A Denotational Approach for Type-Checking in Object-Oriented Programming Languages

Roberto Ierusalimschy
Departamento de Informática
Pontifícia Universidade Católica
22453-900 – Rio de Janeiro – Brazil
e-mail:roberto@inf.puc-rio.br

August, 1992

Abstract

Starting with a pragmatical (but formal) definition of type in object-oriented languages, this paper proposes a method to test type safety in this kind of language. We say that a language is (type) safe if it ensures that, during the execution of a correct program, every message sent to an object is matched by an appropriate method.

We define a "typical" object oriented programming language, featuring multiple inheritance, recursive types, and separation between specifications and implementations. Then, we give a formal definition for its type system, and a denotational semantics for the execution of the language, based on message passing. Finally, we formally prove that our language is type safe.

Along the work, better understanding is gained about many problems related with type systems in object-oriented languages.

Keywords: Semantics of Programming Languages, Type Systems, Object Oriented Programming.

1 Introduction

In recent years, there has been a great interest in type systems for Object-Oriented programming languages (OOPL). This field has received important results from groups working towards formal grounds, addressing issues like data abstraction [7], inheritance and subtyping [5], recursive types [1], etc. Also, there have been many contributions on a more practical side. Almost every language presents its own concept of type system, differing from others in points like single versus multiple inheritance (e.g. Beta [15] versus CommonLisp [4]), separate hierarchies for types and classes (e.g. Pool [2] and DuoTalk [16]), possibility of redefinition of signatures of inherited methods (e.g. C++ [10] does not allow, while Eiffel [17] does), and many other topics.

Usually, the formalization of OO type concepts is done by translating these concepts into a more "formal" language, like typed lambda calculus or an algebraic framework [9]. Some advantages of this approach are the use of a simple and well understood framework and the possibility of isolating the concepts of the study. Moreover, the concept of *type* in these frameworks is more exactly defined than in most programming languages.

However, some of those pros are also cons. Frequently, results developed in an applicative language are not easily translated back into an imperative language, a category that includes most of the widely-used OOPLs (like Smalltalk [13] and C++). Moreover, most OO characteristics are not orthogonal; for example, [7] studies abstract types with inheritance (without recursive types), while [1] discusses the interaction between recursive types with subtyping (but without data abstraction).

In this paper we propose an alternative approach for formalizing OO type systems, wherein we can study aspects of "real" OOPLs. This approach assumes a denotational description of the language, and proceeds by proving that the language is "well-typed". In order to do that, our fist step is an appropriate definition of type.

In object-oriented programming languages, objects carry their own operations, and these operations are the only visible part of an object; the only thing one can do with an object is to send a message to it. So, if we want to characterize types as, for example, "a collection of values sharing a common structure or shape"[1], we should use only the operations as the visible structure of an object. Therefore, a good definition for types in OO languages is that the type of an object is its interface, that is, its collection of operations.[2] Following the above definition of type, we can say that a type error in an OO program is the sending of a message to an object that has no method for it.[3] *Statically typed* languages are those that can ensure the absence of such errors by a type-checking procedure performed during compile-time.

The remainder of the paper is devoted to show how one can prove that an OOPL is statically typed, using our approach. Along the work we will adopt VDM [14] as our formal language. The next section presents a somehow typical language, called School. This language follows the imperative tradition, and includes features like multiple inheritance, freely recursive types, the pseudo-variable Self, and late-binding. Section 3 gives a denotational description of the type system of School. In section 4 we describe the run-time behavior of a School program, following an approach similar to [18]. Section 5 is the main part of the paper, and presents a formal proof that correctly typed programs run without "message not understood" errors. Some extensions for our language are presented in section 6. Finally, section 7 exposes some interesting conclusions we have arrived at during this work.

2 The School Programming Language

A program in School is composed by a sequence of modules. Each module defines a specification (type) or an implementation (class). A specification defines the interface of an object, that is, the operations it must export, and lists its supertypes. That list is used to build the specification hierarchy, which defines type compatibility along the language. Implementations, on the other hand, define the internal structure of objects, like their internal variables, and code for the various operations (methods).

A specification module has the following syntax[4]:

```
Type typeName
  {Subtype of superTypeName
    [redefine opName {, opName}] }
```

[1] [1], pp. 104

[2] Most OOPLs, like Simula [3], Beta, C++, and Eiffel, identify classes with types; few, like Smalltalk (without a concept of type) and Pool [2], do not. This identification implies that types are not really abstract, because their internal structure can be used to differentiate them. Therefore, this work considers types and classes as two different concepts. Nevertheless, we can easily model that identity assuring a one-to-one correspondence between types and classes.

[3] What is called a "message not understood" error in Smalltalk.

[4] As usual, "{ }" means 0 or more repetitions, and "[ ]" means an optional item.

```
{Function functionName ({parameter}) :  resultType ; }
End typeName
```

Each type can have multiple supertypes. Each "**Subtype of** " declaration includes the operation definitions from the supertype into the new specification. Every operation is included, unless its name is in the *redefinition* list; in this case the new specification must provide a (re)definition for the operation. Whenever an operation is redefined, the new definition must be "compatible" with the old one. To be compatible means that the result type of the new definition must be a subtype of the old result type, and the parameter types of the old definition must be subtypes of the respective parameter types of the new definition.[5]

After the hierarchy declarations, follow the operation definitions. These definitions include the new versions for redefined operations. In order to keep simplicity, we assume that an operation always has a result type, that can be VOID[6] when the operation has nothing else to return. A `parameter` in a function definition has the form `name : typeName`.

Any clash between operation names inherited from different supertypes, or between an inherited operation and a local one, is considered an error (that is, the meaning of the specification module is undefined). Notice that this remark does not apply for operations in the redefinition list, as these operations are not inherited. Specifically, if two or more supertypes have operations with equal names, but all these operations are redefined, there is no conflict. In that case, the new unique definition must be compatible with all old versions.

A note about recursive definitions: School, as most object oriented languages, supports recursive types. So, function definitions inside a specification can refer to names of types not yet defined. However, School, again as most OO-languages, does not allow cycles in the inheritance graph. To avoid that possibility, a type must have been already declared in order to be used as a supertype.

An implementation module is written as:

```
Implementation implementationName :  typeName
   [Subimplementation of {implementationName} ]
  {Var varName :  typeName ; }
  {Function opName ({parameter}) :  resultType is
    expression } ;
End implementationName
```

Every class implements an explicit type, which is named in the beginning of the module, after the implementation name. This type is also the type of the pseudo-variable **self** (see below). The "**Subimplementation of** " declaration states the superclasses of the implementation. This declaration makes available all methods and variables from the superclasses. All inherited variables and methods must have distinct names, but the methods can be overwritten by new ones. An implementation can also declare new instance variables for the objects of that class, and new operations. These operations have free access to the new instance variables and the inherited ones.

As usual in object oriented languages, inside every class there is an implicitly declared pseudo-variable[7] named **self** that always refers to the object executing that method. In order

[5] Concepts like compatibility are formally defined in section 3.

[6] The type Void is a proper supertype of all other types, and has no operations.

[7] **self** is not a real variable because one can not assign a value to it.

to properly type **self** the type of an implementation must be a subtype of the types of its superclasses.

School is an expression-oriented language. That means that different levels of command structure are merged into the single level of expressions. In particular, the body of a function is also an expression, and the value returned by a function is the value of its body.

An expression can be any of the following options:

- **self**
- **nil**
- `varName := expression`
- **while** `expression` **do** `expression'`
- **if** `expression` **then** `expression' [else expression'']`
- `varName`
- `expression.opName ({expression})`
- **new** `implementationName`
- `block`

Unlike more conventional programming languages, School does not assign a type for each expression; instead, it defines when an expression *satisfies* a type. The main reason for this shift of emphasis is the **if** expression. In a language with multiple inheritance, like School, not all pairs of types have a unique common supertype. Therefore, there is no single type that we can assign to an **if** expression without being much too restrictive. Another reason is the **nil** expression: although it satisfies any type, there is no type to denote it. From now on, whenever we say "the expression E has type t" we mean "the expression E satisfies any supertype of t".

