A Weakest Precondition Semantics for an Object-Oriented Language of Refinement

Ana Cavalcanti[1] and David A. Naumann[2]

[1] Departamento de Informática
Universidade Federal de Pernambuco, Po Box 7851 50740-540 Recife PE Brazil
Phone: +55 81 271 8430 Fax: +55 81 271 8438
alcc@di.ufpe.br www.di.ufpe.br/∼alcc
[2] Department of Computer Science
Stevens Institute of Technology, Hoboken NJ 07030 USA
naumann@cs.stevens-tech.edu www.cs.stevens-tech.edu/∼naumann

Abstract. We define a predicate-transformer semantics for an object-oriented language that includes specification constructs from refinement calculi. The language includes recursive classes, visibility control, dynamic binding, and recursive methods. Using the semantics, we formulate notions of refinement. Such results are a first step towards a refinement calculus.

Keywords: refinement calculi, semantic models, object-orientation, verification

1 Introduction

There has been extensive study of formal type-systems for object-oriented languages, and some study of formal specification, but formalization of development methods [BKS98, Lan95] lags behind both the language features and the informal methods presently used. This paper presents a semantic basis for formal development of programs in languages like Java and C++. Our language, called ROOL (for Refinement Object-oriented Language), is sufficiently similar to Java to be used in meaningful case studies and to capture some of the central difficulties, yet it is sufficiently constrained to make it possible to give a comprehensible semantics.

We assume the reader is familiar with basic concepts and terminology of object-oriented programming. We address the following challenging issues.

- Dynamic binding of methods means that the version of a method that will be invoked is determined only at run time. Such programs exhibit phenomena similar to higher-order imperative programs.
- Classes are important in practice for modularity, but they are complicated to model (for which reason many studies focus on instance-oriented subtyping).
- Object-oriented programs involve fine-grained control of visibility in terms of private, inherited, and public identifiers.

Our language has mutually recursive classes and recursive methods. We omit reference types, however. Pointers are ubiquitous in practice, but so are techniques

J. Wing, J. Woodcock, J. Davies (Eds.): FM'99, Vol. II, LNCS 1709, pp. 1439–1459, 1999.

to isolate deliberate sharing from the many situations where value semantics is preferable. Our object values are tuples with recursive nesting but no sharing. We leave pointers as an important but separate issue [AdB94].

Our work is part of a project that aims to extend to object-oriented programming the most widely-used and well-developed formal methods – those associated with Hoare logic and weakest preconditions. Because behavioural subclassing involves intermingled programs and specifications [LW94], it is natural to extend a refinement calculus [Mor94, BvW98]. As usual in refinement calculi, our semantics is based on weakest preconditions.

In the approach we adopt, commands denote functions on formulas. In isolation, purely syntactic transformer semantics is dubious. While our collaborators are developing a state-transformer semantics which will make it possible to prove operational soundness, we have taken the preliminary step of giving a set-theoretic semantics for predicate formulas and expressions, for the type-correctness results. Object states are represented by tuples of attribute values, and in general types denote sets of values. Methods are treated as procedures with a distinguished self parameter. Classes denote tuples of method meanings. Predicate formulas denote sets of states. The interplay between the value-oriented semantics of expressions and the formula-oriented semantics of commands is mediated by the semantics of formulas.

The semantics is based on a typing system. In the methodological literature simpler approaches are usually taken: there is a fixed global typing, or untyped variables are used and types are treated as predicates. A fixed global typing is unsuitable for formulating laws about subclasses and inheritance; and treating types as predicates risks inconsistency in the presence of higher-order phenomena. We employ techniques that have become standard in type theory and denotational semantics; the semantics is defined in terms of typing derivations, which provides convenient access to necessary contextual information.

We do not treat more advanced notions of subtyping than those in Java: we are interested in reasoning about type casts and tests as they are used in Java and its cousin languages. The typing system is similar to that of [Nau98b], and also to those used in typing algorithms, where subsumption is incorporated into rules for different constructs rather than being present as a general rule. Nonetheless, soundness of our definitions is not at all obvious, due to subtleties of modelling dynamic binding as well as mutually recursive classes. In this paper, we disallow mutually recursive methods, which lets us use a simpler, though non-trivial, well-ordering to show the semantics is well defined.

The main contribution of this paper is the extension of standard weakest precondition semantics to a Java-like language with classes, visibility, dynamic binding, and recursion. We give basic results on soundness of the definitions, define notions of program and class refinement, and show that the constructors of ROOL are monotonic with respect to refinement. Our semantics is being used in ongoing research on practical specification and verification. For reasons of space we omit many definitions and proofs that appear in [CN99].

$e \in Exp$	$::=$	**self** | **super** | **null** | **new** N	
		| 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
$\psi \in Pred$	$::=$	e | e **isExactly** N	boolean expression, exact type test
		| $(\vee\ i \bullet \psi_i)$ | $\psi \Rightarrow \psi$ | $\forall x : T \bullet \psi$	
$c \in Com$	$::=$	$le := e$ | $c;\ c$	multiple assignment, sequence
		| $x : [\psi, \psi]$	specification statement
		| $pc(e)$	parameterized command application
		| **if** $[]i \bullet \psi_i \rightarrow c_i$ **fi**	alternation
		| **rec** $Y \bullet c$ **end** | Y	recursion, recursive call
		| **var** $x : T \bullet c$ **end**	local variable block
		| **avar** $x : T \bullet c$ **end**	angelic variable block
$pc \in PCom$	$::=$	$pds \bullet c$	parameterization
		| m | $le.m$	method calls
$pds \in Pds$	$::=$	$\varnothing$ | pd | $pd;\ pds$	parameter declarations
$pd \in Pd$	$::=$	**val** $x : T$ | **res** $x : T$ | **vres** $x : T$	

Table 1. Expressions, selected predicates, commands, and parameterized commands.

2 Language

The imperative constructs of ROOL are based on the language of Morgan's refinement calculus [Mor94], which extends Dijkstra's language of guarded commands. Specifications are regarded as commands; we use the word command to refer to specifications, commands in the traditional sense, and hybrids where programming structures and specifications are mixed.

Data types T are the types of attributes, local variables, method parameters, and expressions. They are either primitive (**bool**, **int**, and others) or class names N. Primitives may include functional types such as arrays of integers.

The expressions e are generated by a rule in Table 1. We assume that x stands for a variable identifier, and f for a literal or built-in function. Built-ins should include primitive predicates like equality. The update $(e_1;\ x : e_2)$ denotes a fresh object copied from e_1 but with the attribute x mapped to a copy of e_2. Attribute selection $e.x$ is a run-time error in states where e denotes **null**, and $(N)e$ is an error if the value of e is not of dynamic type N. The type test e **is** N checks whether non-null e has type N; it is false if e is **null**, like `instanceof` in Java. The predicates ψ of ROOL include formulas of first-order logic, program expressions of type **bool**, and exact type tests e **isExactly**N.

We identify a subset Le of Exp; these left-expressions can appear as the target of assignments and method calls, and as result and value-result arguments.

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

Assignments to **self** and method calls with **self** as a result or value-result argument would never appear in user programs, but they are used in the seman-

tics. We allow le in $le := e$ to range over finite non-empty sequences of left-expressions, and e over corresponding lists.

For alternation we use an informal indexed notation for finite sets of guarded commands. Specification statements are as in [Mor94]. Methods are defined using procedure abstractions in the form of Back's parameterized commands **val** $x : T \bullet c$, **res** $x : T \bullet c$, or **vres** $x : T \bullet c$ [CSW99]. These correspond to parameter passing by copy: call-by-value, by-result, and by-value-result, respectively. In each case, x stands for a finite sequence of variable identifiers, T for a corresponding list of types, and c for a command. We use x to stand both for a single variable and for lists of variables; the context should make clear which one is meant. The same comment applies to our uses of e and T.

