.3. Method elimination

We prove Law 11 using the following lemma. It states that the weakest precondition of commands is not affected by the elimination of a method that is not called in any command of any method in the environment. We define new environments Γ' and η' from the environments Γ and η by removing the parameter declaration and the parametrised command that defines the method m.

Lemma 7. *Let Γ and Γ' be such that $\Gamma'.meth\ N = \Gamma.meth\ N \backslash \{m \mapsto pds'\}$, but are otherwise identical. Consider also environments η and η' such that $\eta'\ N = \eta\ N \backslash \{m \mapsto (\mathbf{vres}\ me : N;\ pds' \bullet meI\ \Gamma'\ N\ m'\ c')\}$. If m is not used in η, then,*

$$[\![\Gamma, N \rhd c : \mathbf{com}]\!]\eta\ \psi = [\![\Gamma', N \rhd c : \mathbf{com}]\!]\eta'\ \psi$$

Proof. By induction. As remarked, the environments Γ' and η' are relevant when dealing with method calls. In this case, the called method cannot call the method that is being removed.

Case. $le.m'(e)$, with $m' \neq m$

$$\begin{aligned}
&[\![\Gamma, N \rhd le.m'(e) : \mathbf{com}]\!]\eta\ \psi \\
&= \vee_{N' \leq_\Gamma N''} \bullet le\ \mathbf{isExactly}\ N' \wedge \\
&\qquad [\![\Gamma, N \rhd (\eta\ N'\ m')\ ((N')le, e) : \mathbf{com}]\!]\eta\ \psi \qquad \text{[semantics]} \\
&= \vee_{N' \leq_{\Gamma'} N''} \bullet le\ \mathbf{isExactly}\ N' \wedge \\
&\qquad [\![\Gamma', N \rhd (\eta'\ N'\ m')\ ((N')le, e) : \mathbf{com}]\!]\eta'\ \psi \qquad \text{[hypothesis]} \\
&= [\![\Gamma', N \rhd le.m'(e) : \mathbf{com}]\!]\eta'\ \psi \qquad \text{[semantics].}
\end{aligned}$$

□

The proof of our law follows from the semantics of programs, and the above lemma. The proviso guarantees that the programs are well-formed, and the environments η and η' that record their methods are as in the proviso of the lemma.

4.4.4. Method call elimination

We can prove Law 13 as follows.

$$
\begin{aligned}
&[\![\Gamma, A \rhd le.m(e) : \mathbf{com}]\!]\eta\ \psi \\
&= \vee_{N' \leq_\Gamma C} \bullet\ le\ \mathbf{isExactly}\ N' \wedge \\
&\qquad [\![\Gamma, A \rhd (\eta\ N'\ m)\ ((N')le, e) : \mathbf{com}]\!]\eta\ \psi \qquad \text{[semantics]}
\end{aligned}
$$

Provided le is not **null** or **error**, le **isExactly** N' holds for some subclass N'' of C.

Moreover, since m is not redefined in the subclasses of C, then, by the way the environment is constructed, we have that $[\![\Gamma, A \rhd (\eta\ N'\ m)\ ((N')le, e) : \mathbf{com}]\!]\eta$ is the same for all subclasses N' of C, except only for the type of the extra me parameter. In other words, for all N', $\eta\ N'\ m$ is equal to $\eta\ C\ m$, except only that in the former, the type of me is N', and, in the latter, it is C.

More specifically, the value of $\eta\ N'\ m$ is $\mu(\mathbf{vres}\ me : N';\ pds \bullet meI\ \Gamma\ N'\ m\ c)$. As already mentioned, the command $meI\ \Gamma\ N'\ m\ c$ is a modified version of c that accesses the attributes of N' through the parameter me.

$$
\begin{aligned}
&\vee_{N' \leq_\Gamma C} \bullet le\ \mathbf{isExactly}\ N' \wedge \\
&\qquad [\![\Gamma, A \rhd (\eta\ N'\ m)\ ((N')le, e) : \mathbf{com}]\!]\eta\ \psi \\
&= le \neq \mathbf{null} \wedge le \neq \mathbf{error} \wedge [\![\Gamma, A \rhd (\eta\ N'\ m)\ ((N')le, e) : \mathbf{com}]\!]\eta\ \psi \\
&= le \neq \mathbf{null} \wedge le \neq \mathbf{error} \wedge \\
&\quad [\![\Gamma, A \rhd \mu(\mathbf{vres}\ me : N';\ pds \bullet meI\ \Gamma\ N'\ m\ c)((N')le, e) : \mathbf{com}]\!]\eta\ \psi
\end{aligned}
$$

The whole parametrised command $(\mathbf{vres}\ me : N';\ pds \bullet meI\ \Gamma\ N'm\ c)$ is regarded as a function of m. We use the unfold property of fixed points to proceed; based on this property, any occurrences of m in $meI\ \Gamma\ N'\ m\ c$ can be left untouched, if they are interpreted as calls, as indeed they are in the semantics.

$$
\begin{aligned}
&[\![\Gamma, A \rhd \mu(\mathbf{vres}\ me : N';\ pds \bullet meI\ \Gamma\ N'\ m\ c)((N')le, e) : \mathbf{com}]\!]\eta\ \psi \\
&= [\![\Gamma, A \rhd (\mathbf{vres}\ me : N';\ pds \bullet meI\ \Gamma\ N'\ m\ c)((N')le, e) : \mathbf{com}]\!]\eta\ \psi \\
&= [\![\Gamma, A \rhd \mathbf{var}\ me : N' \bullet \\
&\qquad me := (N')le;\ (pds \bullet meI\ \Gamma\ N'm\ c)(e);\ (N')le := me \\
&\qquad \mathbf{end} : \mathbf{com}]\!]\eta\ \psi \qquad \text{[semantics]} \\
&= [\![\Gamma, A \rhd (pds \bullet meI\ \Gamma\ A\ m\ c)[le/me](e) : \mathbf{com}]\!]\eta\ \psi \qquad \text{[semantics]} \\
&= [\![\Gamma, N' \rhd (pds \bullet c)[le/\mathbf{self}](e) : \mathbf{com}]\!]\eta\ \psi \qquad \text{[property of } meI\text{]}
\end{aligned}
$$

If we return to the semantics of method call, we can proceed as follows.

$$
\begin{aligned}
&le \neq \mathbf{null} \wedge le \neq \mathbf{error} \wedge \\
&[\![\Gamma, A \rhd \mu(\mathbf{vres}\ me : N';\ pds \bullet meI\ \Gamma\ N'\ m\ c)((N')le, e) : \mathbf{com}]\!]\eta\ \psi
\end{aligned}
$$

$$= le \neq \mathbf{null} \wedge le \neq \mathbf{error} \wedge [\![\Gamma, A \rhd (pds \bullet c)[le/\mathbf{self}](e) : \mathbf{com}]\!]\eta\ \psi$$

[result above]

$$= [\![\Gamma, A \rhd \{le \neq \mathbf{null} \wedge le \neq \mathbf{error}\}; (pds \bullet c)[le/\mathbf{self}](e) : \mathbf{com}]\!]\eta\ \psi$$

[semantics]

This concludes our proof.

The proof of Law 9 amounts to showing that the programs are well-typed. The semantics of a parametrised command does not depend on the type of its parameters, but on it being well-typed. Therefore, the main risk in the change of a parameter type is to render method calls ill-typed. If this is not the case, then the semantics of the calls depend on the value of the arguments. These are not changed in Law 9. The same comments apply to Law 10.