The expression **nil** denotes a predefined value, and it satisfies all types. This value is the default initial value for all variables. An assignment has the usual meaning in object oriented languages, that is, the expression evaluates to an object, and a reference to this object is stored in the variable. The final value of an assignment is the result of the expression. The type of the expression must satisfy the type of the variable, and the assignment itself has the type of the variable.

To avoid the introduction of a type *boolean* inside the language, **while** and **if** test for a nil value. A while expression executes its second expression while the first one is different from Nil; it only satisfies the type VOID. An **if** expression evaluates its first expression; if the result is different from Nil, it evaluates `expression'`; otherwise, `expression''` is evaluated. Anyway, the final value of the expression is the value of the last evaluated expression. An **if** expression satisfies any type that is satisfied by both `expression'` and `expression''`.

Next we have function calls (or message passing). This expression calls the operation `opName` from the object that results from the first sub-expression, passing the values of the expression list as parameters. There must exist a type satisfying the first expression which has an operation named `opName`. That operation must have the same number of parameters as supplied, and each actual parameter must satisfy its respective formal parameter. The whole expression has the type of the result type in the operation definition.

In order to create new objects, we use the **new** expression. The result of this expression is a new object of the class `implementationName`, and it has the type associated with that implementation.

The last kind of expression is a block, that allows the declaration of local variables and sequential execution of expressions. A `block` has a declaration section followed by a list of expressions, and its value is the value of its last expression:

```
[Declare
{Var varName : typeName} ]
Begin
  {expression ;} expression
End
```

The attentive reader may have noticed that School has no I/O facilities. Any input must be hard-wired in the program code, while the output comes as a single value plus raw memory. For our purposes here that is enough; section 6 discusses how conventional I/O capabilities can be added to the language.

A Small Example

As we have already noted, School does not have a predefined type *boolean*. In fact, its only predefined type is VOID, and its only primitive value is NIL. Nevertheless, School is computationally complete. Here we show how the type *boolean* can be defined and implemented in the language.

The definition of that type can be as follows:

```
Type Boolean
  Function isTrue () : Void ;
  Function and (x : Boolean) : Boolean ;
  Function or (x : Boolean) : Boolean ;
  Function not () : Boolean ;
End Boolean
```

The function `isTrue` returns NIL or a value different from NIL, so that boolean values can be used as condition expressions in ifs and whiles. The functions `and`, `or` and `not` have the usual meaning.

To implement this type, we define two classes: one to represent the value *true* and other to represent the value *false*. Everything in a line after a "`--`" is a comment.

```
Implementation True : Boolean
  Function isTrue () : Void is self ; -- returns a value different from NIL
  Function and (x : Boolean) : Boolean is x ; -- true and x = x
  Function or (x : Boolean) : Boolean is self ; -- true or x = true
  Function not () : Boolean is new False ; -- not true = false
End True

Implementation False : Boolean
  Function isTrue () : Void is nil ;
  Function and (x : Boolean) : Boolean is self ; -- false and x = false
  Function or (x : Boolean) : Boolean is x ; -- false or x = x
  Function not () : Boolean is new True ; -- not false = true
End False
```

The next lines show how those definitions can be used in a program.

```
Declare Var a,b :  Boolean
Begin
  ...
  b := b.and(a.not()) ; -- b := b ∧ (¬ a)
  ...
  if (a.or(b)).isTrue then ...
  ...
End
```

Abstract Syntax

In order to simplify the formal definitions in the next sections, we will not use the syntax presented above for our language. Instead, we define here an abstract syntax, equivalent to the previous one, but without concrete details like reserved words and semi-colons. We assume the existence of a primitive type *Name*, that corresponds to identifiers.

$$\begin{aligned} \mathit{TypeDec} :: \quad \mathit{name} &: \mathit{Name} \\ \mathit{superTypes} &: \mathit{SuperType}^* \\ \mathit{functions} &: \mathit{FunctionHead}^* \end{aligned}$$

$$\begin{aligned} \mathit{SuperType} :: \mathit{name} &: \mathit{Name} \\ \mathit{redef} &: \mathit{Name}^* \end{aligned}$$

$$\begin{aligned} \mathit{FunctionHead} :: \quad \mathit{name} &: \mathit{Name} \\ \mathit{result} &: \mathit{Name} \\ \mathit{parameters} &: \mathit{VarDeclaration}^* \end{aligned}$$

$$\begin{aligned} \mathit{VarDeclaration} :: \mathit{name} &: \mathit{Name} \\ \mathit{type} &: \mathit{Name} \end{aligned}$$

$$\begin{aligned} \mathit{ImplementationDec} :: \quad \mathit{name} &: \mathit{Name} \\ \mathit{type} &: \mathit{Name} \\ \mathit{superClasses} &: \mathit{Name}^* \\ \mathit{vars} &: \mathit{VarDeclaration}^* \\ \mathit{functions} &: \mathit{Function}^* \end{aligned}$$

$$\begin{aligned} \mathit{Function} :: \mathit{header} &: \mathit{FunctionHead} \\ \mathit{body} &: \mathit{Expression} \end{aligned}$$

$$\begin{aligned} \mathit{Expression} = {} & \textsc{Self} \mid \mathsf{nil} \mid \mathit{Assignment} \mid \mathit{WhileExp} \mid \mathit{IfExp} \mid \mathit{VarExp} \mid \\ & \mathit{FunctionCall} \mid \mathit{NewExp} \mid \mathit{Block} \end{aligned}$$

$$\begin{aligned} \mathit{Assignment} :: \mathit{var} &: \mathit{Name} \\ \mathit{exp} &: \mathit{Expression} \end{aligned}$$

$$\begin{aligned} \mathit{WhileExp} :: \mathit{cond} &: \mathit{Expression} \\ \mathit{do} &: \mathit{Expression} \end{aligned}$$

$$\begin{aligned} \mathit{IfExp} :: \mathit{cond} &: \mathit{Expression} \\ \mathit{then} &: \mathit{Expression} \\ \mathit{else} &: \mathit{Expression} \end{aligned}$$

$$\mathit{VarExp} :: \mathit{var} : \mathit{Name}$$

$$\begin{aligned} \mathit{FunctionCall} :: \quad \mathit{receiver} &: \mathit{Expression} \\ \mathit{opName} &: \mathit{Name} \\ \mathit{parameters} &: \mathit{Expression}^* \end{aligned}$$

$$\mathit{NewExp} :: \mathit{impl} : \mathit{Name}$$

$$\begin{aligned} \mathit{Block} :: \quad \mathit{decl} &: \mathit{VarDeclaration}^* \\ \mathit{exps} &: \mathit{Expression}^* \end{aligned}$$

Finally, we can define a whole program as a sequence of specifications, then implementations, and an expression to start its execution.

$$\begin{aligned} \mathit{Program} :: \quad \mathit{specs} &: \mathit{TypeDec}^* \\ \mathit{impl} &: \mathit{ImplementationDec}^* \\ \mathit{exp} &: \mathit{Expression} \end{aligned}$$

3 The Type System of School

In the previous section we have described the language School. However, many type related concepts, like *compatibility* and *subtype*, were not precisely defined. Here we put the type system on a formal ground, using VDM. The main goal of this section is to provide two functions: the first one, applied to specifications, returns their denotation. The second one ranges over implementations, and returns a boolean indicating if that implementation is type correct or not.