A parameterized command can be applied to a list of arguments to yield a command. A method call is a parameterized command. A call m refers to a method of the current object; a call $le.m$ refers to a method associated with the object that is the current value of le. We do not allow method calls $e_1.m(e)$ where e_1 is a general expression, because it is convenient in the semantic definitions that the object is named by a variable (the same is done in [AL97]). If e_1 is not a left-expression, $e_1.m(e)$ is equivalent to **var** $x : T \bullet x := e_1;\ x.m(e)$ **end**, where x is fresh. This is not an adequate replacement for $e_1.m(e)$, when e_1 is a left-expression, because it does not make persistent the changes to e_1. However, calls of the form $le.m(e)$ are available in ROOL.

A program is a sequence of class declarations followed by a command.

$$
\begin{array}{lll}
\mathit{Program} & ::= & \mathit{cds} \bullet c \\
\mathit{cds} \in \mathit{Cds} & ::= & \varnothing \mid \mathit{cd}\ \mathit{cds} \\
\mathit{cd} \in \mathit{Cd} & ::= & \textbf{class}\ N_1\ [\textbf{extends}\ N_2] \\
 & & \quad \textbf{pri}\ x_1 : T_1;\ \textbf{prot}\ x_2 : T_2;\ \{\textbf{meth}\ m \mathrel{\widehat{=}} (\mathit{pds} \bullet c)\ \textbf{end}\}^*; \\
 & & \textbf{end}
\end{array}
$$

A class declaration cd introduces a class named N_1. The optional **extends**-clause determines the immediate superclass of N_1. In its absence, N_1 extends **object**, which has no attributes or methods. The **pri** and **prot** clauses introduce the private and protected attributes of N_1 (recall that x_1 and x_2 can be lists). The visibility mechanism is similar to that of Java: private attributes are visible just inside the class, and protected attributes are visible in the class and in its subclasses. Following the **pri** and **prot** clauses, there is a list of method declarations. The method introduced by **meth** $m \mathrel{\widehat{=}} (\mathit{pds} \bullet c)$ **end** is named m; its body is the parameterized command $(\mathit{pds} \bullet c)$. All methods are considered to be public.

3 Typing

Besides the data types T, other phrase types θ are available for predicate formulas, commands, parameterized commands, and complete programs.

$$\theta ::= T \mid \textbf{pred} \mid \textbf{com} \mid \textbf{pcom}(\mathit{pds}) \mid \textbf{program}$$

The types of phrases, in the context of a collection of class declarations, a specific class, and some method parameter and local variable declarations, are given by the typing relation $\rhd$. For example, $\Gamma, N \rhd c : \mathbf{com}$ asserts that c is a command that can appear in the body of a method in class N. Here Γ is a typing environment; it records class declarations as well as locals for c: the attributes visible in N, and method parameters and local variables in scope. Similarly, $\Gamma, N \rhd e : T$ asserts that in a method of N, e is an expression of type T.

3.1 Typing Environment

We assume the existence of two disjoint sets of names: the set *CName* of class names and the set *LName* of local names. A local may be either an attribute, a method, a method parameter, or a local variable. We also distinguish two class names: **object** and **main**. The former is the superclass of all classes. The latter does not refer to a class itself, but to the main part of a complete program.

A typing environment Γ is a record with six fields: *attr*, *meth*, *vis*, *cnames*, *supcls*, and *locals*. The first, *attr*, is a finite partial function $CName \twoheadrightarrow LSignature$. An *LSignature* associates a local name with a type: $LSignature = LName \twoheadrightarrow Type$. The field *attr* records the names and types of all declared and inherited attributes of every declared class. Similarly, *meth* records the names and signatures of all declared and inherited methods of the known classes: *meth* has type $CName \twoheadrightarrow MDecs$ where $MDecs = LName \twoheadrightarrow Pds$.

The third field of a typing environment, *vis*, records the visibility of the attributes of the declared classes: *vis* has type $CName \twoheadrightarrow (LName \twoheadrightarrow Visibility)$ where $Visibility = \{pri, prot, ipri\}$. If, we have that for an attribute x of a class N, $vis\ N\ x = pri$, then x is a private attribute of N that was declared (and not inherited) by N; the inherited private attributes of N are associated to *ipri*. Finally, *prot* refers to the protected (either inherited or declared) attributes.

The *cnames* field of a typing environment is a set containing the name of all declared classes: $cnames = \mathrm{dom}\, attr = \mathrm{dom}\, meth = \mathrm{dom}\, vis$. The distinguished class name **object** is supposed to be in *cnames*, while **main**, which does not refer to a class, is supposed not to be in *cnames*. Moreover, the class **object** is associated to the empty signature in both *attr* and *meth*.

The *supcls* field of a typing environment associates a class name to the name of its immediate superclass: *supcls* has type $CName \twoheadrightarrow CName$. All declared classes have a direct superclass: either a class mentioned explicitly in their declarations or **object**. On the other hand, **object** itself does not have a superclass. Furthermore, a superclass is a declared class and the inheritance relationship is not allowed to have circularities. The subtype relation $\leq_\Gamma$ is defined by $T_1 \leq_\Gamma T_2 \Leftrightarrow (T_1, T_2) \in (\Gamma.supcls)^+ \vee T_1 = T_2$.

The last component of a typing environment, *locals*, is an *LSignature* that records the types of the visible atributes of the current class, and of any method parameter and local variables in scope. The attributes are also recorded in *attr*; this redundancy simplifies typing rules. The classes referred to in the signatures in the range of either *attr* or *meth* and in *locals* must be declared.

$$\frac{N \neq \textbf{main}}{\Gamma, N \rhd \textbf{self} : N} \qquad \frac{N' \in \Gamma.cnames}{\Gamma, N \rhd \textbf{new}\ N' : N'} \qquad \frac{\Gamma, N \rhd e : N' \quad N'' \leq_\Gamma N'}{\Gamma, N \rhd e\ \textbf{is}\ N'' : \textbf{bool}}$$

$$\frac{\Gamma, N \rhd e : N' \quad N'' \leq_\Gamma N'}{\Gamma, N \rhd (N'')e : N''} \qquad \frac{\Gamma, N \rhd e : N' \quad \Gamma.attr\ N'\ x = T \quad visib\ \Gamma\ N'\ N\ x}{\Gamma, N \rhd e.x : T}$$

$$\frac{\Gamma \rhd e : \textbf{bool}}{\Gamma \rhd e : \textbf{pred}} \qquad \frac{\Gamma \rhd \psi_i : \textbf{pred} \quad \text{for all } i}{\Gamma \rhd (\vee\ i \bullet \psi_i) : \textbf{pred}} \qquad \frac{\Gamma;\ x : T \rhd \psi : \textbf{pred}}{\Gamma \rhd \forall\, x : T \bullet \psi : \textbf{pred}}$$

$$\frac{\Gamma, N \rhd e : N' \quad N'' \leq_\Gamma N'}{\Gamma, N \rhd e\ \textbf{isExactly}\ N'' : \textbf{pred}} \qquad \frac{(\Gamma;\ x : N'') \rhd \psi : \textbf{pred} \quad N'' \leq_\Gamma N'}{(\Gamma;\ x : N') \rhd x\ \textbf{isExactly}\ N'' \wedge \psi : \textbf{pred}}$$

Table 2. Typing of selected expressions and predicates.

A typing $\Gamma, N \rhd phrase : \theta$ holds just if it is well formed and is derivable using the rules to be presented in the sequel. Well formedness is characterised by three properties. First, Γ has to satisfy the conditions above for environments. Secondly, the current class must be declared: $N \neq \textbf{main} \Rightarrow N \in \Gamma.cnames$. Thirdly, $\mathrm{dom}\,\Gamma.locals$ should include all visible attributes of N, i.e. the declared private and the declared and inherited protected attributes – all but the inherited private ones. We assume that no parameter or local variable has the same name as an attribute of the class. If N is **main** there are no restrictions on $\Gamma.locals$, which contains only parameters and local variables.