5. Completeness

After considering the soundness of our laws, we now show that the proposed set of laws is comprehensive. We do that by defining a reduction strategy, based on the laws, whose target is a normal form described in terms of a restricted subset of our language. This normal form uses classes and inheritance only to preserve the notion of subtyping; all classes have empty bodies, except **object**, which may include attribute declarations. This suggests that our laws are expressive enough to reason about the object-oriented structure of programs.

The definition of the normal form is as follows.

Definition 8. A program $cds \bullet c$ is in subtype normal form if it obeys the following conditions.

- Each class declaration in *cds*, except **object**, has an empty body.
- The **object** class may include only attribute declarations, each with either a primitive type or **object**.
- All local declarations in the main command c are declared with either a primitive type or **object**.
- No type cast is allowed in c. □

In a program in subtype normal form, the class **object** is explicitly included. All other classes may include the inheritance and subtype clause **extends**, but no declaration of methods, constructors or attributes is allowed. Fig. 1 illustrates that with part of the normal form for an arithmetic expressions interpreter structured according to the Interpreter design pattern [19]. The original interpreter is shown in Fig. 2.

Although this normal form preserves object-oriented features, namely the subtype hierarchy, object creation, and type tests, it is substantially close to an imperative program. The class **object**, the only one with explicitly declared elements, takes the form of a recursive record, as it contains only public attributes. As no methods are allowed, the main command c is similar to an imperative program, even though object creation and type test can still be used.

```
class object pub val : ℤ; leftExp, rightExp, exp : object end
class Expression end
class Value extends Expression end
class Integer extends Value end
class BinaryExpression extends Expression end
class Sum extends BinaryExpression end
class Interpreter end
• var int, n1, n2, s, v : object •
    {new Integer is Integer}; n1 := new Integer;
    {n1 is Integer};
    (val v : ℤ • {n1 is Integer}; n1.val := v)(x);
    {new Integer is Integer}; n2 := new Integer;
    {n2 is Integer};
    (val v : ℤ • {n2 is Integer}; n2.val := v)(y);
    {new Sum is Sum}; s := new Sum;
    {s is Sum}; {n1 is Integer}; {n2 is Integer};
    (val le, re : object •
      {s is Expression}; {s is BinaryExpression};
      if ¬(s is Sum) → {s is BinaryExpression}; {le is Expression};
                       s.leftExp := le;
                       {s is BinaryExpression}; {re is Expression};
                       s.rightExp := re
      [] s is Sum → {s is BinaryExpression}; {le is Expression};
                    s.leftExp := le;
                    {s is BinaryExpression}; {re is Expression};
                    s.rightExp := re
      fi)(n1, n2);
    ...
end
```

Fig. 1. Example program in normal form.

For the elimination of all object-oriented features, the natural normal form is the imperative subset of our language extended with recursive records. A reduction to such a form, which would yield a stronger completeness result, requires some sort of a mapping from an object to a relational model; an extra variable is necessary to keep the type information. The subtype normal form, however, is close to an imperative program, and some of the additional laws for a reduction to a pure imperative program are presented in Section 6.

It is important to note that the reduction of a program to normal form does not suggest a compilation process. Its sole purpose is to show that we have a comprehensive set of laws, which can be used to yield an equivalent program written with a small subset of constructs. Roughly, the less constructs the better the normal form; the same approach has been used for other programming paradigms [24].

```
class Expression
  meth eval ≙ (res x : Value • abort)
end

class Value extends Expression end

class Integer extends Value
  pri val : ℤ
  meth setVal ≙ (val v : ℤ • self.val := v)
  meth getVal ≙ (res v : ℤ • v := self.val)
  meth eval ≙ (res x : Value • x := self)
end

class BinaryExpression extends Expression
  pri leftExp, rightExp : Expression
  meth set ≙ (val le, re : Expression •
    self.leftExp := le;
    self.rightExp := re)
  meth getLeft ≙ (res e : Expression • e := self.leftExp)
  meth getRight ≙ (res e : Expression • e := self.rightExp)
end

class Sum extends BinaryExpression
  meth eval ≙ (res x : Value •
    var le, re : Expression; lint, rint : ℤ •
      self.getLeft(le); le.eval(le); ((Integer)le).getVal(lint);
      self.getRight(re); re.eval(re); ((Integer)re).getVal(rint);
      x := new Integer; x.setVal(lint + rint)
    end )
end

class Interpreter
  pri exp : Expression
  meth setExp ≙ (val e : Expression • self.exp := e)
  meth getExp ≙ (res e : Expression • e := self.exp)
  meth run ≙ (res x : Value • self.exp.eval(x))
end

• var int : Interpreter; n1, n2 : Integer; s : Sum; v : Value •
    n1 := new Integer; n1.setVal(x);
    n2 := new Integer; n2.setVal(y);
    s := new Sum; s.set(n1, n2);
    int := new Interpreter; int.setExp(s); int.run(v);
    ((Integer)v).getVal(z)
  end
```

Fig. 2. Example program to be normalised.

The reduction strategy involves the following major steps.

- Move all the attribute and method declarations in the classes of *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. We illustrate the process using the example program presented in Fig. 2, but the process is actually general. The program models an interpreter for a very simple expression language that includes only integers and sums. The global variables (inputs and outputs) of our program are x, y, and z of type integer; its result is the assignment to z of $x + y$. This is achieved by running the interpreter.

The class *Expression* contains only the method *eval*, which is supposed to return the result of the expression evaluation. It is actually defined in *Expression* as **abort**. Like a Java abstract class or interface, *Expression* defines a type, but it is not really intended to be used for the creation of objects. Subclasses of *Expression* model particular forms of expression and redefine *eval*.

The simplest expression is a value, which is modelled in our example by the class *Value*. We are also not supposed to create objects of this class, which again plays the role of an abstract class and does not redefine *eval*. Particular kinds of values are modelled as subclasses of *Value*; we present one such class: *Integer*. It contains one private attribute, *val*, which holds an integer value, set and get methods, and a redefinition of *eval* which simply returns itself as result.

Binary expressions are modelled by the class *BinaryExpression*. The operands are recorded by the attributes *leftExp* and *rightExp*. An example of a binary expression is modelled by the class *Sum*; its evaluation method takes each operand expression, evaluates it with a recursive call, and casts the result to an *Integer* object to get the integer value it holds. The result returned is the sum of the values so obtained.

The class *Interpreter* holds an expression in the attribute *exp*. Besides the set and get methods, this class includes a *run* method, which evaluates *exp*. The main command creates *Integer* objects $n1$ and $n2$ to hold x and y; they are used to create the expression s that represents $x + y$. An interpreter is created and initialised with s. By running it, a value v is obtained as the result of the evaluation of s; the value of z is determined by the integer in v.

5.1. Make attributes public

The first major step in our reduction strategy is to move up the attributes. Nonetheless, before that, we need to 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: Law 2 to make protected attributes public, and Law 3 to make private attributes public. In the strategy all the laws are applied from left to right. We need to exhaustively apply these two laws to all classes in *cds*.

In our example, only Law 3 is effectively applied to classes *Integer*, *BinaryExpression*, and *Interpreter*.

5.2. Move attributes up

After making all attributes public, we move them up to the **object** class using Law 4. Starting from the bottom of the class hierarchy, and moving upwards, the exhaustive application of this law moves all attributes to **object**. We assume that two distinct classes are not allowed to declare attributes with the same name. Therefore, name conflicts do not arise and the proviso of the law is valid. Our assumption imposes no significant restriction on our approach, since renaming can be used to meet this requirement.