School adopts a kind of *name compatibility* for types. That means that two types are considered equivalent if and only if they have the same name.[8] Moreover, a type is considered a subtype of another one only when there are declarations asserting that. Therefore, all types in a program can be denoted by their names, and we will use a *type environment* to associate those names with the actual type descriptions.

The first definitions are the domains to express names; all of them are equal to *Name*, introduced in the previous section. The use of different identifiers is only a matter of clarity.

$$\mathit{TypeName} = \mathit{OpName} = \mathit{ImpName} = \mathit{Name}$$

We assume the existence of one constant, VOID, belonging to this type. VOID is (the name of) the type without operations, and is a supertype of any other type.

A specification signature is a set of operations, each one with an arity. This can be modeled by the following types:

$$\begin{aligned} \mathit{Arity} :: \quad \mathit{result} &: \mathit{TypeName} \\ \mathit{parms} &: \mathit{TypeName}^* \end{aligned}$$

$$\mathit{Signature} = \mathit{OpName} \xrightarrow{m} \mathit{Arity}$$

A type consists of its signature plus a record of its supertypes:

$$\begin{aligned} \mathit{Type} :: \quad \mathit{sig} &: \mathit{Signature} \\ \mathit{superTypes} &: \mathit{TypeName}\text{-}\mathsf{set} \end{aligned}$$

To keep track of all types in a program, we declare a type environment, that maps all type names into their definitions:

$$\mathit{TypeEnv} = \mathit{TypeName} \xrightarrow{m} \mathit{Type}$$

[8] This option can be contrasted with *structural compatibility*, where two types are equivalent if they have the same structure. Section 6 shows how to adopt structural compatibility in School.

where

$$inv\text{-}TypeEnv(e) \quad \triangleq \quad e(\textsc{Void}) = mk\text{-}Type(\{\,\},\{\,\}) \wedge$$
$$\exists m \in TypeName \xrightarrow{m} \mathbf{N} \cdot$$
$$\mathsf{dom}\, m = \mathsf{dom}\, e \wedge \forall t \in \mathsf{dom}\, e \cdot \forall st \in superTypes(e(t)) \cdot$$
$$st \in \mathsf{dom}\, e \wedge m(st) < m(t)$$

The invariant of *TypeEnv* avoids the existence of loops in the hierarchy. In order to do that, it assigns a natural number for each type name, through the map m, and ascertains that supertypes have smaller numbers than its subtypes. It also ensures that all supertype names have a definition in the environment. The constant VOID needs special treatment, to assure that it is always available, without operations and without supertypes. The fact that all types have VOID as supertype must be dealt with as a special case (see function *subtype*).

Variable declarations can be easily modeled by the following map:

$$VarEnv = Name \xrightarrow{m} TypeName$$

A similar environment can be used to carry type information about implementations:

$$\begin{array}{lrcl} Implementation :: & type & : & TypeName \\ & superClasses & : & Name\text{-}\mathsf{set} \\ & vars & : & VarEnv \\ & sig & : & Signature \\ & bodies & : & Function^* \end{array}$$

$$ImpEnv = Name \xrightarrow{m} Implementation$$

where

$$inv\text{-}ImpEnv(ie) \quad \triangleq$$
$$\exists m \in ImpName \xrightarrow{m} \mathbf{N} \cdot \mathsf{dom}\, m = \mathsf{dom}\, ie \wedge$$
$$\forall i \in \mathsf{dom}\, ie \cdot \forall sc \in superClasses(ie(i)) \cdot$$
$$sc \in \mathsf{dom}\, ie \wedge m(sc) < m(i) \wedge vars(ie(sc)) \subseteq vars(ie(i))$$

In the above structure, the fields *vars* and *sig* store all attributes for that class, including the inherited ones, while the field *bodies* stores only the new operations. The invariant assures that an implementation has all variables from its superclasses, with the same types. It also assures the absence of loops in the hierarchy, like *inv-TypeEnv*.

A complete environment joins together specifications, implementations, and variables:

$$\begin{array}{lrcl} Env :: & tEnv & : & TypeEnv \\ & iEnv & : & ImpEnv \\ & vEnv & : & VarEnv \end{array}$$

After defining the necessary domains, we can start the definition of some useful functions. The first one defines the subtype relation:

$$subtype : TypeName \times TypeName \times TypeEnv \to \mathbf{B}$$

$$subtype(t1, t2, te) \quad \triangleq \quad t2 = \textsc{Void} \vee t1 = t2 \vee$$
$$\exists t \in superTypes(te(t1)) \cdot subtype(t, t2, te)$$

Notice that the invariant of *TypeEnv* assures the correctness of the recursion. It is easy to check that, for any given environment, *subtype* is transitive, reflexive and anti-symmetrical.

The subclass relationship is defined in a similar way:

$$subclass : ImpName \times ImpName \times ImpEnv \to \mathbf{B}$$

$$subclass(i1, i2, ie) \ \triangleq \ i1 = i2 \vee \exists i \in superClasses(ie(i1)) \cdot subclass(i, i2, te)$$

Another useful predicate describes compatibility between arities; it returns true if $s1$ is a sub-arity of $s2$ in an environment te.

$$subArity : Arity \times Arity \times TypeEnv \to \mathbf{B}$$

$$\begin{array}{l} subArity(a1, a2, te) \ \triangleq \\ \quad subtype(result(a1), result(a2), te) \wedge \mathsf{len}\, parms(a1) = \mathsf{len}\, parms(a2) \wedge \\ \quad \forall i \in \mathsf{inds}\, parms(a1) \cdot subtype(parms(a2)(i), parms(a1)(i), te) \end{array}$$

The next predicate checks compatibility between signatures:

$$subSignature : TypeName \times TypeName \times TypeEnv \to \mathbf{B}$$

$$\begin{array}{l} subSignature(tn1, tn2, te) \ \triangleq \\ \quad \mathsf{let}\ s1 = sig(te(tn1)), s2 = sig(te(tn2))\ \mathsf{in} \\ \quad \mathsf{dom}\, s2 \subseteq \mathsf{dom}\, s1 \wedge \forall f \in \mathsf{dom}\, s2 \cdot subArity(s1(f), s2(f), te) \end{array}$$

The above predicates have the following informal translation: A function $f1$ is compatible (subArity) with $f2$ if the body of $f2$ can be a call to $f1$, passing straight the input parameters into $f1$ and returning the result value from $f1$. A type $t1$ is compatible with a type $t2$ if $t1$ exports all operations that $t2$ exports, and each of these operations in $t1$ is compatible with its equivalent in $t2$. Again, it is an easy exercise to prove that both predicates are reflexive, transitive, and anti-symmetrical.

In order to ease the definition of the denotational functions, we define some auxiliary functions to manipulate syntactic structures. Most of them only change formats, from some kind of sequence to some kind of map.

$$paramType : VarDeclaration^* \to TypeName^*$$

$$paramType(p) \ \triangleq \ \mathsf{if}\ p = [\,]\ \mathsf{then}\ [\,]\ \mathsf{else}\ \mathsf{cons}(type(\mathsf{hd}\, p), paramType(\mathsf{tl}\, p))$$

$$fh : FunctionHead \to Signature$$

$$fh(h) \ \triangleq \ \{name(h) \mapsto mk\text{-}Arity(result(h), paramType(parameters(h)))\}$$

$$heads : FunctionHead^* \to Signature$$

$$heads(h) \ \triangleq \ \mathsf{if}\ h = [\,]\ \mathsf{then}\ \{\,\}\ \mathsf{else}\ fh(\mathsf{hd}\, h) \cup heads(\mathsf{tl}\, h)$$

$$heads' : Function^* \to Signature$$

$$heads'(h) \ \triangleq \ \mathsf{if}\ h = [\,]\ \mathsf{then}\ \{\,\}\ \mathsf{else}\ fh(header(\mathsf{hd}\, h)) \cup heads'(\mathsf{tl}\, h)$$

$$st : SuperType^* \rightarrow (TypeName \xrightarrow{m} OpName\text{-}\mathsf{set})$$

$$st(s) \quad \triangleq \quad \mathsf{if}\ s = [\,]\ \mathsf{then}\ \{\,\}\ \mathsf{else}\ \{name(\mathsf{hd}\ s) \mapsto \mathsf{elems}\ redef(\mathsf{hd}\ s)\} \cup st(\mathsf{tl}\ s)$$

$$vd : VarDeclaration^* \rightarrow VarEnv$$

$$vd(v) \quad \triangleq \quad \mathsf{if}\ v = [\,]\ \mathsf{then}\ \{\,\}\ \mathsf{else}\ \{name(\mathsf{hd}\ v) \mapsto type(\mathsf{hd}\ v)\} \cup vd(\mathsf{tl}\ v)$$

Notice the use of the $\cup$ operation to join those maps. This assures that programs with an identifier declared twice have no correct denotation.