3.2 Expressions and Predicates

Typing rules for some expressions and predicates are in Table 2. The boolean expression e **is** N'' is well-typed when the type of e is a superclass of N''. The type of $e.x$ is that of the x attribute of the class of e, provided this attribute is visible from the current class. In a hypothesis like $\Gamma.attr\ N'\ x = T$, which involves partial functions, we mean that the expressions are defined and equal. Visibility is considered in $visib\ \Gamma\ N'\ N\ x$, a condition stating that, according to Γ, x is an attribute of N' visible from inside N. We define $visib\ \Gamma\ N'\ N\ x$ to hold if and only if $N \leq_\Gamma N'$, $\Gamma.vis\ N\ x \neq ipri$, and $N \neq N' \Rightarrow \Gamma.vis\ N\ x \neq pri$. The attributes visible in N are those declared in N itself and those inherited from its superclasses that are not private.

A typing $\Gamma, N \rhd \psi : \textbf{pred}$ is for a predicate on the state space of a method in class N, where $\Gamma.locals$ declares local variables, parameters, and attributes to which ψ may refer. We say ψ is typable in Γ, N, meaning $\Gamma, N \rhd \psi : \textbf{pred}$ is derivable; similarly for command typings later. In some rules we omit the current class N because it does not change throughout the rule. The environment $\Gamma;\ x : T$, differs from Γ just in the *locals* field: we define $(\Gamma;\ x : T).locals$ to be $\Gamma.locals \oplus \{x \mapsto T\}$, where $\oplus$ denotes function overriding.

The rule for **isExactly** is similar to the rule for **is**, but we also need coercion rules for **is** and **isExactly** in combination with $\wedge$ and $\Rightarrow$. As an example, consider

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

$$\frac{\Gamma.meth\ N\ m = pds}{\Gamma, N \rhd m : \mathbf{pcom}(pds)} \qquad \frac{\Gamma, N \rhd le : N' \quad \Gamma.meth\ N'\ m = pds}{\Gamma, N \rhd le.m : \mathbf{pcom}(pds)}$$

$$\frac{\Gamma \rhd le : T \quad \Gamma \rhd e : T' \quad T' \leq_\Gamma T \quad sdisjoint\ le}{\Gamma \rhd le := e : \mathbf{com}}$$

$$\frac{\Gamma \rhd pc : \mathbf{pcom}(\mathbf{val}\ x : T) \quad \Gamma \rhd e : T' \quad T' \leq_\Gamma T}{\Gamma \rhd pc(e) : \mathbf{com}}$$

$$\frac{\Gamma \rhd pc : \mathbf{pcom}(\mathbf{vres}\ x : T) \quad \Gamma \rhd le : T \quad sdisjoint\ le}{\Gamma \rhd pc(le) : \mathbf{com}}$$

$$\frac{\Gamma \rhd \psi_i : \mathbf{pred} \quad \Gamma \rhd c_i : \mathbf{com}}{\Gamma \rhd \mathbf{if}\ []i \bullet \psi_i \rightarrow c_i\ \mathbf{fi} : \mathbf{com}} \qquad \frac{(\Gamma;\ x : T) \rhd c : \mathbf{com}}{\Gamma \rhd (\mathbf{var}\ x : T \bullet c\ \mathbf{end}) : \mathbf{com}}$$

Table 3. Typing of selected parameterized commands and commands.

a class *Pt* of points and an extended class *Cpt* with an added attribute *color*. The predicate $(x.color = red)$ is not typable in a context $(\Gamma;\ x : Pt), N$. However, if for instance $(\Gamma, x : Cpt), N \rhd x.color = red : \mathbf{pred}$, we would like the predicate x **is** $Cpt \Rightarrow x.color = red$ to be typable in a context where x has type *Pt*. Using only the separate rules for **is** and $\Rightarrow$, it is not typable as such; but it can be typed by a coercion rule for **is** like the one for **isExactly** in Table 2. Rules like this allow the derivation of typings in more than one way, but the semantic definitions ensure that the meaning is independent of derivation (Lemma 6).

Substitution on formulas and expressions is standard, but it is worth noting that the free variables of $e.x$ are those of e. This is because x is in the role of an attribute name.

3.3 Parameterized Commands, Commands, and Programs

Typing rules for selected commands and parameterized commands are presented in Table 3. The type of a parameterized command records its parameter declarations. In the cases of m and $le.m$, the declarations are recorded in the *meth* attribute of the typing environment. Of course, $le.m$ is well-typed only if the type of le is a class with a method m. An omitted rule deals with multiple parameters.

To preclude aliasing, the rule for assignment stipulates *sdisjoint le*. This means that, if le is a list, then no member of le is a prefix of another, after deleting **self**. For example, neither $x, x.y$ nor $x, \mathbf{self}.x$ is *sdisjoint*, but $x, y.x$ is. If pc is a parameterized command with parameter declaration $\mathbf{val}\ x : T$, then $pc(e)$ is well-typed when the type of e is a subtype of T. If x is a result or a

value-result parameter, then *pc* can only be applied to *sdisjoint* left-expressions. If x is a result parameter, $pc(le)$ is well-typed when T is a subtype of the type of *le*. When x is a value-result parameter, these types have to be the same.

A complete program $cds \bullet c$ is well-typed in an environment where only global variables x are in scope, just when c is well-typed in the environment Γ determined by *cds* and $x : T$, and considering that the current class is **main**.

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

The fields of the environment $\varnothing$ are all empty, so that in $(\varnothing;\ x : T)$ the only non-empty field is *locals*, which records the global variables $x : T$ of the program. The function *VDecs* extracts information from and checks a sequence of class declarations. In the environment determined by this function, the classes are associated with both its declared and inherited methods. The condition $Vmeth\ \Gamma\ cds$ checks that the method bodies in *cds* are well-typed in the environment Γ. The method bodies are checked in an environment that includes their signatures, so recursive calls are appropriately dealt with. Mutually recursive calls, however, are not allowed. This is verified by the condition $nomrec\ \Gamma\ cds$.

The absence of mutual recursion between methods can not be checked as easily as the absence of mutual recursion between procedures of a traditional imperative program. By way of illustration, consider classes C, D and C'; the class C has an attribute a of type integer and a method $m1$ that, for instance, increments a by 1. The class D has an attribute c of class C, a method $m2$ with a call $c.m1()$, and some other methods. There is no mutual recursion, as $m1$ does not call $m2$. However, suppose that in a subclass C' of C we declare an attribute $d : D$ and redefine $m1$ introducing a call $d.m2()$. Now, if the private attribute c of D happens to have dynamic type C' when $m2$ is called, then mutual recursion will arise. To rule out mutual recursion, we require that if a method $m2$ calls a method $m1$ then neither $m1$ nor any of its redefinitions calls $m2$.

3.4 Properties of Typing

To a large extent, a context determines the type of an expression; an exception is **null**, for which we have $\Gamma, N \rhd \mathbf{null} : N'$ for all N, N'. Some phrases, however, can be typed in many contexts. For example, considering again the class *Pt* and its subclass *CPt*, the command $x := \mathbf{new}\ CPt$ can be typed in $\Gamma;\ x : Pt$ and also in $\Gamma;\ x : CPt$. Nonetheless, an expression typing does determine a derivation.

Lemma 1. *For all typings $\Gamma, N \rhd e : T$, there is at most one derivation.*

For predicates, the coercion rules make it possible to derive certain typings in more than one way. For example, if ψ is derivable in $(\Gamma;\ x : N'), N$, then $(\Gamma;\ x : N'), N \rhd x\ \mathbf{is}\ N' \Rightarrow \psi : \mathbf{pred}$ can be derived using the rules for **is** and $\Rightarrow$, or using a coercion rule; more on this later.

To show type-correctness of method calls we need the following result. It is similar to the coercion rules, but in fact it does not depend on them.

Lemma 2. *The following rule is admissible, in the sense that the conclusion is derivable if the hypothesis are.*

$$\frac{\Gamma, N \rhd \psi_{N'} : \mathbf{pred} \quad \textit{for all } N' \leq_\Gamma N \qquad N \neq \mathbf{main}}{\Gamma, N \rhd (\vee_{N' \leq_\Gamma N} \bullet \mathbf{self\ isExactly}\ N' \wedge \psi_{N'}) : \mathbf{pred}}$$

Many type systems include a rule of subsumption, but this would make coherence (Lemma 6) harder to prove. The useful effects of subsumption are built-in to the typing rules.