In our example, the attribute *val* of *Integer* is moved to *Value*, from there to *Expression*, and then to **object**. Similarly, *leftExp* and *rightExp* of the class *BinaryExpression* goes up to *Expression*, and then to **object**. Finally, *exp* moves from *Interpreter* to **object**. The resulting program is sketched in Fig. 3. The class **object** is explicitly defined to include all the attributes of the original classes, which now do not declare any attributes; the main program is not touched. Part of the object-oriented design is lost, but the program still behaves as before. Recall that the purpose here is to establish the expressiveness of the laws. In practical applications of program transformation, like refactoring (see Section 7), the laws are applied in the reverse order.

5.3. Trivial cast introduction

To enable and simplify the next steps, we generate a uniform program text in which every non-assignable expression is cast. To see why this is necessary consider, for example, the method *eval* in class *Integer*:

$$\textbf{meth}\ eval \mathrel{\widehat{=}} (\textbf{res}\ x : Value \bullet x := \textbf{self})$$

We cannot move *eval* to the class *Expression*, as required in the subsequent steps of our strategy, since the type of **self** in *Expression* is *Expression*, and, therefore, the assignment $x := \textbf{self}$ would be ill-typed.

Law 16 is sufficient to introduce trivial casts to non-assignable expressions in an arbitrary program, including the main command. Fig. 4 presents part of the result of including all the needed casts in our example program. In the main command, the global variables, which have a primitive type, are not cast. Also, the existing cast is not touched. As a result, all non-assignable expressions are cast, either because they were in the original program, or because casts were introduced by the current step of our reduction strategy.

5.4. Introduce trivial method redefinitions

In this step, we further unify the program text, again to simplify the next steps. We introduce trivial method redefinitions using **super**. The methods of a class include those it declares and those it inherits. An inherited method may have a redefinition. If it does not, in this step, we provide a trivial redefinition that simply calls the method of the superclass.

We exhaustively apply Law 6, from left to right, considering all methods of all classes with subclasses. We start from **object** and move downwards in the class hierarchy.

```
class object
  pub val : ℤ; leftExp, rightExp : Expression; exp : Expression
end

class Expression
  meth eval ≙ (res x : Value • abort)
end

class Value extends Expression end

class Integer extends Value
  meth setVal ≙ (val v : ℤ • self.val := v)
  meth getVal ≙ (res v : ℤ • v := self.val)
  meth eval ≙ (res x : Value • x := self)
end

class BinaryExpression extends Expression
  meth set ≙ (val le, re : Expression •
    self.leftExp := le;
    self.rightExp := re)
  ...
end

...

• var int : Interpreter; n1, n2 : Integer; s : Sum; v : Value •
    n1 := new Integer; n1.setVal(x);
    n2 := new Integer; n2.setVal(y);
    s := new Sum; s.set(n1, n2);
    int := new Interpreter; int.setExp(s); int.run(v);
    ((Integer)v).getVal(z)
  end
```

Fig. 3. Example program—attributes up.

At the end, all classes have a definition for the methods they provide: either a trivial redefinition or that in the original program.

For our example, we include redefinitions for the method *eval* in the classes *Value* and *BinaryExpression*, and for the methods *set*, *getLeft*, and *getRight* in the class *Sum*. For instance, in class *Value*, we define **meth** *eval* ≙ **super**.*eval*. This is an abbreviation for **meth** *eval* ≙ (**res** *x* : *Value* • **super**.*eval*(*x*)).

5.5. Eliminate *super*

Before moving methods up, we need to make sure that their bodies do not contain references to **super**, otherwise the program semantics may not be preserved. This is

```
class object
  pub val : ℤ; leftExp, rightExp : Expression; exp : Expression
end
...

class Integer extends Value
  meth setVal ≙ (val v : ℤ • ((Integer)self).val := v)
  meth getVal ≙ (res v : ℤ • v := ((Integer)self).val)
  meth eval ≙ (res x : Value • x := (Integer)self)
end

...
• var int : Interpreter; n1, n2 : Integer; s : Sum; v : Value •
    n1 := (Integer)new Integer; ((Integer)n1).setVal(x);
    n2 := (Integer)new Integer; ((Integer)n2).setVal(y);
    s := (Sum)new Sum; ((Sum)s).set((Integer)n1, (Integer)n2);
    int := (Interpreter)new Interpreter;
    ((Interpreter)int).setExp((Sum)s); ((Interpreter)int).run(v);
    ((Integer)v).getVal(z)
  end
```

Fig. 4. Example program—trivial cast introduction.

because, when moving up a method that includes a method call of the form **super**.*m*, instead of referring to a method *m* of the immediate superclass *C*, we may end up referring to a method *m* of a superclass of *C*. Furthermore, when we move such a method to **object**, the resulting program is invalid, since **super** cannot appear in **object**.

Our approach for eliminating **super** relies on Law 12, which is a form of copy rule for calls of the form **super**.*m* in a class *C*, based on a declaration of *m* in the immediate superclass of *C*. Since in the previous step we introduced a definition for all methods available in a class, a method called via **super** is always declared in the immediate superclass of the class where the call appears. Therefore, we can exhaustively apply Law 12 to eliminate all method calls using **super**.

This elimination process starts at the immediate subclasses of **object** and moves downwards. As the methods of **object** cannot refer to **super**, and all attributes are already public at this point, the condition of Law 12 is valid for the immediate subclasses of **object**. After eliminating **super** from those classes, the condition will be valid for their immediate subclasses, and so on.

For our example, the result of the previous and of this step is shown in Fig. 5. The main command is not affected and is omitted for conciseness. All classes explicitly define all methods that are available for their objects directly, or rather, without using calls to the corresponding methods of the superclass. We use the fact that $(pds \bullet (pds \bullet c)(\alpha pds))$ is equivalent to $(pds \bullet c)$, in any context; this is convenient for our use of the abbreviated notation **meth** $m \mathrel{\widehat{=}}$ **super**.m.

```
class object
  pub val : ℤ; leftExp, rightExp : Expression; exp : Expression
end
class Expression
  meth eval ≙ (res x : Value • abort)
end
class Value extends Expression
  meth eval ≙ (res x : Value • abort)
end
class Integer extends Value
  meth setVal ≙ (val v : ℤ • ((Integer)self).val := v)
  meth getVal ≙ (res v : ℤ • v := ((Integer)self).val)
  meth eval ≙ (res x : Value • x := (Integer)self)
end
class BinaryExpression extends Expression
  meth set ≙ (val le, re : Expression •
    ((BinaryExpression)self).leftExp := (Expression)le;
    ((BinaryExpression)self).rightExp := (Expression)re)
  meth getLeft ≙ (res e : Expression •
    e := (Expression)((BinaryExpression)self).leftExp)
  meth getRight ≙ (res e : Expression •
    e := (Expression)((BinaryExpression)self).rightExp)
  meth eval ≙ (res x : Value • abort)
end
class Sum extends BinaryExpression
  meth set ≙ (val le, re : Expression •
    ((BinaryExpression)self).leftExp := (Expression)le;
    ((BinaryExpression)self).rightExp := (Expression)re)
  meth getLeft ≙ (res e : Expression •
    e := (Expression)((BinaryExpression)self).leftExp)
  meth getRight ≙ (res e : Expression •
    e := (Expression)((BinaryExpression)self).rightExp)
  meth eval ≙ (res x : Value •
    var le, re : Expression; lint, rint : ℤ •
      ((Sum)self).getLeft(le);
      ((Expression)le).eval(le); ((Integer)le).getVal(lint);
      ((Sum)self).getRight(re);
      ((Expression)re).eval(re); ((Integer)re).getVal(rint);
      x := (Integer)new Integer; ((Value)x).setVal(lint + rint)
    end )
end
class Interpreter
  meth setExp ≙ (val e : Expression •
    ((Interpreter)self).exp := (Expression)e)
    ...
end
```

Fig. 5. Example program—eliminate **super**.