An important feature of our language is that it allows free recursion among types. In order to accommodate this requirement, the "type checking" is divided in two passes. The first pass (done by functions *type* and *types*) checks only inheritance, and creates a *TypeEnv* with all types in the program. Then the second pass applies the function *checkTypes* to ensure that all operation redefinitions are consistent.

$$type : TypeDec \times TypeEnv \rightarrow Type$$

$$\begin{array}{l}
type(t, te) \quad \triangleq \\
\quad \mathsf{let}\ sts = st(superTypes(t)), f = heads(functions(t))\ \mathsf{in} \\
\quad \mathsf{if}\ \mathsf{dom}\, sts \subseteq \mathsf{dom}\ te \wedge \bigcup \mathsf{rng}\ sts \subseteq \mathsf{dom}\, f \\
\quad \mathsf{then}\ \mathsf{let}\ f' = \bigcup_{st \in \mathsf{dom}\ sts}(sts(st) \triangleleft sig(te(st)))\ \mathsf{in} \\
\qquad mk\text{-}Type((f' \cup f), \mathsf{dom}\, sts) \\
\quad \mathsf{else}\ \textsc{Error}
\end{array}$$

This function deserves some explanations. The if condition is a conjunction of two terms: the first one checks that all supertypes are already declared, while the second term assures that the new type supplies definitions for all functions whose names belong to a redefinition list. If that test is satisfied, we can create the new type. In f' we join all operations from supertypes that were not redefined, and so the union of f' with f has all operations of the new type (inherited and new ones). Again, the use of unions over maps assures there is no name clash among the operations inherited from different supertypes, or between inherited and new ones.

When we have a sequence of specifications, we can create an appropriate environment with the following function:

$$types : TypeDec^* \times TypeEnv \rightarrow TypeEnv$$

$$\begin{array}{l}
types(tl, te) \quad \triangleq \\
\quad \mathsf{if}\ tl = [\,] \\
\quad \mathsf{then}\ te \\
\quad \mathsf{else}\ \mathsf{let}\ te' = te \cup \{name(\mathsf{hd}\ tl) \mapsto type(\mathsf{hd}\ tl, te)\}\ \mathsf{in} \\
\qquad types(\mathsf{tl}\ tl, te')
\end{array}$$

The function *checkTypes* ensures that every type is compatible with its supertypes:[9]

[9] In fact, by the way we build each type, it is already known that every type provides all operations exported by its supertypes. Moreover, operations which were not redefined are trivially compatible. Therefore, the only real check performed by *checkTypes* is that the redefinitions are consistent.

$$checkTypes : TypeEnv \to \mathbf{B}$$

$$checkTypes(te) \quad \triangleq$$
$$\quad \forall n \in \mathsf{dom}\, te \cdot \forall st \in superTypes(te(n)) \cdot subSignature(n, st, te)$$

The above function only checks compatibility with direct supertypes. The following lemma assures that every type is compatible with all its supertypes:

Lemma 1 *In a correct type environment, every type has a signature that is a subsignature of the signature of its supertypes. Formally:*

$$checkTypes(te) \;\Rightarrow\; \forall t1, t2 \in \mathsf{dom}\, te \cdot subtype(t1, t2, te)$$
$$\Rightarrow\; subSignature(t1, t2, te)$$

Proof: The proof is by (strong) induction over the natural numbers assigned to each type name in the invariant of *TypeEnv*. If $t1 = t2$ or $t2 = \textsc{Void}$, the result is trivial. Otherwise, if $t1$ is a subtype of $t2$, there must be a $t \in superTypes(te(t1))$ such that t is a subtype of $t2$. By the invariant of te, the number assigned to t is smaller than the number assigned to $t1$, and so we can use the induction hypothesis to conclude that $subSignature(t, t2, te)$. By the definition of *checkTypes* we also know that $subSignature(t1, t, te)$. Finally, from the transitivity of *subSignature*, we infer that $subSignature(t1, t2, te)$. ▮

Our next step is to build semantic functions for implementations. To allow free recursion among implementations we follow the same approach adopted with specifications: a first pass to collect all implementations and take care of inheritance (that can not be recursive), and a second pass checking correctness.

The first pass is done by the functions *inherImp* and *imps*. The first function returns a new environment including the given implementation. The implementation is stored with the fields *vars* and *functions* augmented in order to include the inherited attributes.

$$inherImp : ImplementationDec \times ImpEnv \to ImpEnv$$

$$inherImp(mk\text{-}ImplementationDec(n, t, sc, v, f), ie) \quad \triangleq$$
$$\begin{array}{l}
\quad \mathsf{if}\ \mathsf{elems}\, sc \subseteq \mathsf{dom}\, ie \\
\quad \mathsf{then}\ \mathsf{let}\ v' = allVars(sc, ie) \cup vd(v)\ \mathsf{in} \\
\quad\quad \mathsf{let}\ sig = allFuncs(sc, ie) \dagger heads'(f)\ \mathsf{in} \\
\quad\quad ie \cup \{n \mapsto mk\text{-}Implementation(t, \mathsf{elems}\, sc, v', sig, f)\} \\
\quad \mathsf{else}\ \textsc{Error}
\end{array}$$

Notice the use of $\cup$ and $\dagger$ defining whether overwriting is allowed or not. In the above definition, *allVars* and *allFuncs* are auxiliary functions that return all inherited attributes of an implementation, given its superclasses:

$$allVars : Name^* \times ImpEnv \to VarEnv$$

$$allVars(nl, ie) \quad \triangleq \quad \mathsf{if}\ nl = [\,]\ \mathsf{then}\ \{\,\}\ \mathsf{else}\ vars(ie(\mathsf{hd}\, nl)) \cup allVars(\mathsf{tl}\, nl, ie)$$

$$allFuncs : Name^* \times ImpEnv \to Signature$$

$$allFuncs(nl, ie) \quad \triangleq \quad \mathsf{if}\ nl = [\,]\ \mathsf{then}\ \{\,\}\ \mathsf{else}\ sig(ie(\mathsf{hd}\, nl)) \cup allFuncs(\mathsf{tl}\, nl, ie)$$

The function *imps* joins a list of implementations:

$$imps : ImplementationDec^* \times ImpEnv \to ImpEnv$$

$$imps(il, ie) \quad \triangleq \quad \mathsf{if}\ il = [\,]\ \mathsf{then}\ ie\ \mathsf{else}\ imps(\mathsf{tl}\ il, inherImp(\mathsf{hd}\ il, ie))$$

Finished with the first pass, we turn our attention to how to check the implementations. In an implementation we must assure three facts: that all superclasses have appropriate types, that the implementation provides all functions (with correct arity) specified by its type, and that every new function is correctly typed.