4 Semantics

Since ROOL includes infeasible (discontinuous) constructs, recursive class definitions cannot be interpreted by standard domain-theoretic techniques. We deal with recursive classes by separating attributes from methods, so the domain equations to be solved are simple "polynomials" involving first-order records.

The semantics $[\![\Gamma, N \rhd \mathit{phrase} : \theta]\!]$ of each derivable typing, except method call, is defined as a function of the semantics of its constituent phrases. Most typing rules have a corresponding semantics which we present in a form that mimics the typing rule, to remind the reader of the typings for constituent phrases and any side conditions on those typings. Some phrases are treated indirectly through syntactic transformations described later.

Method calls are the most complicated part of the semantics, and they are discussed last. Semantics of method call goes beyond recursion on typing derivations. Moreover, we need the semantics to be defined for any phrase typable in an extended typing system defined as follows. The first change is that constraints involving the predicate *visib* are dropped. The second is that, in the rules for type tests and type casts, the subtyping constraint is dropped.

Semantically, e **is** N'', for example, can only hold if N'' is a subtype of the declared type of e. Nevertheless, this constraint is incompatible with the semantics of assignment, which as usual is interpreted by substitution. Consider, for instance, a context Γ with locals $x : Pt, z : SCPt$ where $SCPt \leq_\Gamma CPt \leq_\Gamma Pt$. In this context, both $x := z$ and x **is** CPt are typable, but substitution yields z **is** CPt which is not typable in the original system because $CPt \not\leq_\Gamma SCPt$.

All results in Section 3.4 hold for both typing systems. The constraints we drop are natural for user programs, but such constraints are not found in semantic studies. Although user specifications would not refer to non-visible attributes, such predicates can be used in proofs of laws.

4.1 Environments, Data Types, and States

An environment is a finite partial function $\mathit{CName} \twoheadrightarrow (\mathit{LName} \twoheadrightarrow \mathit{PCom})$ that for a given class associates method names to parameterized commands. As formalized later on, the parameterized command corresponding to a method will be that given in its declaration, with an extra parameter *me*. This parameter

is passed by value-result and provides the attributes of the object upon which the method is called. This facilitates interpretation of the method body in the context of its calls.

For a given typing environment Γ, we define the set $[\![\Gamma]\!]$ of environments compatible with Γ. The environments η in $[\![\Gamma]\!]$ are characterized by the following conditions. First, $\mathrm{dom}\,\eta = \Gamma.cnames$. Also, $\mathrm{dom}(\eta\ N) = \mathrm{dom}(\Gamma.meth\ N)$ for all $N \in \mathrm{dom}\,\eta$. Finally, the parameter declarations are those recorded in $\Gamma.meth$, along with the extra value-result parameter me; for all N, m there is some c such that $\eta\ N\ m = (\mathbf{vres}\ me : N;\ \Gamma.meth\ N\ m \bullet c)$. In the environments we construct later, c is derived from the declared body as a fixpoint.

In addition to the environment, the semantic function for expressions also takes a state as argument. A state assigns type-correct values to the attributes of the current object, and to the parameters and local variables. It also records the class of the current object. Object values, like states, assign values to attribute names. Our formalization begins with a universal set of untyped values, which are then used for the semantics of specific data types and state types.

The sets *Value* and *ObjValue* are the least solutions to the equations below. We assume the unions are disjoint. The symbol $\lhd$ means domain subtraction.

$$\begin{aligned} \mathit{Value} &= \{\mathbf{error}, \mathbf{null}\} \cup \{\mathrm{true}, \mathrm{false}\} \cup \mathbb{Z} \cup \mathit{ObjValue} \\ \mathit{ObjValue} &= \{f : (\{\mathit{myclass}\} \cup \mathit{LName}) \pfun (\mathit{CName} \cup \mathit{Value}) \mid \\ &\qquad \mathit{myclass} \in \mathrm{dom}\, f \land f\ \mathit{myclass} \in \mathit{CName} \land \\ &\qquad (\{\mathit{myclass}\} \lhd f) \subseteq (\mathit{LName} \pfun \mathit{Value})\} \end{aligned}$$

Values for other primitive types should also be included. An object value is a mapping from field names to values, with the distinguished name *myclass* mapped to a class name.

The meanings of data types are parameterized by a typing environment. For primitives, we define $[\![\mathbf{bool}]\!]_\Gamma = \{\mathbf{error}, \mathrm{true}, \mathrm{false}\}$ and $[\![\mathbf{int}]\!]_\Gamma = \{\mathbf{error}\} \cup \mathbb{Z}$. For N in $\Gamma.cnames$, we define $[\![N]\!]_\Gamma$ to be the correctly-typed object values.

$$\begin{aligned} [\![N]\!]_\Gamma = \{&\mathbf{error}, \mathbf{null}\} \cup \\ &\{f : \mathit{ObjValue} \mid \\ &\quad \mathrm{dom}\ f = \mathrm{dom}(\Gamma.attr\ (f\ \mathit{myclass})) \cup \{\mathit{myclass}\} \land \\ &\quad f\ \mathit{myclass} \leq_\Gamma N \land \\ &\quad \forall x : \mathrm{dom}(\Gamma.attr\ (f\ \mathit{myclass})) \bullet f\ x \in [\![\Gamma.attr\ (f\ \mathit{myclass})\ x]\!]_\Gamma\} \end{aligned}$$

It is straightforward to prove that $N \leq_\Gamma N'$ implies $[\![N]\!]_\Gamma \subseteq [\![N']\!]_\Gamma$.

States are elements of *ObjValue*, although the "attributes" in a state include values of parameters and local variables. We write $[\![\Gamma, N]\!]$ for the set of states for class N and typing environment Γ. An state σ is in $[\![\Gamma, N]\!]$ just if it satisfies the following conditions. First, σ gives values to the attributes of the actual class, if it is not **main**, and to the variables in $\Gamma.locals$.

$$N \neq \mathbf{main} \Rightarrow \mathrm{dom}\,\sigma \setminus \{\mathit{myclass}\} = \mathrm{dom}(\Gamma.attr\ (\sigma\ \mathit{myclass})) \cup \mathrm{dom}(\Gamma.locals)$$

The union is not disjoint: $\Gamma.locals$ declares the visible attributes and any local variables and method parameters; $\Gamma.attr(\sigma\ \mathit{myclass}))$ declares all attributes,

including inherited private ones. If N is **main**, σ gives values just to the variables in $\Gamma.locals$. Also, if N is not **main**, then *myclass* is a subclass of N; otherwise, *myclass* is **main** itself. The last condition is that σ assigns values of the correct type. For $N \neq \mathbf{main}$ and x in $\mathrm{dom}\,\sigma \setminus \{myclass\}$ we require $x \in \mathrm{dom}(\Gamma.attr\ N)$ to imply $\sigma\ x \in [\![\Gamma.attr\ N\ x]\!]_\Gamma$, and $x \in \mathrm{dom}\,\Gamma.locals$ to imply $\sigma\ x \in [\![\Gamma.locals\ x]\!]_\Gamma$. Just the latter implication applies if $N = \mathbf{main}$.

4.2 Expressions and Predicates

For $\eta \in [\![\Gamma]\!], \sigma \in [\![\Gamma, N]\!]$, and derivable $\Gamma, N \rhd e : T$, we define $[\![\Gamma, N \rhd e : T]\!]\eta\ \sigma$, the value of e in state σ. It is an element of $[\![T]\!]_\Gamma$ (Lemma 5).

We assume that for built-in function $f : T \to U$ a semantics is given, as a total function $[\![T]\!]_\Gamma \to [\![U]\!]_\Gamma$. The semantics of **self** is as follows.

$$[\![\Gamma, N \rhd \mathbf{self} : N]\!]\eta\ \sigma = (\{myclass\} \cup \mathrm{dom}(\Gamma.attr\ (\sigma\ myclass))) \lhd \sigma$$

This uses domain restriction ($\lhd$) of σ: the attributes and *myclass* are retained; local variables and parameters are dropped. The similar definition for **super** and those for **null** and variables are omitted. We define $[\![\Gamma, N \rhd \mathbf{new}\ N' : N']\!]\eta\ \sigma$ as $init\ \Gamma\ N'$ where $init\ \Gamma\ N'$ is an object initialized with default values: false for boolean attributes, 0 for integers and **null** for objects. For other primitive types a default initial value should be given.