As our example has not originally included any occurrence of **super**, it might give the impression that the previous two steps could be combined and carried out in a single step. This is, however, not the case when the original program already includes references to **super**.

5.6. Move methods up

After eliminating **super**, we can safely move methods up to **object**. This is justified by Laws 7 and 8. 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. The second should be applied when the method that we want to move is not a redefinition. We start applying Laws 7 and 8 from the bottom of the class hierarchy and move upwards towards **object**. The application of Law 7 introduces new occurrences of **self** in the program. These need to be cast as described in Section 5.3.

Using this strategy, the conditions for applying Law 7 are always valid: at this stage, all attributes are public and declared in **object**, and all method bodies do not use the **super** construct. This also explains why most of the conditions for applying Law 8 from left to right are valid. The only proviso we need to worry about are those related to the declaration of m in B and in its superclasses and subclasses. At this stage, every class redefines the methods in its superclass. So, if m is declared in C, but not in B, then it is not declared in any superclass of B. It is also not declared in any subclass of B, as, similarly to attribute names (see Section 5.2), we can assume that method names are only reused for redefinitions.

For our example, all the methods of *Interpreter* go directly to **object**. Fig. 6 presents the result of this and the next two steps of the reduction process. The method *eval* of *Integer* is combined with that of *Value*, and the resulting method is combined with the *eval* method of *Expression*. Similarly, the *eval* method of *Sum* is combined with that of *BinaryExpression*; the result is combined with the extended *eval* method in *Expression*. The result is a method definition that tests for all the possible dynamic types of an *Expression* object; this method declaration is moved up to **object**. The *set*, *getLeft*, and *getRight* methods of *Sum* and *BinaryExpression* are combined and moved all the way up to **object**. The program in Fig. 6 can be simplified if we consider that an alternation of the form **if** $b \rightarrow c \,[]\, \neg b \rightarrow c$ **fi** can be simplified to c, as this is the command to be executed regardless of the condition b (see Law 20); other laws of alternation can also be applied to combine and simplify nested alternations. Nevertheless, this is not relevant for the purpose of obtaining a program in our normal form.

5.7. Change type to *object*

Here we use the laws that formalise the fact that the types of attributes, variables, and parameters can be replaced with a supertype, if all non-assignable occurrences of these identifiers in expressions are cast: Laws 5, 9, 10 and 15. The exhaustive application of these laws, instantiating the type T' with **object**, allows the replacement of the types of all identifiers with the **object** class. The provisos of the laws are valid, since we already have casts in expressions, and every class is a subclass of **object**. Variables of primitive types,

```
class object
  pub val : ℤ; leftExp, rightExp, exp : object;
  meth setExp ≙ (val e : object •
    {self is Interpreter};
    {e is Expression}; self.exp := e)
  meth getExp ≙ (res e : object •
    {self is Interpreter}; {self.exp is Expression}; e := self.exp)
  meth run ≙ (res x : object •
    {self is Interpreter}; {self.exp is Expression}; self.exp.eval(x))
  meth eval ≙ (res x : object •
    {self is Expression};
    if ¬(self is BinaryExpression) →
        {self is Expression};
        if ¬(self is Value) → abort
        [] self is Value →
                {self is Value};
                if ¬(self is Integer) → abort
                [] self is Integer → {self is Integer}; x := self
                fi
        fi
    [] self is BinaryExpression →
        {self is BinaryExpression};
        if ¬(self is Sum ) → abort
        [] self is Sum →
            var le, re : object; lint, rint : ℤ •
              {self is Sum}; self.getLeft(le); {le is Expression};
              le.eval(le); {re is Expression}; re.eval(re);
              {re is Integer}; re.getVal(rint); {new Integer is Integer};
              x := new Integer; {x is Value}; x.setVal(lint + rint)
            end
        fi
    fi)
  meth setVal ≙ (val v : ℤ • {self is Integer}; self.val := v)
  meth getVal ≙ (res v : ℤ • {self is Integer}; v := self.v)
  ...
end
class Expression end
class Value extends Expression end
class Integer extends Value end
class BinaryExpression extends Expression end
class Sum extends BinaryExpression end
class Interpreter end
```

Fig. 6. Example program—move methods up, change type to **object**, and cast elimination.

including global variables, which we assume to be of a primitive type, are not affected by this reduction step.

5.8. Cast elimination

After the previous step, the trivial casts introduced previously are not trivial anymore, since the types of the identifiers were changed to **object**. Furthermore, the program may include arbitrary casts previously introduced by a developer. Therefore, the laws we use to eliminate casts are different from those we use to introduce them.

Since a type cast may occur arbitrarily nested in an expression, it is convenient to reduce expressions to a simple form, so that we can consider only a fixed number of patterns. This form is as defined in the BNF for expressions (see Section 2), with arbitrary expressions (denoted by *e*) replaced 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 in [42].

To deal with the elimination of casts in the remaining expression patterns, we use Laws 14 and 17, and others that are similar and omitted here. 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 object identifier is **object**, and that all methods and attributes have been moved to the **object** class. Therefore, the provisos of each law are always satisfied. As a result, the exhaustive application of these laws eliminates all casts in the program.

5.9. 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, we need only Law 13, which can be regarded as a version of the copy rule. The reason is that we deal with dynamic binding when we move methods up to the **object** class. In fact, there are no method redefinitions at this point, since all methods are in **object**.

In this step, we apply Law 13 exhaustively. Before doing so, however, we need to change all recursive calls of the form $le.m$. We eliminate them by defining the method *m* with the use of the recursive command **rec** $X \bullet c$ **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 all calls to a method are replaced with its body using Law 13, the method definition itself can be eliminated using Law 11. These two laws are sufficient to eliminate all methods. 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, since in every place where *m* is invoked, we can replace this invocation by the body which includes an invocation to *n*; this is no problem since *n* is still in scope. At this point there are no private attributes, method redefinitions, or references to **super**.

A relevant subset of the generated normalised program is presented at the beginning of the section, in Fig. 1. The main command obtained just before the elimination of methods is presented in Fig. 7.

```
var int : Interpreter; n1, n2 : Integer; s : Sum; v : Value •
  {new Integer is Integer}; n1 := new Integer;
  {n1 is Integer}; n1.setVal(x);
  {new Integer is Integer}; n2 := new Integer;
  {n2 is Integer}; n2.setVal(y);
  {new Sum is Sum}; s := new Sum; {s is Sum};
  {n1 is Integer}; {n2 is Integer}; s.set(n1, n2);
  {new Interpreter is Interpreter}; int := new Interpreter;
  {int is Interpreter}; {s is Sum}; int.setExp(s);
  {int is Interpreter}; int.run(v);
  {v is Integer}; v.getVal(z)
end
```

Fig. 7. Example program—main command, after cast elimination.

5.10. Summary of the strategy

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

Theorem 9 (Reduction Strategy). *An arbitrary program can be reduced to subtype normal form.*

Proof. From the application of the steps described in Sections 5.1–5.9, in that order, eventually renaming attributes and methods for avoiding naming conflicts. □

The proof of the above theorem is straightforward because the details of the strategy are discussed in each individual step.

Although our normal form reduction strategy provides reassurance as to the expressiveness of our set of laws, it might be surprising that some of the laws presented in Section 3 are not referenced here. This is a consequence of the fact that our subtype normal form preserves classes, attributes, type tests, and object creation. We decided to aim at this normal form because it is close to an imperative program and its reduction process is entirely algebraic; as mentioned before, reduction to a pure imperative form requires some sort of encoding of the object data model.