$$typeImp : ImpName \times TypeEnv \times ImpEnv \to \mathbf{B}$$

$$\begin{array}{l} typeImp(n, te, ie) \quad \triangleq \\ \quad \mathsf{let}\ mk\text{-}Implementation(t, sc, v, sg, b) = ie(n)\ \ \mathsf{in} \\ \quad (\forall s \in sc \cdot subtype(t, type(ie(s)), te)) \wedge \\ \quad sg = sig(te(t)) \wedge \forall f \in \mathsf{elems}\ b \cdot typeFunction(f, te, ie, n) \end{array}$$

$$checkImps : TypeEnv \times ImpEnv \to \mathbf{B}$$

$$checkImps(te, ie) \quad \triangleq \quad \forall n \in \mathsf{dom}\ ie \cdot typeImp(n, te, ie)$$

To check a function, we check if its body satisfies its result type; the body is evaluated in an environment augmented with the formal parameters:

$$typeFunction : Function \times TypeEnv \times ImpEnv \times ImpName \to \mathbf{B}$$

$$\begin{array}{l} typeFunction(mk\text{-}Function(h, b), te, ie, n) \quad \triangleq \\ \quad \mathsf{let}\ env = mk\text{-}Env(te, ie, vd(parameters(h)))\ \mathsf{in} \\ \quad typeExp(b, env, n, result(h)) \end{array}$$

Now we need a function that checks if an expression satisfies a type in a given environment.

$$typeExp : Expression \times Env \times ImpName \times TypeName \to \mathbf{B}$$

$$\begin{array}{l} typeExp(ex, env, in, t) \quad \triangleq \\ \quad \mathsf{cases}\ ex\ \ \mathsf{of} \\ \quad \begin{array}{rl} \textsc{Self} \to & subtype(type(iEnv(env)(in)), t, tEnv(env)) \\ \mathsf{nil} \to & \mathsf{true} \\ mk\text{-}NewExp(n) \to & subtype(type(iEnv(env)(n)), t, tEnv(env)) \\ mk\text{-}VarExp(n) \to & typeVarExp(ex, env, in, t) \\ mk\text{-}Assignment(n, ex') \to & typeAssignment(ex, env, in, t) \\ mk\text{-}WhileExp(ex', ex'') \to & typeWhileExp(ex, env, in, t) \\ mk\text{-}IfExp(ex', ex'', ex''') \to & typeIfExp(ex, env, in, t) \\ mk\text{-}FunctionCall(r, o, p) \to & typeFunctionCall(ex, env, in, t) \\ mk\text{-}Block(dec, exl) \to & typeBlock(ex, env, in, t) \end{array} \\ \quad \mathsf{end} \end{array}$$

The expression SELF has always the implementation type, and nil satisfies all types. A new object has the type given by its implementation.

The type of a variable is given by the variable environment, if the variable is local, or by the implementation environment, in the case of an instance variable:

$$typeVarExp : VarExp \times Env \times ImpName \times TypeName \to \mathbf{B}$$

$$\begin{array}{l} typeVarExp(mk\text{-}VarExp(n), env, in, t) \quad \triangleq \\ \quad \mathsf{let}\ mk\text{-}Env(te, ie, ve) = env\ \mathsf{in} \\ \quad \mathsf{let}\ vt = \mathsf{if}\ n \in \mathsf{dom}\, ve\ \mathsf{then}\ ve(n)\ \mathsf{else}\ vars(ie(in))(n)\ \mathsf{in} \\ \quad subtype(vt, t, te) \end{array}$$

An assignment is correct if the type of the expression satisfies the type of the variable:

$$typeAssignment : Assignment \times Env \times ImpName \times TypeName \to \mathbf{B}$$

$$\begin{array}{l} typeAssignment(mk\text{-}Assignment(n, ex), env, in, t) \quad \triangleq \\ \quad \mathsf{let}\ mk\text{-}Env(te, ie, ve) = env\ \mathsf{in} \\ \quad \mathsf{let}\ vt = \mathsf{if}\ n \in \mathsf{dom}\, ve\ \mathsf{then}\ ve(n)\ \mathsf{else}\ vars(ie(in))(n)\ \mathsf{in} \\ \quad typeExp(ex, env, in, vt) \wedge subtype(vt, t, te) \end{array}$$

To type check a while expression, we only need to check its sub-expressions. Notice that a while expression only satisfies the type VOID.

$$typeWhileExp : WhileExp \times Env \times ImpName \times TypeName \to \mathbf{B}$$

$$\begin{array}{l} typeWhileExp(mk\text{-}WhileExp(c, d), env, in, t) \quad \triangleq \\ \quad typeExp(c, env, in, \mathrm{VOID}) \wedge typeExp(d, env, in, \mathrm{VOID}) \wedge t = \mathrm{VOID} \end{array}$$

An if expression is similar, but we must verify the result type:

$$typeIfExp : IfExp \times Env \times ImpName \times TypeName \to \mathbf{B}$$

$$\begin{array}{l} typeIfExp(mk\text{-}IfExp(ex, ex', ex''), env, in, t) \quad \triangleq \\ \quad typeExp(ex, env, in, \mathrm{VOID}) \wedge typeExp(ex', env, in, t) \wedge typeExp(ex'', env, in, t) \end{array}$$

A function call is the most complex expression to type. First, we must check if the receiver exports an appropriate method. Then, the real parameters are checked against the formal ones. If everything goes fine, the type of the expression is the result type of the method.

$$typeFunctionCall : FunctionCall \times Env \times ImpName \times TypeName \to \mathbf{B}$$

$$\begin{array}{l} typeFunctionCall(mk\text{-}FunctionCall(ex, n, exl), env, in, t) \quad \triangleq \\ \quad \mathsf{let}\ te = tEnv(env)\ \mathsf{in} \\ \quad \exists rt \in \mathsf{dom}\, te \cdot \\ \quad\quad typeExp(ex, env, in, rt) \wedge n \in \mathsf{dom}\, sig(te(rt)) \wedge \\ \quad\quad \mathsf{let}\ mk\text{-}Arity(result, parms) = sig(te(rt))(n)\ \mathsf{in} \\ \quad\quad subtype(result, t, te) \wedge \mathsf{len}\, exl = \mathsf{len}\, parms \wedge \\ \quad\quad \forall i \in \mathsf{inds}\, exl \cdot typeExp(exl(i), env, in, parms(i)) \end{array}$$

In order to get the type of a block, we introduce its variable declarations in the variable environment, check if all expressions have a correct type in the new environment, and check if the last expression satisfies the given type.

$$
\begin{aligned}
& \mathit{typeBlock} : \mathit{Block} \times \mathit{Env} \times \mathit{ImpName} \times \mathit{TypeName} \to \mathbf{B} \\
& \mathit{typeBlock}(\textit{mk-Block}(dec, exl), env, in, t) \quad \triangleq \\
& \quad \mathsf{let}\ ve' = vEnv(env) \dagger vd(dec)\ \mathsf{in} \\
& \quad \mathsf{let}\ env' = \mu(env, vEnv \mapsto ve')\ \mathsf{in} \\
& \quad \mathit{typeExp}(exl(\mathsf{len}\, exl), env', in, t) \wedge \forall ex \in \mathsf{elems}\, exl \cdot \mathit{typeExp}(ex, env', in, \textsc{Void})
\end{aligned}
$$

Finally, we can put everything together to type check a whole program. The next function returns **true** if the given program is correctly typed. The only predefined type is VOID, that has no supertypes and no operations. There is also a predefined implementation VOID, without operations or variables, which is used as the implementation of Self inside the main expression.

$$
\begin{aligned}
& \mathit{typeProgram} : \mathit{Program} \to \mathit{TypeName} \\
& \mathit{typeProgram}(\textit{mk-Program}(sp, im, e)) \quad \triangleq \\
& \quad \mathsf{let}\ ty = \textit{mk-Type}(\{\,\}, \{\,\})\ \mathsf{in} \\
& \quad \mathsf{let}\ te = \mathit{types}(sp, \{\textsc{Void} \mapsto ty\})\ \mathsf{in} \\
& \quad \mathsf{let}\ imp = \textit{mk-Implementation}(\textsc{Void}, \{\,\}, \{\,\}, \{\,\}, [\,])\ \mathsf{in} \\
& \quad \mathsf{let}\ ie = \mathit{imps}(im, \{\textsc{Void} \mapsto imp\})\ \mathsf{in} \\
& \quad \mathsf{let}\ env = \textit{mk-Env}(te, ie, \{\,\})\ \mathsf{in} \\
& \quad \mathit{checkTypes}(te) \wedge \mathit{checkImps}(te, ie) \wedge \mathit{typeExp}(e, env, \textsc{Void}, \textsc{Void})
\end{aligned}
$$

An important property of the above functions is given in the following lemmas. These lemmas assure that inherited functions preserve their correctness.