The value of the boolean expression e **is** N'' is determined by whether the value of e is an object of class N''. We omit the **null** and **error** cases.

$$\frac{[\![\Gamma, N \rhd e : N']\!]\eta\ \sigma = v \qquad v \notin \{\mathbf{null}, \mathbf{error}\}}{[\![\Gamma, N \rhd e\ \mathbf{is}\ N'' : \mathbf{bool}]\!]\eta\ \sigma = (v\ myclass \leq_\Gamma N'')}$$

Semantics of attribute selection, update, and cast are straightforward; they yield **error** for **null**.

The semantics $[\![\Gamma, N \rhd \psi : \mathbf{pred}]\!]\eta$ of a predicate ψ is a subset of $[\![\Gamma, N]\!]$ (Lemma 6). The semantics of expressions as formulas, and of the logical operations, is standard and omitted. The semantics of **isExactly** is similar to that of **is**.

$$\frac{[\![\Gamma, N \rhd e : N']\!]\eta = f}{\begin{array}{l}[\![\Gamma, N \rhd e\ \mathbf{isExactly}\ N'' : \mathbf{pred}]\!]\eta = \\ \quad \{\sigma : [\![\Gamma, N]\!] \mid f\ \sigma \notin \{\mathbf{null}, \mathbf{error}\} \wedge (f\ \sigma)\ myclass = N''\}\end{array}}$$

The coercion rules have similar semantics; we consider that involving **is** and $\wedge$.

$$\frac{[\![(\Gamma;\ x : N''), N \rhd \psi : \mathbf{pred}]\!]\eta = \Sigma \qquad N'' \leq_\Gamma N'}{\begin{array}{l}[\![(\Gamma;\ x : N'), N \rhd x\ \mathbf{is}\ N'' \wedge \psi : \mathbf{pred}]\!]\eta = \\ \{\sigma : [\![(\Gamma;\ x : N'), N]\!] \mid (\sigma\ x) \notin \{\mathbf{null}, \mathbf{error}\} \wedge (\sigma\ x)\ myclass \leq_\Gamma N'' \wedge \sigma \in \Sigma\}\end{array}}$$

This combines the interpretations of the combined operators.

$(\mathbf{val}\ x : T \bullet c)(e)$	$\longrightarrow (\mathbf{var}\ l : T \bullet l := e;\ c[l/x])$	if $l \notin (FV\ e) \cup (FV\ c)$
$(\mathbf{res}\ x : T \bullet c)(le)$	$\longrightarrow (\mathbf{var}\ l : T \bullet c[l/x];\ le := l)$	if $l \notin (FV\ le) \cup (FV\ c)$
$(\mathbf{vres}\ x : T \bullet c)(le)$	$\longrightarrow (\mathbf{var}\ l : T \bullet l := le;\ c[l/x];\ le := l)$	if $l \notin (FV\ le) \cup (FV\ c)$
$(pd;\ pds \bullet c)(e, e')$	$\longrightarrow (pd \bullet (pds \bullet c)(e'))(e)$	if $\alpha(pd) \notin (FV\ e')$
$(\bullet\ c)()$	$\longrightarrow c$	
$le.x := e$	$\longrightarrow le := (le;\ x : e)$	
$le.x, y := e, e'$	$\longrightarrow le, y := (le;\ x : e), e'$	
$le, le' := e, e'$	$\longrightarrow le', le := e', e$	
$m(e)$	$\longrightarrow \mathbf{self}.m(e)$	

Table 4. Syntactic transformations

4.3 Commands and Parameterized Commands

For command typing $\Gamma, N \rhd c : \mathbf{com}$ and environment $\eta \in [\![\Gamma]\!]$, the semantics $[\![\Gamma, N \rhd c : \mathbf{com}]\!]\eta$ is a total function on formulas (Theorem 1) which, when applied to a formula typable in Γ, N yields a result typable in Γ, N (Theorem 2).

Assignments to general left-expressions are dealt with using syntactic transformations that yield assignments of update expressions to simple variables and to **self**. Assignment to simple variables is interpreted using substitution.

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

We use an expression "**error**". In this paper we omit **error** from the grammar because it has no other use; its typing rule and semantics are straightforward.

User programs should not include assignments to **self** and method calls where **self** is used as a result or value-result argument. Assignments to **self** are introduced only in the syntactic transformations for parameter passing, when the argument corresponding to the *me* parameter of a method is **self**. This guarantees that **self** is always assigned an object of the current class, but the semantics cannot depend on this assumption.

$$\frac{\Gamma, N \rhd e : N' \quad N' \leq_\Gamma N}{\begin{array}{l}[\![\Gamma, N \rhd \mathbf{self} := e : \mathbf{com}]\!]\eta\ \psi = \\ \quad (\vee_{N' \leq_\Gamma N} \bullet\ e\ \mathbf{isExactly}\ N' \wedge \psi[e, e.x/\mathbf{self}, x]) \text{ where } x = \mathrm{dom}(\Gamma.attr\ N')\end{array}}$$

This uses a disjunction over the subclasses N' of N; each disjunct involves a substitution for appropriate attributes. There is no need to check that e is not **error** because **error isExactly** N' is false, for all N'. If the only assignments to **self** are those introduced in the semantics, **self** is always assigned an object of the current class, in which case the semantics simplifies to $\psi[e, e.x/\mathbf{self}, x]$. We need not give an operational justification for the general case.

We define $[\![\Gamma;\ x : T \rhd x : [\psi_1, \psi_2] : \mathbf{com}]\!]\eta\ \psi$ to be $\psi_1 \wedge (\forall\, x : T \bullet \psi_2 \Rightarrow \psi)$ as in Morgan's work. We also use the standard semantics for control constructs and blocks.

Parameter passing and various forms of assignment are reduced by the rule below to more basic constructs using the relation $\longrightarrow$ defined in Table 4.

$$\frac{c \longrightarrow^* c' \quad [\![\Gamma, N \rhd c' : \mathbf{com}]\!]\eta = g}{[\![\Gamma, N \rhd c : \mathbf{com}]\!]\eta = g}$$

If $\Gamma, N \rhd c : \mathbf{com}$ and $c \longrightarrow c'$ then $\Gamma, N \rhd c' : \mathbf{com}$ (Lemma 3). The reflexive-transitive closure $\longrightarrow^*$ of $\longrightarrow$ reduces every derivable command typing to one for which there is a direct semantic definition (Theorem 1). The first five transformations rewrite parameter passing in the usual way; $\alpha(pd)$ denotes the set of variables declared in pd, and FV gives free variables. The next three transformations rewrite assignments to left-expressions into assignments to simple variables or **self**. The last transformation inserts the missing object (**self**) in a method call $m(e)$.

4.4 Programs and Method Calls

The meaning of a complete program is the meaning of its main command, in an appropriate environment. The typing includes global variables x of c.

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

The environment η records the methods available for objects of each of the classes declared in *cds*; these methods are extracted from *cds* by the function *Meths* which builds η as follows.

For each class N and method m, the parameterized command $\eta\ N\ m$ has an extra value-result parameter me, and in its body each occurrence of an attribute x of N or of a call to a method m of N is replaced by $me.x$ and $me.m$. Only "top level" occurrences of attributes are changed: if x is an attribute, then $x.x$ becomes $me.x.x$. For a class that inherits m, me must be given the more specific type; it always has exactly the type of the object, compatible with the typing rule for value-result parameters.

If the declared body of a method m contains recursive invocations, then $\eta\ N\ m$ is the least fixed point of the context determined by the body. This approach is also used in Back's work and [CSW98] to deal with recursive parameterized procedures. We forbid mutual recursion so that fixpoints can be taken separately for each method. We justify existence of the least fixed point by techniques used in the cited works; it depends on monotonicity (Theorem 3).