6. Class refinement

In addition to the equivalence laws used by the normal form strategy, for the transformation of programs we usually need to apply class refinement, which, as already mentioned, is a notion related to data refinement. The traditional techniques of data refinement deal with modules that encapsulate variables. In our approach, this is extended to consider hierarchies of classes whose attributes are not necessarily private: they can be protected or public. Law 22 below allows us to change attributes in a class, relating them with already existing attributes by means of a coupling invariant. The application of this law changes the bodies of the methods declared in the class and in its subclasses; it is a simulation law. The changes follow the traditional laws for data refinement [35].

Law 22. ⟨superclass attribute − coupling invariant⟩

class A **extends** C $adsA$; ops **end** cds'	$\preceq_{CI}$	**class** A **extends** C $adsC$; $CI(ops)$ **end** $CI(cds')$

provided

(↔) (1) CI refers only to public and protected attributes in $adsA$; (2) cds' only contains 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 attributes. The notation $CI(cds')$ indicates that CI acts on the class declarations of cds': it is applied to each of them. The application of CI to a class declaration changes the methods according to the laws of data refinement [35]: every guard may assume the coupling invariant and 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 $CI(ops)$. The coupling invariant CI must refer only to public and protected attributes in $adsA$, since it is used in the subclasses of A.

The law below, which can be used to introduce and eliminate attributes, is a direct application of the previous law.

Law 23. ⟨attribute elimination⟩

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

provided

(→) $B.a$ does not appear in ops;

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

If a private attribute is not in use inside the class in which it is declared, we can remove it. This can be proved with an application of Law 22; the attribute a should be regarded as the abstract attribute, there should be no concrete attributes, and the coupling invariant should be true. As already mentioned, the fact that simulation entails class refinement is addressed in [9].

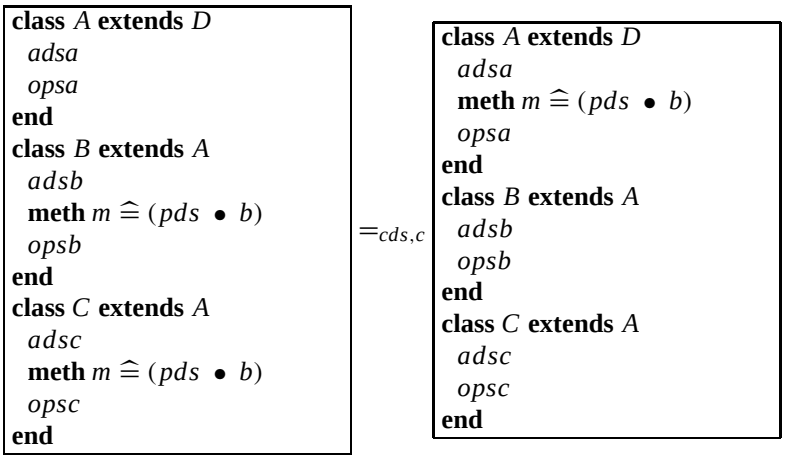
Law 22, together with laws of commands and of the object-oriented features, form a solid basis for proving more elaborate transformations of object-oriented programs. This is illustrated in the next section.

7. Formal refactoring

One of the main applications of the laws introduced in previous sections is the formal derivation of refactorings. In fact, developers often wish to use and define new refactorings. Our laws give them a basis for proving that the transformations they define preserve behaviour and, therefore, are indeed refactorings. In this section, we present some refactorings as refinement laws (a more extensive list can be found in [13]), and show how the laws previously introduced are used for justifying the refactoring laws.

In the refactoring laws, we explicitly present the conditions that must be satisfied in order to apply a refactoring. If the conditions are satisfied, the application of a refactoring to a program yields a new program that preserves the behaviour of the original one. As a first example, we present the refactorings ⟨Pull Up Method⟩ and ⟨Push Down Method⟩, which combine and organise redundant method declarations. Here 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. The class A that appears on the left-hand side of this law is the superclass of B and C, which declare a method m defined with the same parameters and body. As they have a common superclass and the method m is the same in both classes, we can move this method to the superclass. This helps maintenance as any modification will occur in just one method definition.

Refactoring 1. ⟨Pull Up/Push Down Method⟩



provided

- (↔) (1) **super** and private attributes do not appear in b; (2) **super**.m does not appear in $opsb$ or $opsc$; (3) m is not declared in any superclass of A in cds;
- (→) m is not declared in $opsa$, and can only be declared in a class N, for any $N \leq A$, if it has parameters pds;
- (←) (1) m is not declared in $opsb$ or $opsc$; (2) $N.m$, for any $N \leq A$ and $N \not\leq B$ or $N \not\leq C$, does not appear in cds, c, $opsa$, $opsb$ or $opsc$. □

The provisos are similar to those of Laws 7 and 8. Notice that if the method in B uses elements of B through **self**, this method could not be the same as that of C, which clearly does not have access to B elements. We also require that m is not defined in a superclass of A, as otherwise the method m available in A ends up being different when we apply this refactoring.

Proof. In order to derive the above refactoring, we assume that the provisos are valid and begin the derivation with the class declarations on the left-hand side.

We cast occurrences of **self** in b to A, so that later we can move the methods m to A. Every command in which there is an occurrence of **self** is preceded by **skip**, the specification statement : [**true**, **true**]. By the definition of assumptions, we can write this specification statement as {**true**}. By Law 18, we have that the expression **self is** A is true in classes B and C. Applying this law, from right to left, we obtain the assumption {**self is** A}. In this way, every command with occurrences of **self** is now preceded by the assumption {**self is** A}. By applying Law 17, from right to left, we cast every occurrence of **self** in classes B and C with A. The result is denoted by b'.

By using Law 8, we move the method m that is declared in class B to its superclass A, obtaining the following declarations.

class A **extends** D
  $adsa$
  meth $m \mathrel{\widehat{=}} pds \bullet b'$
  $opsa$
end

class B **extends** A
  $adsb$
  $opsb$
end

class C **extends** A
  $adsc$
  meth $m \mathrel{\widehat{=}} pds \bullet b'$
  $opsc$
end

The next step moves the method m declared in C to its superclass A. However, this method is already declared in A. So, we have to use Law 7, which allows us to move a redefined method from a subclass to its superclass. This introduces an alternation in the method declared in the superclass, yielding the following:

class A **extends** D
  $adsa$
  meth $m \mathrel{\widehat{=}} pds \bullet$
    if $\neg$(**self is** C) $\rightarrow$ b'
    [] **self is** C $\rightarrow$ b'
    fi
    $opsa$
end

class B **extends** A
  $adsb$
  $opsb$
end

class C **extends** A
  $adsc$
  $opsc$
end

The disjunction of the guards of the alternation we have introduced in the previous step is true, and the same command b' is guarded in both branches of the alternation. This allows us to apply Law 20 that reduces this alternation just to the command b'. Now we can remove the casts to A by applying Law 16, from right to left, obtaining the original command b. With this step we finish the proof of the refactoring ⟨Pull Up/Push Down Method⟩. □

In the previous derivation, we used only the laws of Section 3, however the class refinement law is often necessary, as illustrated in the proof of the following refactoring.

It allows us to move attributes from subclasses to their common superclass. Generalizing the typical refactoring presented in the literature, here the attributes may have different names, but their types have to be the same. We consider public attributes as this is the most general case; private and protected attributes can be made public using Laws 2 and 3.