Lemma 2 *Whenever a function is correct in an implementation sn, it is also correct in any implementation n which has sn as a superclass.*

$$
\begin{aligned}
& \mathit{checkTypes}(te) \wedge \mathit{checkImps}(te, ie) \wedge sn \in \mathit{superClasses}(ie(n)) \Rightarrow \\
& \quad \mathit{typeFunction}(F, te, ie, sn) \;\Rightarrow\; \mathit{typeFunction}(F, te, ie, n)
\end{aligned}
$$

Proof: Notice that, by the definition of *typeFunction*, our lemma can be reduced to:

$$
\begin{aligned}
& \mathit{checkTypes}(te) \wedge \mathit{checkImps}(te, ie) \wedge sn \in \mathit{superClasses}(ie(n)) \Rightarrow \\
& \quad \mathit{typeExp}(E, env, sn, t) \;\Rightarrow\; \mathit{typeExp}(E, env, n, t)
\end{aligned}
$$

We prove the above formula by induction over the structure of the expression E. Notice that the only expressions that use the parameter *in* in their type checking are SELF, *VarExp*, and *Assignment*. If the expression is SELF, the result follows by transitivity of *subtype* and the fact that, in a correct environment,

$$
sn \in \mathit{superClasses}(ie(n)) \;\Rightarrow\; \mathit{subtype}(\mathit{type}(ie(n)), \mathit{type}(ie(sn)), te)
$$

(see definition of *typeImp*). If the expression is an assignment or a variable, the result follows from the invariant of *ImpEnv*, that ensures that an implementation always has all variables from its superclasses, with the same types. ∎

Lemma 3 *Whenever a function is correct in an implementation n, it is also correct in any subclass of n.*

$$
\begin{aligned}
& \mathit{checkTypes}(te) \wedge \mathit{checkImps}(te, ie) \wedge \mathit{subclass}(sn, n, ie) \Rightarrow \\
& \quad \mathit{typeFunction}(F, te, ie, n) \;\Rightarrow\; \mathit{typeFunction}(F, te, ie, sn)
\end{aligned}
$$

Proof: The proof is by induction on the length of the class hierarchy. If $sn = n$ the result is trivial. Otherwise, there is a superclass (let us call it c) from sn that is a subclass of n. By the induction hypothesis, $\mathit{typeFunction}(F, te, ie, n) \Rightarrow \mathit{typeFunction}(F, te, ie, c)$, and from the lemma above, $\mathit{typeFunction}(F, te, ie, c) \Rightarrow \mathit{typeFunction}(F, te, ie, sn)$. ▮

4 Execution of School Programs

After finishing with type correctness, we are going now to define formally the execution, or the "meaning", of School programs. An interesting point about the language School is that its execution semantics is completely independent of type declarations. So, the only definitions we are going to use from the previous section are the domains *ImpEnv* and *Implementation*, and the function *imps*.

As usual in denotational description of programming languages, we define some domains describing the state of a computation, and then denote operations and programs as functions mapping old states into new ones. To reflect the referential semantics of object oriented programming languages, all objects are accessed through unique identifiers (or pointers), that here will be represented by natural numbers. We reserve 0 to denote the object Nil.

$$\mathit{ObId} = \mathbf{N}$$

$$\textsc{Nil} = 0$$

An Object has a local state, that associates instance variable names with their values, and a reference to its class, that is used to access its methods.

$$\mathit{LocalMemory} = \mathit{Name} \xrightarrow{m} \mathit{ObId}$$

$$\begin{aligned} \mathit{Object} :: \ & \mathit{state} : \mathit{LocalMemory} \\ & \mathit{class} : \mathit{ImpName} \end{aligned}$$

Finally, all objects different from Nil are stored in a global memory.

$$\mathit{GlobalMemory} = \mathit{ObId} \xrightarrow{m} \mathit{Object}$$

where

$$\begin{aligned} & \mathit{inv\text{-}GlobalMemory}(g) \ \triangleq \\ & \quad \textsc{Nil} \notin \mathsf{dom}\, g \wedge \forall o \in \mathsf{rng}\, g \cdot \mathsf{rng}\, \mathit{state}(o) \subseteq \mathsf{dom}\, g \cup \{\textsc{Nil}\} \end{aligned}$$

The invariant states that Nil is the only primitive value. Therefore, it has no representation in the memory. Moreover, every *ObId* stored in an instance variable must be represented in the memory, unless it is Nil.

Besides objects, we also need to store classes. Classes contain methods for their objects.

$$\mathit{ClassEnv} = \mathit{ImpName} \xrightarrow{m} \mathit{Class}$$

$$\mathit{Class} = \mathit{OpName} \xrightarrow{m} \mathit{Method}$$

Our next domains are used to describe the semantics of methods (operations). A method acts over a global state, a receiver and a list of parameters, and returns a new global state and a result. To model this behavior we use the following declarations:

$$\mathit{Method} = \mathit{GlobalMemory} \times \mathit{ObId} \times \mathit{ObId}^* \rightarrow \mathit{MethodResult}$$

$$\begin{aligned} \mathit{MethodResult} :: \ & \mathit{state} : \mathit{GlobalMemory} \\ & \mathit{value} : \mathit{ObId} \end{aligned}$$

The last domains we need are for expressions and lists of expressions. Expressions do not take parameters; on the other hand, they access and modify not only a global state, but also local variables. Expressions return values, while lists of expressions return lists of values.

$$Exp = GlobalMemory \times LocalMemory \times ObId \rightarrow ExpResult$$

$$\begin{aligned} ExpResult \;::\; global &: GlobalMemory \\ local &: LocalMemory \\ value &: ObId \end{aligned}$$

$$ExpList = GlobalMemory \times LocalMemory \times ObId \rightarrow ExpListResult$$

$$\begin{aligned} ExpListResult \;::\; global &: GlobalMemory \\ local &: LocalMemory \\ value &: ObId^* \end{aligned}$$

Our next step are the denotational functions, that is, functions that take a piece of program and return its denotation. The first one is a function to take care of expressions:

$$codeExp : Expression \times ClassEnv \times ImpEnv \rightarrow Exp$$

$$\begin{aligned} & codeExp(ex, ce, ie) \quad \triangleq \\ & \quad \textsf{cases } ex \textsf{ of} \\ & \qquad \begin{aligned} \textsc{Self} &\rightarrow \lambda g, l, s \cdot mk\text{-}ExpResult(g, l, s) \\ \textsf{nil} &\rightarrow \lambda g, l, s \cdot mk\text{-}ExpResult(g, l, \textsc{Nil}) \\ mk\text{-}Assignment(n, ex') &\rightarrow codeAssignment(ex, ce, ie) \\ mk\text{-}WhileExp(ex', ex'') &\rightarrow codeWhileExp(ex, ce, ie) \\ mk\text{-}IfExp(ex', ex'', ex''') &\rightarrow codeIfExp(ex, ce, ie) \\ mk\text{-}VarExp(n) &\rightarrow codeVarExp(ex, ce, ie) \\ mk\text{-}FunctionCall(r, o, p) &\rightarrow codeFunctionCall(ex, ce, ie) \\ mk\text{-}NewExp(n) &\rightarrow codeNewExp(ex, ce, ie) \\ mk\text{-}Block(dec, exl) &\rightarrow codeBlock(ex, ce, ie) \end{aligned} \\ & \quad \textsf{end} \end{aligned}$$