Finally we consider method calls $le.m(e)$. Even though $le.m$ is a parameterized command, typed for example as $\Gamma, N \rhd le.m : \mathbf{pcom}(\mathbf{val}\ x : T)$, no transformation rule is applicable. In a state where the dynamic type of le is N', $\eta\ N'\ m$ takes the form $(\mathbf{vres}\ me : N';\ \mathbf{val}\ x : T \bullet c)$, and if we define $f_{N'}$ as $[\![\Gamma, N \rhd (\mathbf{vres}\ me : N';\ \mathbf{val}\ x : T \bullet c)(le, e) : \mathbf{com}]\!]\eta$, then we should define $[\![\Gamma, N \rhd le.m(e) : \mathbf{com}]\!]\eta\ \psi$ to be $f_{N'}\ \psi$. The semantics of method call 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 the static type N'' of le, determined by the derivation of le.

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

The hypothesis depends on $\eta\ N'\ m$ being typable in Γ, N. The free variables in the original declaration of m are attributes visible in the class, now accessed through the *me* parameter. Those attributes are not necessarily visible in the context of the call, so references $me.x$ are only typable in the extended system.

4.5 Example

The program below acts on a global variable c of type C. For clarity, we write the body of a method with no parameters as a command, instead of as a parameterized command with an empty declaration.

class C **pri** x : **int**; **meth** $Inc \mathrel{\widehat{=}} x := x + 1$; **meth** $Dec \mathrel{\widehat{=}} x := x - 1$ **end**
• $c.Inc()$

We calculate the weakest precondition for this program to establish $c.x > 0$. Writing CD to stand for the declaration of C above, we begin.

$$[\![\varnothing;\ c : C;\ \rhd CD \bullet c.Inc() : \mathbf{program}]\!]\ (c.x > 0)$$

$$= [\![\Gamma, \mathbf{main} \rhd c.Inc() : \mathbf{com}]\!]\eta\ (c.x > 0)$$

Here the typing environment $\Gamma = (VDecs\ CD\ \mathbf{main});\ c : C$ is as follows.

$$\begin{array}{l}(attr = \{\mathbf{object} \mapsto \varnothing, C \mapsto \{x \mapsto \mathbf{int}\}\ \},\\ \ meth = \{\mathbf{object} \mapsto \varnothing, C \mapsto \{Inc \mapsto \varnothing, Dec \mapsto \varnothing\}\ \},\\ \ vis = \{\mathbf{object} \mapsto \varnothing, C \mapsto \{x \mapsto pri\}\ \},\\ \ cnames = \{\mathbf{object}, C\},\ supcls = \{C \mapsto \mathbf{object}\},\ locals = \{c \mapsto C\})\end{array}$$

The environment $\eta = Meth\ \Gamma\ CD$ is shown below.

$$\{\mathbf{object} \mapsto \varnothing, C \mapsto \{Inc \mapsto (\mathbf{vres}\ me : C \bullet me.x := me.x + 1),\ Dec \mapsto \ldots\}\ \}$$

We proceed as follows.

$$[\![\Gamma, \mathbf{main} \rhd c.Inc() : \mathbf{com}]\!]\eta\ (c.x > 0)$$

$$= (\vee_{N' \leq_\Gamma C} \bullet\ c\ \mathbf{isExactly}\ N' \wedge [\![\Gamma, \mathbf{main} \rhd (\eta\ N'\ Inc)(c) : \mathbf{com}]\!]\ (c.x > 0))$$

[by the semantics of method call]

$$= c \notin \{\mathbf{null}, \mathbf{error}\} \wedge [\![\Gamma, \mathbf{main} \rhd (\eta\ C\ Inc)(c) : \mathbf{com}]\!]\ (c.x > 0)$$

[by C has no proper subclasses and the semantics of **isExactly**]

$$= c \notin \{\mathbf{null}, \mathbf{error}\} \wedge$$

[by the definition of η]

$$[\![\Gamma, \mathbf{main} \rhd (\mathbf{vres}\ me : C \bullet me.x := me.x + 1)(c) : \mathbf{com}]\!]\eta\ (c.x > 0)$$

$$= c \notin \{\mathbf{null}, \mathbf{error}\} \wedge \qquad \text{[by a syntactic transformation]}$$
$$[\![\Gamma, \mathbf{main} \rhd (\mathbf{var}\ l : C \bullet l := c;\ l.x := l.x + 1;\ c := l) : \mathbf{com}]\!]\eta\ (c.x > 0)$$
$$= c \notin \{\mathbf{null}, \mathbf{error}\} \wedge \qquad \text{[by the semantics of variable blocks]}$$
$$\forall\, l \bullet [\![\Gamma;\ l : C, \mathbf{main} \rhd (l := c;\ l.x := l.x + 1;\ c := l) : \mathbf{com}]\!]\eta\ (c.x > 0)$$
$$= c\ \notin \{\mathbf{null}, \mathbf{error}\} \wedge c.x \neq \mathbf{error} \wedge (c.x > 0)[l/c][(l;\ x : l.x + 1)/l][c/l]$$
[by the semantics of sequence and assignment]
$$= c \notin \{\mathbf{null}, \mathbf{error}\} \wedge c.x \neq \mathbf{error} \wedge c.x + 1 > 0$$
[by a properties of substitution and update expressions]

The result obtained is exactly what should be expected.

5 Properties of the Semantics

This section shows that the semantics is a well-defined function of typings, and that it is type-correct. Before presenting these theorems, however, we present auxiliary results.

Lemma 3. *The syntactic transformations preserve typing, in the sense that* $\Gamma, N \rhd c : \mathbf{com}$ *and* $c \longrightarrow c'$ *imply* $\Gamma, N \rhd c' : \mathbf{com}$*, for all* c, c'*.*

To prove the type-correctness theorem, we need typability to be preserved by substitution on formulas. This result holds only in the extended type system, where subtyping constraints are dropped from the rules for type tests and casts.

Lemma 4. *(a) Suppose* $\Gamma, N \rhd \psi : \mathbf{pred}$ *is derivable and* x *is free in* ψ*; let* T *be the type of* x *(which is uniquely determined by* Γ, N*). If* $T' \leq_\Gamma T$ *and* $\Gamma, N \rhd e : T'$ *is derivable then* $\Gamma, N \rhd \psi[e/x] : \mathbf{pred}$ *is derivable. (b) Same as part (a) but with* **self** *in place of* x*.*

The rules for assignment and result-parameter passing also involve subtyping constraints, but that does not invalidate Lemma 4 because predicate typings do not depend on command typings.

Because the semantics of ROOL is not defined by structural recursion on program texts, we need to show that the notation is coherent, in the sense that $[\![\Gamma, N \rhd \mathit{phrase} : \theta]\!]$ is a function of the typing $\Gamma, N \rhd \mathit{phrase} : \theta$. Expression typings have unique derivations (Lemma 1), and the semantics is defined directly in terms of the typing rules, so coherence for expressions is immediate. As a result, type-correctness for expressions is straightforward.

Lemma 5. *If* $\Gamma, N \rhd e : T$ *then* $[\![\Gamma, N \rhd e : T]\!]\eta\ \sigma \in [\![T]\!]_\Gamma$ *for all* $\eta \in [\![\Gamma]\!]$ *and* $\sigma \in [\![\Gamma, N]\!]$*.*

Due to the coercion rules, predicate typings are not unique. We need a coherence lemma.

Lemma 6. *The semantics* $[\![\Gamma, N \rhd \psi : \mathbf{pred}]\!]$ *of a predicate typing is a function of the typing* $\Gamma, N \rhd \psi : \mathbf{pred}$*, and* $[\![\Gamma, N \rhd \psi : \mathbf{pred}]\!] \subseteq [\![\Gamma, N]\!]$.

For command typings, derivations are unique except for derivations of predicates that occur within commands. Nevertheless, the semantics of commands does not depend on semantics of predicates, so there is no issue of coherence.

There are two parts of the semantics of commands, however, that are not simply defined by structural recursion on derivations. The first is that for some commands the semantics is given indirectly by syntactic transformation. Nonetheless, these transformations preserve typing (Lemma 3), and the derivations of the transformed phrases are built from the derivations of the original phrases in such a way that the semantics depends only on the semantics of subderivations.