Refactoring 2. ⟨Pull Up/Push Down Field⟩

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

provided

- (↔) cds contains no subclasses of B and C in which there are references to x and y;
- (→) (1) The attribute name z is not declared in $adsa$, $adsb$, $adsc$, nor in any subclass or superclass of A in cds and cds'; (2) and the attribute names x and y are not declared by $adsb$, $adsc$, nor by any subclass of A in cds; (3) $N.x$, 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;
- (←) (1) $N.z$, for any $N \leq A$, $N \not\leq B$, and $N \not\leq C$, does not appear in cds or c; (2) x (y) is not declared in $adsa$, $adsb$ ($adsc$), nor in any subclass or superclass of B (C) in cds and cds'. □

Again, the provisos guarantee that moving the attributes does not give rise to syntactic errors. 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.

Proof. Here we prove the derivation of this refactoring from left to right. The first step is to apply Law 4 twice. The applications of this law move the attributes x and y of classes B and C to their common superclass A; x and y are public as required by Law 4. The result is as follows.

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

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 $opsc$.

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 [35] transforms assignments of the form $t := \mathbf{self}.x$ into a 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 22, 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 [35]. 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 preconditions and postconditions. 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 $\mathbf{self}.x, \mathbf{self}.z := exp, exp$.

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 command laws [35].

The next step is the elimination of occurrences of abstract variables in subclasses of A. We diminish assignments $\mathbf{self}.x, \mathbf{self}.z := exp, exp$ 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 18 and 19 to simplify the conjunction of the coupling invariant. Inside B, the application of Law 18 reduces the test **self is** B to true. On the other hand, Law 19 allows us to reduce the test **self is** N, for a class N that is not a superclass or a subclass of B, to false. This simplifies the coupling invariant to the predicate $(\mathbf{true}\ \Rightarrow\ z = x) \wedge (\mathbf{false}\ \Rightarrow\ z = y)$, which is $z = x$. The specification statement, at this moment, is $t, z : [z = x, t = \mathbf{self}.x \wedge z = x]$ which is refined by the assignment $t := \mathbf{self}.z$, or rather $t := \mathbf{self}.x[z/x]$, a renaming of the original code in $opsb$. Guards must be rewritten using standard imperative command laws. We proceed in the same way with the commands of C that are augmented with concrete variables and that assume the coupling invariant.

The coupling invariant relates abstract and concrete variables via an equality between attribute names. Therefore, 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, nor in their subclasses, where they were originally declared, we can remove them from A. First we

apply Law 3, from right to left, in order to change the visibility of these attributes to private. Then we apply Law 23 that allows us to remove a private attribute that is not read or written inside the class in which it is declared. We proceed in the same way for C. We obtain the following class declarations.

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. □

Following the approach illustrated in this section, more than 25 refactorings have been formalised and proved [12,13]. This includes Extract Method, which is considered the Rubicon of refactoring tools. It can be derived by using two main laws. Law 11 introduces the declaration of the extracted method. Law 13 replaces the occurrence of the extracted command block by a **self** call to the extracted method; note that when le is **self**, the assumption statement in Law 13 reduces to skip. Before applying Law 13, we have to apply a couple of imperative laws for transforming a command block into a parameterized command application. Moreover, in order to deal with extracted blocks that refer to **super** or non-public attributes, we must apply Laws 2 and 3, which focus on attribute visibility, and then Laws 6 and 12, which focus on the semantics of **super**. These laws should be applied before and after Law 13; for instance, they are first applied to eliminate **super** (as in Sections 5.4 and 5.5) and then to introduce **super** back into the extracted method and other places.

8. Related work

Algebraic laws for other programming paradigms have been addressed before [4,24,35, 38,41,43]. Laws for small-grain object-oriented constructs have been considered [27,32], but with no completeness result. A great deal of work [16,17,20,26] has been carried out on transformations of design models in the unified modelling language (UML) [5], but those do not consider programming and behavioural specification constructs. Moreover, although some of those UML transformations are proved sound with respect to a formal semantics, as far as we know, no completeness result has been reported.

Previous works informally discuss refactorings for object-oriented programs [18], or formalize them with automation purposes only [37,39]. In particular, Opdyke [37] formally describes the conditions that have to be satisfied for applying a refactoring. Besides preconditions, Roberts [39] describes postconditions, which are useful for efficiently implementing mechanical support for composing refactorings. Kniesel and Koch [25] present an alternative approach for efficiently composing conditional program transformations, including refactorings. They consider the correctness of the preconditions of the composed refactorings, assuming the correctness of some basic refactorings. None of these works is concerned with the formal proof of refactorings, whereas we derive

refactorings from our algebraic laws, which, when compared with most refactorings, are simpler, separate concerns, and involve localised changes to the code.

Bergstein [3] presents a small set of primitive transformations which forms a basis for object-preserving class reorganizations, meaning that programs accept the same inputs and produce the same outputs. The set of transformations is shown to be correct, complete, and minimal. Bergstein's rule for abstracting common parts in a hierarchy can be seen as a derived rule in our framework, not a minimal one. We presented basic laws for moving attributes and methods up and down in a hierarchy. Bergstein's rule is similar to refactoring for pulling up and pushing down attributes and methods. There is no argument for completeness in terms of a normal form expressed in terms of a small set of object-oriented constructs. As a consequence, his notion of completeness does not cover all possible transformations that can be applied to object-oriented programs. He does not present transformations for dealing with type tests and casts as a consequence of changing the class hierarchy in his proof of completeness, nor does he deal with type changes. On the other hand, he goes beyond our work when alternation vertices, which are equivalent to abstract classes, are added or deleted from a hierarchy. In our framework, this corresponds to laws that change the class hierarchy, but we have not presented laws for dealing with the **extends** clause.

Utting [46] extends the refinement calculus to support a variety of object-oriented programming styles. He presents a model for multiple dispatch late binding, and then specialises this model to the single dispatch case. Both models are restricted to deal with modular reasoning, with a subtyping relation that is not attached to inheritance. He also distinguishes types, which contain procedures, from objects, which only contain data values; a particular model of objects is presented in which objects are tagged with their types. Utting, however, does not consider visibility control and recursive method calls, and, moreover, he does not propose programming laws.

Moore and Clement [34] present an algorithm for inferring inheritance hierarchies, resulting in creation or restructuring of hierarchies. The algorithm was implemented in a re-engineering tool for the dynamically typed language Self [45]. They define some criteria that must be met if a hierarchy is to be a representation of a structure that might be inferred from objects. The criteria involve sharing of features between objects, and the use of the fewest possible internal nodes in a hierarchy, among others. Their work is concerned with restructuring class hierarchies from a set of objects and their features. However, they are not concerned with the definition of laws that allow restructuring object-oriented systems.

Moore [33] considers automatic refactoring of methods for the language Self; this is supported by the Guru tool, which restructures inheritance hierarchies and refactors methods simultaneously. However, there is no formal proof of correctness, just an empirical argument. Refactoring of methods basically deals with factoring of expressions out of methods, and the consequent introduction of method calls in the place of the original expression. Refactoring of methods is performed as part of inheritance hierarchy restructuring, as refactoring of methods improves hierarchies by eliminating duplication of the expressions which it factors out. Some laws we presented here deal with moving methods up and down in a hierarchy. Method refactoring is considered in [12], along with class hierarchy refactorings.

Recently, Hoare and He's unifying theories of programming [23] have been used to give a semantics to an integration of timed CSP [40] and Object-Z [44] called TCOZ [30]. In that model, multiple inheritance, dynamic binding, and visibility are considered. In the spirit of the unifying theories of programming, we have a relational semantics in which predicates over observation variables are used to specify relations. Information about classes is recorded by observation variables that model a typing environment similar to that used here. A class denotes a program that updates the typing environment; objects are denoted by tuples that record type and attribute value information. Laws of object-oriented programming, and refinement, do not seem to be a concern of the authors at this stage. The semantics of TCOZ, however, contemplates aspects of time, concurrency, and communication.