If the expression is nil or Self, then its denotation is a function that does not change the memory, and returns the appropriate value. All other expressions are handled by more specific functions, that we will see now.

$$codeVarExp : VarExp \times ClassEnv \times ImpEnv \rightarrow Exp$$

$$\begin{aligned} & codeVarExp(mk\text{-}VarExp(n), ce, ie) \quad \triangleq \\ & \quad \lambda g, l, s \cdot \textsf{if } n \in \textsf{dom } l \\ & \qquad\qquad \textsf{then } mk\text{-}ExpResult(g, l, l(n)) \\ & \qquad\qquad \textsf{else let } mystate = state(g(s)) \textsf{ in} \\ & \qquad\qquad\qquad mk\text{-}ExpResult(g, l, mystate(n)) \end{aligned}$$

If the variable is local, then one only has to get its value. Otherwise, Self is used to access the state of the object in the global memory.[10]

[10] Notice that in a real implementation the decision about whether a variable is local or not could be done in "compile time", that is, outside the lambda expression. However, our goal here is simplicity, not "efficiency".

$$codeIfExp : IfExp \times ClassEnv \times ImpEnv \rightarrow Exp$$

$$\begin{array}{l}
codeIfExp(mk\text{-}IfExp(ex, ex', ex''), ce, ie) \quad \triangleq \\
\quad \textsf{let } e = codeExp(ex, ce, ie), e' = codeExp(ex', ce, ie), e'' = codeExp(ex'', ce, ie) \textsf{ in} \\
\quad \lambda g, l, s \cdot \\
\quad\quad \textsf{let } mk\text{-}ExpResult(g', l', r) = e(g, l, s) \textsf{ in} \\
\quad\quad \textsf{if } r \neq \mathrm{NIL} \textsf{ then } e'(g', l', s) \textsf{ else } e''(g', l', s)
\end{array}$$

$$codeWhileExp : WhileExp \times ClassEnv \times ImpEnv \rightarrow Exp$$

$$\begin{array}{l}
codeWhileExp(mk\text{-}WhileExp(ex, ex'), ce, ie) \quad \triangleq \\
\quad \textsf{let } e = codeExp(ex, ce, ie), e' = codeExp(ex', ce, ie) \textsf{ in} \\
\quad \mu\,[\lambda x \cdot \\
\quad\quad \lambda g, l, s \cdot \\
\quad\quad\quad \textsf{let } mk\text{-}ExpResult(g', l', r) = e(g, l, s) \textsf{ in} \\
\quad\quad\quad \textsf{if } r = \mathrm{NIL} \\
\quad\quad\quad \textsf{then } mk\text{-}ExpResult(g', l', \mathrm{NIL}) \\
\quad\quad\quad \textsf{else let } mk\text{-}ExpResult(g'', l'', r) = e'(g', l', s) \textsf{ in} \\
\quad\quad\quad\quad x(g'', l'', s)\,]
\end{array}$$

As usual, we use the least fixed-point operator μ to express the semantics of the while expression.

Let us now see how to create new objects. The function *declare* creates a new local memory containing the variables of a given variable environment, with all variables initialized to Nil:

$$declare : VarEnv \rightarrow LocalMemory$$

$$declare(ve) \quad \triangleq \quad \{n \mapsto \mathrm{NIL} \mid n \in \textsf{dom } ve\}$$

The function *allot* allocates a new "position" in a global memory:

$$\begin{array}{l}
allot\ (g\colon GlobalMemory)\ i\colon ObId \\
\textsf{post } i \notin \textsf{dom } g \wedge i \neq \mathrm{NIL}
\end{array}$$

Finally, the code to create new objects is given by:

$$codeNewExp : NewExp \times ClassEnv \times ImpEnv \rightarrow Exp$$

$$\begin{array}{l}
codeNewExp(mk\text{-}NewExp(n), ce, ie) \quad \triangleq \\
\quad \textsf{let } o = mk\text{-}Object(declare(vars(ie(n))), n) \textsf{ in} \\
\quad \lambda g, l, s \cdot \\
\quad\quad \textsf{let } i = allot(g) \textsf{ in} \\
\quad\quad mk\text{-}ExpResult(g \cup \{i \mapsto o\}, l, i)
\end{array}$$

The code of a block is the code of its expression list, executed in an environment augmented with the local declarations. Notice the manipulation of local memories. First, we use the overwrite operator to create a memory l', so all new variables get the value *Nil*. Then, to create the final local memory (l'''), we eliminate from l'' the variables declared inside the block, and combine the result with the initial memory. In this way, any variable that has been redeclared restores its previous value (from l), while variables visible inside the block get the new value (from l''). The value returned by the block is the value of its last expression.

$$
\begin{array}{l}
\mathit{codeBlock} : \mathit{Block} \times \mathit{ClassEnv} \times \mathit{ImpEnv} \to \mathit{Exp}\\[1ex]
\mathit{codeBlock}(\textit{mk-Block}(d, el), ce, ie) \quad \triangleq\\
\quad \mathsf{let}\ exl = \mathit{codeExpList}(el, ce, ie)\ \mathsf{in}\\
\quad \mathsf{let}\ nv = \mathit{declare}(vd(d))\ \mathsf{in}\\
\quad \lambda g, l, s \cdot\\
\qquad \mathsf{let}\ l' = l \dagger nv\ \mathsf{in}\\
\qquad \mathsf{let}\ \textit{mk-ExpListResult}(g', l'', vl) = exl(g, l', s)\ \mathsf{in}\\
\qquad \mathsf{let}\ l''' = l \dagger ((\mathsf{dom}\, nv) \lhd l'')\ \mathsf{in}\\
\qquad \textit{mk-ExpResult}(g', l''', vl(\mathsf{len}\, el))
\end{array}
$$

The function vd is defined in section 3, and only converts a list of variable declarations to a variable environment. The function *codeExpList* is shown below.

$$
\begin{array}{l}
\mathit{codeExpList} : \mathit{Expression}^* \times \mathit{ClassEnv} \times \mathit{ImpEnv} \to \mathit{ExpList}\\[1ex]
\mathit{codeExpList}(el, ce, ie) \quad \triangleq\\
\quad \mathsf{if}\ el = [\,]\\
\quad \mathsf{then}\ \lambda g, l, s \cdot \textit{mk-ExpListResult}(g, l, [\,])\\
\quad \mathsf{else}\ \mathsf{let}\ ex = \mathit{codeExp}(\mathsf{hd}\, el, ce, ie)\ \mathsf{in}\\
\qquad \mathsf{let}\ exl = \mathit{codeExpList}(\mathsf{tl}\, el, ce, ie)\ \mathsf{in}\\
\qquad \lambda g, l, s \cdot\\
\qquad\quad \mathsf{let}\ \textit{mk-ExpResult}(g', l', v) = ex(g, l, s)\ \mathsf{in}\\
\qquad\quad \mathsf{let}\ \textit{mk-ExpListResult}(g'', l'', vl) = exl(g', l', s)\ \mathsf{in}\\
\qquad\quad \textit{mk-ExpListResult}(g'', l'', \mathsf{cons}(v, vl))
\end{array}
$$