Method calls are the second difficult part: $[\![\Gamma, N \rhd le.m(e) : \mathbf{com}]\!]\eta$ depends on the semantics of method calls $[\![\Gamma, N \rhd \eta\ N'\ m(e) : \mathbf{com}]\!]\eta$ where N' ranges over subtypes of the type N'' of le. The parameterized command $\eta\ N'\ m$ can contain method calls, so the semantics of a method call depends on the semantics of method calls, which are certainly not part of the derivation of $le.m(e)$.

However, we are only concerned recursion-free environments: those obtained from *Meth* Γ *cds*, in which recursion has been resolved already. The semantics of a method m of a class N depends only on methods N', m' that do not depend on N, m, and the relation "can call" on pairs N', m' is well founded. We combine this lexicographically with the order "is a subderivation" to obtain a well founded order. We define the notion of the semantics $[\![\Gamma, N \rhd \mathit{phrase} : \theta]\!]$ in the context of some method N', m'; this depends on subderivations of $\mathit{phrase} : \theta$ and also on semantics for phrases in context of methods N'', m'' smaller than N', m'.

Theorem 1. *For all derivable* $\Gamma, N \rhd c : \mathbf{com}$ *and all* $\eta \in [\![\Gamma]\!]$*, the semantics* $[\![\Gamma, N \rhd c : \mathbf{com}]\!]\eta$ *is a total function on all formulas, regardless of type, provided that* η *is recursion-free.*

Proof By induction with respect to the order discussed above.
Case assignment: for assignments to simple identifiers, and for assignments to **self**, the semantics is given directly. Others are reduced by syntactic transformations to the simple case. By Lemma 3 the transformed assignments are typable in Γ, N. Any assignment can be rewritten to a simple one which is unique up to the order in which variables are listed; and order does not affect the semantics.
Case specification statement: this has a single typing rule and the semantics is given directly.
Case application $pc(e)$ of an explicit parameterized command (not a method call): the transformation rules eliminate argument(s) e in favor of local variables and assignments. The result is typable (Lemma 3). Moreover, the derivation of the transformed command is composed of subderivations of the original command. Introducing local variables involves the choice of identifier l, but the semantics is independent of the choice because l is bound by $\forall$.
Case method call applied to parameters: a method call $m(e)$ is reduced to $\mathbf{self}.m(e)$, which has the general form $le.m(e)$. Let ψ be any formula. The semantics for $le.m(e)$ is defined provided each $f_{N'}$, i.e. $[\![\Gamma, N \rhd \eta\ N'\ m(le, e) : \mathbf{com}]\!]\eta$,

is defined. By the conditions on environments, $\eta\ N'\ m(le, e)$ is typable. The methods on which $\eta\ N'\ m$ depends are smaller in our ordering, by the proviso that η is recursion-free. By induction, $[\![\Gamma, N \rhd \eta\ N'\ m(le, e) : \mathbf{com}]\!]\eta$ denotes a total function on formulas, and hence so does the semantics of the call.
Cases explicit recursion: this is defined using least fixpoints of program contexts. Because these are monotonic (Theorem 3), the least fixpoints are well defined.
Cases sequence, alternation and variable blocks: in each case there is a direct semantic definition and the result holds by induction. □

Theorem 2. *If* $\Gamma, N \rhd \psi : \mathbf{pred}$ *and* $\Gamma, N \rhd c : \mathbf{com}$ *are derivable then so is* $\Gamma, N \rhd ([\![\Gamma, N \rhd c : \mathbf{com}]\!]\eta\ \psi) : \mathbf{pred}$*, provided* η *is recursion-free.*

Proof By induction, using the order defined above.
Case assignment: for simple variables, the semantics requires that the predicate $e \neq \mathbf{error} \wedge \psi[e/x]$ be typable in Γ provided that ψ is. Thus we need that $\Gamma \rhd x : T$ and $\Gamma \rhd e : T'$ and $T' \leq_\Gamma T$ imply $\Gamma \rhd \psi[e/x] : \mathbf{pred}$. That is by Lemma 4(a). To type $e \neq \mathbf{error}$, we use the typing rule for **error** (which gives it any type), and then the rule for built-in functions to type the equality. For assignments to **self**, suppose ψ is typable in Γ, N. For each N', we have, by Lemma 4(b), $\psi[e, e.x/\mathbf{self}, x]$ typable in Γ, N'. Moreover, if an assignment to **self** is typable in Γ, N, then **self** is typable in Γ, N and so $N \neq \mathbf{main}$. Thus, by Lemma 2, $(\vee_{N' \leq_\Gamma N} \bullet\ \mathbf{self}\ \mathbf{isExactly}\ N' \wedge \psi[e, e.x/\mathbf{self}, x])$ is typable in Γ, N.
Case specification statement: for $\Gamma;\ x : T \rhd x : [\psi_1, \psi_2] : \mathbf{com}$ to be derivable, ψ_1 and ψ_2 are typable in $\Gamma;\ x : T$. For ψ with $\Gamma;\ x : T \rhd \psi : \mathbf{pred}$ the semantics yields $\psi_1 \wedge (\forall x : T \bullet \psi_2 \Rightarrow \psi)$, which can be typed for $\Gamma;\ x : T$ using the rules for $\wedge$, $\forall$, and $\Rightarrow$.
Cases sequence: straightforward use of induction.
Case alternation: by induction, each f_i in the semantics yields well-typed formulas, and the guards have to be typable predicates in Γ, N, so the formula $(\vee\ i \bullet \psi_i) \wedge (\wedge\ i \bullet \psi_i \Rightarrow f_i\ \psi)$ is also typable using the rules for $\wedge$, $\vee$, and $\Rightarrow$.
Case method call: for method calls $le.m(e)$, we have to show that the predicate $(\vee_{N'} \bullet\ le\ \mathbf{isExactly}\ N' \wedge f_{N'}\ \psi)$ is typable in Γ, N. By induction, each $f_{N'}$ applies to formulas typable in Γ, N, and each returns the same. Now le is typable in Γ, N, so by using the rules $\vee$, $\wedge$, and **isExactly** we obtain the desired result.
Case blocks: the weakest precondition $[\![\Gamma \rhd (\mathbf{var}\ x : T \bullet c\ \mathbf{end}) : \mathbf{com}]\!]\eta\ \psi$ is defined as $(\forall x : T \bullet f\ \psi)$, where $f = [\![\Gamma;\ x : T \rhd c : \mathbf{com}]\!]\eta$. If ψ is typable in Γ then it is also typable in $\Gamma;\ x : T$. Therefore f can be applied to ψ and by induction $f\ \psi$ is typable in $\Gamma;\ x : T$, and hence by the typing rule for $\forall$ we get $(\forall x : T \bullet f\ \psi)$ typable in Γ. Similar considerations apply to **avar** blocks. □
It is straightforward to formulate and prove definedness and type-preservation for complete programs, using Theorems 1 and 2.

6 Refinement

In this section we define notions of refinement and give the basic result on monotonicity. To simplify definitions, we assume that all phrases are well-typed.

The fundamental refinement relationship $\sqsubseteq$ is between programs. This is based on pointwise order on predicate transformers, as usual, but restricted to healthy predicates just as in languages where procedures can be assigned to variables [Nau98b, HH98]. As an example, if class *CPt* suitably refines *Pt* we expect the refinement $x := \mathbf{new}\ Pt \sqsubseteq x := \mathbf{new}\ CPt$. But the postcondition x **isExactly** *Pt* is established only by the first assignment. The solution is to restrict attention to monotonic predicates. For our purposes, a predicate ψ is monotonic provided that for any object values $ov1, ov2$, if $ov1$ satisfies ψ and $ov2$ *myclass* $\leq_\Gamma$ $ov1$ *myclass*, and $ov2$ agrees with $ov1$ on all the attributes of $ov1$ *myclass*, then $ov2$ satisfies ψ.