A piece of work that is complementary to our research is the mechanisation of the normal form reduction strategy [29], as well as the mechanical proofs of some refactorings, using the Maude [31] term rewriting system. Each law is coded in Maude as a rewrite rule. The side conditions on the context are implemented as inductive definitions on the structure of our languages's syntax. The reduction strategy itself is directly implemented using Maude's rewriting engine. Nevertheless, because the laws are not Church–Rosser nor confluent, some additional conditions have been included to impose an order on the applications of the laws. Also, the laws are grouped into modules (one for each step of the reduction strategy), so that the laws of the first step are applied first, followed by the ones of the second step, and so on. This work provides extra confidence on the reduction strategy, on the refactoring laws, and on the usefulness of our laws. The example we used to illustrate the normal form reduction strategy is a small subset of a more substantial case study that was developed using the tool.

We are also using our laws to prove compilation rules [15] that support compiler construction in the algebraic style proposed in [42]. An abstract model of the Java virtual machine [28] is defined as the target normal form. The reduction process based on rules allows modular compilation, as in Java. This is possible because the compilation approach unifies source and target languages into a single framework. Each bytcode instruction is defined based on the effect it produces in the abstract model of the virtual machine; this effect is defined in terms of our language constructs. Therefore, in any step of the compilation process, every term is a fragment of a program in our language; the fact that it is already compiled or not is just an interpretation of the relevant syntax.

9. Conclusions

This article presents a comprehensive set of algebraic laws for object-oriented programming. It introduces the laws and explores their soundness, completeness, and application for deriving provably-correct refactorings. The article actually integrates an original soundness result with slightly extended and improved versions of previously published results on formal semantics, completeness, and derivation of refactorings [6,8].

Although the laws presented here are for a particular language, they are of more general utility. In particular, although our language has a copy semantics, whereas most practical object-oriented programming languages have a reference semantics, all the laws are valid

in the absence of sharing. Assignment laws such as

$$(le := e_1; le := e_2) = (le := e_2[e_1/le])$$

rely on copy semantics, because we might have several occurrences of le in e_2. The same happens to some laws related to specification statements and other imperative features, but these laws are not the focus of this article. On the other hand, the object-oriented programming laws considered here do not rely on copy semantics, except the simulation law for change of representation. To be valid with a reference semantics, such a law would have to consider pointer confinement issues as in [2], for example.

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. Our laws have been proved sound with respect to a weakest precondition semantics [8], as illustrated in the article.

Strategies for normal form reduction are usually adopted as a measure of completeness for a set of proposed laws, not as the final aim for a developer. 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 suggests that the laws, together with a law for refinement of class hierarchies, are expressive enough to derive program transformations that capture informal design practices such as refactorings. This was illustrated through the formalisation of two well-known refactorings as laws; this is more extensively explored in [12]. Moreover, our laws could also be used to derive behaviour preserving transformations that decrease object-oriented software qualities. Those might be useful, for example, for optimization.

Our strategy for normal form reduction might resemble a compilation process. The same impression is given by similar approaches for other programming paradigms, especially imperative programming, where we have the copy rule for eliminating procedures. In general, however, these strategies do not quite correspond to compilation. Most compilers do not use the copy rule to in-line the procedure body; control mechanisms are used to allow a single compilation of the body. Nevertheless, the copy rule is still a nice algebraic property of imperative programs. Analogously, we can capture the properties of dynamic binding independently of their use for compilation. Our laws can be used either for introducing polymorphism by dynamic binding or for compiling away dynamic dispatch by using type tests. By systematically showing how each object-oriented feature can be dealt with, or eliminated, in isolation, we uncover algebraic properties of that feature with respect to the more basic (imperative) language constructs. This provides an algebraic connection between the imperative and object-oriented paradigms, but does not suggest a compilation process.

Besides contributing to the formal verification of behaviour preserving transformations, the results presented here might be useful for the implementation of refactoring tools. Our laws suggest essential refactorings, which should be provided by tools that allow the user to compose existing refactorings to define new ones [25]. Similarly, executable languages for specifying new refactorings from scratch [7] should be able to express the laws presented here. A stronger completeness result, considering a normal form withouth object-oriented

constructs at all, could even suggest a minimal set of refactorings to be implemented by such tools, and constructs to be provided by those languages. This could lead to the creation of simpler tools and languages.

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

Perhaps another interesting issue of this research is the particular approach taken for the normal form reduction strategy: 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 at the bottom of the inheritance hierarchy. This, nevertheless, has proved to be unsuitable for a systematisation based on algebraic laws. The reason is that moving a single attribute or method from a superclass to a subclass has great contextual impact. Moving declarations up, on the other hand, is more controllable, as it causes less side-effects due to subtyping and dynamic binding. The particular approach adopted has allowed us to separate concerns to a great extent. For example, the elimination of method invocation (Law 13) has been dissociated from dynamic binding (Law 7), as well as from the behaviour of **super** (Laws 6 and 12).

Although our normal form reduction strategy has uncovered an interesting set of laws, it might be surprising that some obvious laws like class and attribute elimination were not necessary in our reduction process. This is a consequence of the fact that our subtype normal form preserves classes and attributes. An immediate topic for further research is the extension of the reduction strategy to target the imperative subset of the language, through the elimination of classes, attributes, object creation, and type test. In this case, the mentioned laws become necessary. We decided to separate these two reduction strategies because the one presented here is purely algebraic, whereas reduction to a pure imperative form requires some sort of encoding of the object data model.

For efficiency, it might be useful to implement a refactoring by carrying on transformations that not necessarily preserve behaviour. In fact, this is the approach followed by Fowler [18] for describing refactorings, and by developers that do not use refactoring tools. Our approach, however, might suggest that those refactorings could as well be expressed by composing refactorings (behaviour preserving transformations). It would be interesting to further investigate that.

We are also considering the extension of our language to include interfaces, exceptions, and pointers, and plan to further explore the application of our laws for teaching object-oriented programming to developers used to imperative languages. The laws could be used, for example, to show how to better modularize an imperative program by progressively introducing object-oriented constructs. This could be done by applying the laws in the opposite direction applied by the normal form reduction process.

Acknowledgements

We thank our collaborator David Naumann for many discussions that significantly contributed to the research reported here. He jointly defined with Ana Cavalcanti the

semantics of our language. We would also like to thank Ralf Lämmel, Günter Kniesel, and the anonymous referees for making several suggestions 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. We are partially supported by the Brazilian Research Agency, CNPq, grants 521994/96–9 (Paulo Borba), 520763/98-0 and 472204/2001-7 (Ana Cavalcanti), 521039/95–9 (Augusto Sampaio), and 680032/99-1 (DARE CO-OP project, jointly funded by CNPq PROTEM-CC and the National Science Foundation).

References

[1] R.J.R. Back, Procedural Abstraction in the refinement calculus, Technical Report, Ser. A, vol. 55, Department of Computer Science, Åbo, Finland, 1987.
[2] A. Banerjee, D. Naumann, Representation independence, confinement and access control, in: POPL2002, 2001.
[3] P.L. Bergstein, Object-preserving class transformations, in: OOPSLA'91 Conference Proceedings, ACM Press, 1991, pp. 299–313.
[4] R. Bird, O. de Moor, Algebra of Programming, Prentice-Hall, 1997.
[5] G. Booch, I. Jacobson, J. Rumbaugh, The Unified Modeling Language User Guide, Addison-Wesley, 1999.
[6] P.H.M. Borba, A.C.A. Sampaio, M.L. Cornélio, A refinement algebra for object-oriented programming, in: Luca Cardelli (Ed.), European Conference on Object-oriented Programming 2003, ECOOP 2003, Lecture Notes in Computer Science, vol. 2743, Springer-Verlag, 2003, pp. 457–482.
[7] F. Castor, P. Borba, A language for specifying Java transformations, in: V Brazilian Symposium on Programming Languages, Curitiba, Brazil, 23–25 May, 2001, pp. 236–251.
[8] A.L.C. Cavalcanti, D.A. Naumann, A weakest precondition semantics for refinement of object-oriented programs, IEEE Trans. Softw. Eng. 26 (8) (2000) 713–728.
[9] A.L.C. Cavalcanti, D.A. Naumann, Forward simulation for data refinement of classes, in: L. Eriksson, P.A. Lindsay (Eds.), Formal Methods—Getting IT Right, FME 2002, Lecture Notes in Computer Science, vol. 2391, Springer-Verlag, 2002, pp. 471–490.
[10] A.L.C. Cavalcanti, A.C.A. Sampaio, J.C.P. Woodcock, Procedures and recursion in the refinement calculus, J. Braz. Comput. Soc. 5 (1) (1998) 1–15.
[11] A.L.C. Cavalcanti, A.C.A. Sampaio, J.C.P. Woodcock, An inconsistency in procedures, parameters, and substitution the refinement calculus, Sci. Comput. Programming 33 (1) (1999) 87–96.
[12] M.L. Cornélio, Object-Oriented Refactorings and Patterns as Formal Refinements, Ph.D. Thesis, Informatics Center, Federal University of Pernambuco, Brazil, 2004, Ongoing work.
[13] M.L. Cornélio, A.L.C. Cavalcanti, A.C.A. Sampaio, Refactoring by transformation, in: Proceedings of REFINE'2002, Eletronic Notes in Theoretical Computer Science, vol. 70, 2002 (invited paper).
[14] E.W. Dijkstra, A Discipline of Programming, Prentice-Hall, 1976.
[15] A.A. Duran, A.L.C. Cavalcanti, A.C.A. Sampaio, A strategy for compiling classes, inheritance, and dynamic binding, in: K. Araki, S. Gnesi, D. Mandrioli (Eds.), FME 2003: Formal Methods, Lecture Notes in Computer Science, vol. 2805, Springer-Verlag, 2003, pp. 301–320.
[16] A. Evans, Reasoning with UML diagrams, in: Workshop on Industrial Strength Formal Methods, WIFT'98, IEEE Press, 1998.
[17] A. Evans et al., The UML as a formal modeling notation, in: J. Bézivin, P. Muller (Eds.), The Unified Modeling Language, UML'98—Beyond the Notation, Lecture Notes in Computer Science, vol. 1618, Springer-Verlag, 1999, pp. 336–348.
[18] M. Fowler, Refactoring, Addison-Wesley, 1999.
[19] E. Gamma et al., Design Patterns: Elements of Reusable Object-Oriented Software, Addison-Wesley, 1994.
[20] M. Gogolla, M. Richters, Transformation rules for UML class diagrams, in: J. Bézivin, P. Muller (Eds.), The Unified Modeling Language, UML'98—Beyond the Notation, Lecture Notes in Computer Science, vol. 1618, Springer-Verlag, 1999, pp. 92–106.
[21] J. Gosling, B. Joy, G. Steele, The Java Language Specification, Addison-Wesley, 1996.

[22] J. He, J. Bowen, Specification, verification, and prototyping of an optimized compiler, Form. Asp. Comput. 6 (1994) 643–658.
[23] C.A.R. Hoare, J. He, Unifying Theories of Programming, Prentice-Hall, 1998.
[24] C.A.R. Hoare et al., Laws of programming, Commun. ACM 30 (8) (1987) 672–686.
[25] G. Kniesel, H. Koch, Static composition of refactorings, Sci. Comput. Programming (2004) (this issue, doi: 10.1016/j.scico.2004.03.002).
[26] K. Lano, J. Bicarregui, Semantics and transformations for UML models, in: J. Bézivin, P. Muller (Eds.), The Unified Modeling Language, UML'98—Beyond the Notation, Lecture Notes in Computer Science, vol. 1618, Springer-Verlag, 1999, pp. 107–119.
[27] K.R.M. Leino, Recursive object types in a logic of object-oriented programming, in: C. Hankin (Ed.), 7th European Symposium on Programming, Lecture Notes in Computer Science, vol. 1381, Springer-Verlag, 1998.
[28] T. Lindholm, F. Yellin, The Java Virtual Machine Specification, Addison-Wesley, 1997.
[29] B.O. Lira, A.L.C. Cavalcanti, A.C.A. Sampaio, Automation of a normal form reduction strategy for object-oriented programming, in: Proceedings of the 5th Brazilian Workshop on Formal Methods, Gramado, Brazil, October, 2002, pp. 193–208.
[30] B. Mahony, J.S. Dong, Timed communicating object Z, IEEE Trans. Softw. Eng. 26 (2) (2000) 150–177.
[31] J. Meseguer, A logical theory of concurrent objects and its realization in the maude language, in: G. Agha, P. Wegner, A. Yonezawa (Eds.), Object-Oriented Programming, MIT Press, 1993, pp. 314–390.
[32] A. Mikhajlova, E. Sekerinski, Class refinement and interface refinement in object-oriented programs, in: J. Fitzgerald, C.B. Jones, P. Lucas (Eds.), FME'97: Industrial Benefit of Formal Methods, Lecture Notes in Computer Science, vol. 1313, Springer-Verlag, 1997, pp. 82–101.
[33] I. Moore, Automatic inheritance hierarchy restructuring and method refactoring, in: OOPSLA'96 Conference Proceedings, ACM Press, 1996, pp. 235–250.
[34] I. Moore, T. Clement, A simple and efficient algorithm for inferring inheritance hierarchies, in: R. Mitchell (Ed.), Proceedings of TOOLS-Europe'96, Paris, France, Prentice-Hall, Hertfordshire, UK, 1996, pp. 173–184.
[35] C.C. Morgan, Programming from Specifications, 2nd edition, Prentice-Hall, 1994.
[36] C.C. Morgan, K. Robinson, P.H.B. Gardiner, On the refinement calculus, Technical Monograph TM-PRG-70, Oxford University Computing Laboratory, Oxford, UK, 1988.
[37] W. Opdyke, Refactoring Object-oriented Frameworks, Ph.D. Thesis, University of Illinois at Urban Champaign, 1992.
[38] H.A. Partcsh, Specification and Transformation of Programs: a Formal Approach to Software Development, Texts and Monographs in Computer Science, Springer-Verlag, 1990.
[39] D. Roberts, Practical Analysis for Refactoring, Ph.D. Thesis, University of Illinois at Urban Champaign, 1999.
[40] A.W. Roscoe, The Theory and Practice of Concurrency, Prentice-Hall Series in Computer Science, Prentice-Hall, 1998.
[41] A.W. Roscoe, C.A.R. Hoare, The laws of occam programming, Theor. Comput. Sci. 60 (2) (1988) 177–229.
[42] A.C.A. Sampaio, An Algebraic Approach to Compiler Design, AMAST Series in Computing, vol. 4, World Scientific, 1997.
[43] S. Seres, M. Spivey, T. Hoare, Algebra of logic programming, in: ICPL'99, 1999.
[44] G. Smith, The Object-Z Specification Language, Kluwer Academic Publishers, 1999.
[45] D. Ungar, R.B. Smith, Self: the power of simplicity, in: OOPSLA'87 Conference Proceedings, ACM Press, 1987, pp. 227–242.
[46] M. Utting, An Object-Oriented Refinement Calculus with Modular Reasoning, Ph.D. Thesis, University of New South Wales, Kensington, Australia, 1992.