Definition 1. *For sequences of class declarations cds and cds′, commands c and c′ with the same free variables x : T, define* $(cds \bullet c) \sqsubseteq (cds' \bullet c')$ *if and only if, for all monotonic* ψ*,*

$$[\![\varnothing;\ x : T \rhd (cds \bullet c) : \mathbf{program}]\!]\ \psi \Rightarrow [\![\varnothing;\ x : T \rhd (cds' \bullet c') : \mathbf{program}]\!]\ \psi$$

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

A program can be refined by refining its command part and its class declarations. Commands in ROOL apear in the context of a sequence of class declarations, so we first define relation $cds, N \rhd c \sqsubseteq c'$, which establishes that in the context of *cds* the command c occurring in the class N is refined by c'.

Definition 2. *For a sequence of class declarations cds, commands c and c′, and a class N, define* $cds, N \rhd c \sqsubseteq c'$ *if and only if, for all monotonic predicates* ψ*,*

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

where $\Gamma = (VDecs\ cds\ N)$*;* $x : T$*, x are the method parameters and local variables in scope, and* $\eta = Meths\ \Gamma\ cds$*.*

Because methods are parameterized commands, we need the analog of Definition 1 for them.

Definition 3. *For sequence of class declarations cds, parameterized commands pc and pc′, which have the same parameters, and a class N,* $cds, N \rhd pc \sqsubseteq pc'$ *if and only if, for all (lists of) expressions e,* $cds, N \rhd pc(e) \sqsubseteq pc'(e)$

This is a straightforward extension Back's definition (see [CSW98]).

Using induction as in Theorems 1 and 2, the following can be proved.

Theorem 3. *Suppose we have a sequence of class declarations cds, a class N, a parameterized command pc, and a context* $\mathcal{C}[\cdot]$ *which is a parameterized command, and so, a function from parameterized commands to parameterized commands. If we have that* $cds, N \rhd pc \sqsubseteq pc'$*, then* $cds, N \rhd \mathcal{C}[pc] \sqsubseteq \mathcal{C}[pc']$*. Similarly, the command constructors are monotonic.*

This theorem justifies our treatment of recursion and recursive methods.

As a class is a data type, refinement of classes is related to data refinement [HHS87]. We define the relation $view, cds \rhd cds' \preccurlyeq cds''$, for a list of methods *view* and sequences of class declarations *cds*, *cds′*, and *cds″*. The meaning is that in the context of *cds*, if only methods listed in *view* are used, then the class declaration *cds′* can be replaced by *cds″*.

Definition 4. *For a list of methods view, sequences of class declarations cds, cds′, and cds″, view*, $cds \rhd cds' \preccurlyeq cds''$ *if and only if, for all commands c that uses only methods in view,* $(cds;\ cds' \bullet c) \sqsubseteq (cds;\ cds'' \bullet c)$.

Refinement between single classes *cd′* and *cd″* is a special case. By considering a more general relation, we allow for restructuring a collection of class declarations. In practice, Definition 4 would not be used directly, but it is the fundamental notion with respect to which techniques such as downward and upward simulation must be proved sound [HHS87, Nau98a].

7 Discussion

We have shown how the standard predicate-transformer model can be extended to an object-oriented language. The semantics can be modified to allow arbitrary mutual recursion among methods, at the cost of taking a single fixpoint for the entire environment of methods. This makes it more complicated to prove refinement laws, so we have chosen the simpler approach at this stage.

Others [Lei98, MS97, BKS98] have extended existing refinement calculi with object-oriented features, but restricting inheritance or not dealing with classes and visibility. Those works, however, deal with sharing and concurrency. Another approach to objects is implicit in the parametricity semantics of Algol-like languages. It has been adapted to object-oriented programs by Reddy [Red98], with whom we are collaborating to give a semantics for ROOL.

The main shortcoming of our semantics is that it is not entirely compositional. Since our aim is to validate laws like those in [Bor98], for when one class is a behavioural subclass of another, within the context of some other classes, this is a potential problem. However, the touchstone criteria for behavioural refinement is that $cds_1 \bullet c \sqsubseteq cds_2 \bullet c$ should hold whenever cds_2 is obtained from cds_1 by behavioural refinement of some classes. Fortunately, this has a natural formulation with a single context that includes all relevant classes.

Our notion of class refinement corresponds to the notion of behavioural subtyping introduced by Liskov and Wing [LW94]. Definition 4 captures the essence of their subtype requirement. In our framework the property of interest is refinement of programs, which captures the notion of total correctness. The two ways of defining the subtype relation presented in [LW94] are based on the downward simulation technique [HHS87], specialized to the particular case of functional data refinement. We expect that particular techniques like these can be proved sound with respect to Definition 4. Similarly, Liskov and Wing claim, but do not formalize, that their definitions satisfy the subtype requirement.

By using a language of specification and programming, we do not need a distinction between specifications and implementations of classes. As already seen in traditional refinement calculi, this simplifies both the theory of refinement and the presentation and application of refinement laws.

Acknowledgement This work benefitted from discussions with our collaborators Augusto Sampaio, Uday Reddy, Paulo Borba, and Hongseok Yang. UFPE and Stevens provided generous support for travel.

References

[AdB94] Pierre America and Frank de Boer. Reasoning about dynamically evolving process structures. *Formal Aspects of Computing*, 6:269–316, 1994.

[AL97] Martín Abadi and K. Rustan Leino. A logic of object-oriented programs. In *Proceedings, TAPSOFT 1997*. Springer-Verlag, 1997. Expanded in DEC SRC report 161.

[BvW98] R. J. R. Back and J. von Wright. *Refinement Calculus: A Systematic Introduction*. Graduate Texts in Computer Science. Springer-Verlag, 1998.

[BKS98] Marcello M. Bonsangue, Joost N. Kok, and Kaisa Sere. An approach to object-orientation in action systems. In Johan Jeuring, ed., *Mathematics of Program Construction*, LNCS 1422, pages 68–95. Springer, 1998.

[Bor98] Paulo Borba. Where are the laws of object-oriented programming? In *I Brazilian Workshop on Formal Methods*, pages 59–70, Porto Alegre, Brazil, 19th–21st October 1998.

[CN99] A. L. C. Cavalcanti and D. A. Naumann. A Weakest Precondition Semantics for an Object-oriented Language of Refinement - Extended Version. Available at `http://www.di.ufpe.br/~alcc`

[CSW98] A. L. C. Cavalcanti, A. C. A. Sampaio, and J. C. P. Woodcock. Procedures and Recursion in the Refinement Calculus. *Journal of the Brazilian Computer Society*, 5(1):1–15, 1998.

[CSW99] A. L. C. Cavalcanti, A. Sampaio, and J. C. P. Woodcock. An inconsistency in procedures, parameters, and substitution in the refinement calculus. *Science of Computer Programming*, 33(1):87–96, 1999.

[HH98] C. A. R. Hoare and J. He. *Unifying Theories of Programming*. Prentice Hall, 1998.

[HHS87] C. A. R. Hoare and J. He and J. W. Sanders. Prespecification in data refinement. *Information Processing Letters*, 25(2), 1987.

[Lan95] Kevin Lano. *Formal Object-Oriented Development*. Springer, 1995.

[Lei98] K. Rustan M. Leino. Recursive object types in a logic of object-oriented programming. In Chris Hankin, ed., *7th European Symposium on Programming*, LNCS 1381. Springer, 1998.

[LW94] Barbara H. Liskov and Jeannette M. Wing. A behavioral notion of subtyping. *ACM Transactions on Programming Languages and Systems*, 16(6), 1994.

[MS97] A. Mikhajlova and E. Sekerinski, Class refinement and interface refinement in object-oriented programs. In *Proceedings of FME'97: Industrial Benefit of Formal Methods*. Springer, 1997.

[Mor94] Carroll Morgan. *Programming from Specifications*, 2ed. Prentice Hall, 1994.

[Nau98a] David A. Naumann. Validity of data refinement for a higher order imperative language. Submitted.

[Nau98b] David A. Naumann. Predicate transformer semantics of a higher order imperative language with record subtypes. *Science of Computer Programming*, 1998. To appear.

[Red98] U. S. Reddy. Objects and classes in Algol-like languages. In *Fifth Intern. Workshop on Foundations of Object-oriented Languages*. URL: http://pauillac.inria.fr/ remy/fool/proceedings.html, Jan 1998.