An assignment to a local variable is straightforward. In the case of an instance variable, we determine the object identifier through Self, and then do the necessary update:

$$
\begin{array}{l}
\mathit{codeAssignment} : \mathit{Assignment} \times \mathit{ClassEnv} \times \mathit{ImpEnv} \to \mathit{Exp}\\[1ex]
\mathit{codeAssignment}(\textit{mk-Assignment}(n, e), ce, ie) \quad \triangleq\\
\quad \mathsf{let}\ ex = \mathit{codeExp}(e, ce, ie)\ \mathsf{in}\\
\quad \lambda g, l, s \cdot\\
\qquad \mathsf{let}\ \textit{mk-ExpResult}(g', l', v) = ex(g, l, s)\ \mathsf{in}\\
\qquad \mathsf{if}\ n \in \mathsf{dom}\,(l)\\
\qquad \mathsf{then}\ \textit{mk-ExpResult}(g', l' \dagger \{n \mapsto v\}, v)\\
\qquad \mathsf{else}\ \mathsf{let}\ st = \mathit{state}(g'(s)) \dagger \{n \mapsto v\}\ \mathsf{in}\\
\qquad\quad \mathsf{let}\ o = \mu(g'(s), \mathit{state} \mapsto st)\ \mathsf{in}\\
\qquad\quad \textit{mk-ExpResult}(g' \dagger \{s \mapsto o\}, l', v)
\end{array}
$$

Our last case is a function call. The particularity here is the late-binding, that is, the fact that the called function is taken from the class of the object resulting from the first expression; this class can only be known at “run-time”, that is, inside the lambda expression.

$$
\begin{array}{l}
\mathit{codeFunctionCall} : \mathit{FunctionCall} \times \mathit{ClassEnv} \times \mathit{ImpEnv} \to \mathit{Exp}\\[1ex]
\mathit{codeFunctionCall}(\textit{mk-FunctionCall}(e, op, p), ce, ie) \quad \triangleq\\
\quad \mathsf{let}\ ex = \mathit{codeExp}(e, ce, ie)\ \mathsf{in}\\
\quad \mathsf{let}\ exl = \mathit{codeExpList}(p, ce, ie)\ \mathsf{in}\\
\quad \lambda g, l, s \cdot\\
\qquad \mathsf{let}\ \textit{mk-ExpResult}(g', l', i) = ex(g, l, s)\ \mathsf{in}\\
\qquad \mathsf{let}\ \textit{mk-ExpListResult}(g'', l'', vl) = exl(g', l', s)\ \mathsf{in}\\
\qquad \mathsf{if}\ i = \mathrm{N{\scriptstyle IL}}\\
\qquad \mathsf{then}\ \bot\\
\qquad \mathsf{else}\ \mathsf{let}\ c = ce(class(g'(i)))\ \mathsf{in}\\
\qquad\quad \mathsf{let}\ m = c(op)\ \mathsf{in}\\
\qquad\quad \mathsf{let}\ \textit{mk-MethodResult}(g''', v) = m(g'', i, vl)\ \mathsf{in}\\
\qquad\quad \textit{mk-ExpResult}(g''', l'', v)
\end{array}
$$

Inside this function, c is the class of the receiver object, and m is the selected method. Notice that if the receiver object is Nil, the meaning of the call is undefined.

Finished the treatment of expressions, our next step is how to code an operation. First we define an auxiliary function to bind the real parameters with the formal parameters in a new local memory.

$$
\begin{array}{l}
\mathit{bindParms} : \mathit{VarDeclaration}^* \times \mathit{ObId}^* \to \mathit{LocalMemory}\\[1ex]
\mathit{bindParms}(vd, il) \quad \triangleq\\
\quad \mathsf{if}\ vd = [\,]\ \mathsf{then}\ \{\,\}\ \mathsf{else}\ \{name(\mathsf{hd}\ vd) \mapsto \mathsf{hd}\ il\} \cup \mathit{bindParms}(\mathsf{tl}\ vd, \mathsf{tl}\ il)
\end{array}
$$

The code for a method is the code of its body, preceded by the binding of parameters.

$$
\begin{array}{l}
\mathit{codeFunction} : \mathit{Function} \times \mathit{ClassEnv} \times \mathit{ImpEnv} \to \mathit{Method}\\[1ex]
\mathit{codeFunction}(\textit{mk-Function}(h, b), ce, ie) \quad \triangleq\\
\quad \mathsf{let}\ ex = \mathit{codeExp}(b, ce, ie)\ \mathsf{in}\\
\quad \lambda g, s, p \cdot\\
\qquad \mathsf{let}\ l = \mathit{bindParms}(parameters(h), p)\ \mathsf{in}\\
\qquad \mathsf{let}\ \textit{mk-ExpResult}(g', l', v) = ex(g, l, s)\ \mathsf{in}\\
\qquad \textit{mk-MethodResult}(g', v)
\end{array}
$$

The code of a list of functions is straightforward:

$$
\begin{array}{l}
\mathit{codeFunctions} : \mathit{Function}^* \times \mathit{ClassEnv} \times \mathit{ImpEnv} \to \mathit{Class}\\[1ex]
\mathit{codeFunctions}(fl, ce, ie) \quad \triangleq\\
\quad \mathsf{if}\ fl = [\,]\\
\quad \mathsf{then}\ \{\,\}\\
\quad \mathsf{else}\ \mathsf{let}\ cf = \{name(header(\mathsf{hd}\ fl)) \mapsto \mathit{codeFunction}(\mathsf{hd}\ fl, ce, ie)\}\ \mathsf{in}\\
\qquad cf \cup \mathit{codeFunctions}(\mathsf{tl}\ fl, ce, ie)
\end{array}
$$

Now is time to face implementations. In order to allow the kind of recursion that School requires, we will need the least fixed-point operator. That is why we have been using a class environment parameter in all code functions, when in fact these functions are supposed to build the class environment. Let us see how to code one implementation:

$$codeImp : ImplementationDec \times ClassEnv \times ImpEnv \rightarrow Class$$

$$\begin{array}{l} codeImp(mk\text{-}ImplementationDec(n, t, sc, v, f), ce, ie) \quad \triangleq \\ \quad joinParents(sc, ce) \dagger codeFunctions(f, ce, ie) \end{array}$$

$$joinParents : ImpName^* \times ClassEnv \rightarrow Class$$

$$\begin{array}{l} joinParents(nl, ce) \quad \triangleq \\ \quad \mathsf{if}\ nl = [\,]\ \mathsf{then}\ \{\,\}\ \mathsf{else}\ ce(\mathsf{hd}\, nl) \cup joinParents(\mathsf{tl}\, nl, ce) \end{array}$$

So, *joinParents* only puts together all methods from the parent classes; if the program is correct we know there will be no clashes. Then, *codeImp* joins the inherited methods with the local ones. The use of the overwrite operator (†) assures that, in case of redefinitions, the class gets the new methods.

A list of implementations is coded as follows:

$$codeImps : ImplementationDec^* \times ClassEnv \times ImpEnv \rightarrow ClassEnv$$

$$\begin{array}{l} codeImps(il, ce, ie) \quad \triangleq \\ \quad \mathsf{if}\ il = [\,] \\ \quad \mathsf{then}\ ce \\ \quad \mathsf{else\ let}\ ce' = ce \dagger \{name(\mathsf{hd}\, il) \mapsto codeImp(\mathsf{hd}\, il, ce, ie)\}\ \mathsf{in} \\ \qquad codeImps(\mathsf{tl}\, il, ce', ie) \end{array}$$

Finally, we can give the meaning of a whole program. A program executes its main expression in an environment with all types, all implementations, and no local variables. The initial global memory includes an object wherein the main expression is evaluated; this object has implementation VOID (therefore without variables or operations). The output of our program is an object identifier, plus the global memory containing this object and any other objects referred by it.

$$codeProgram : Program \rightarrow MethodResult$$

$$\begin{array}{l} codeProgram(mk\text{-}Program(sp, im, e)) \quad \triangleq \\ \quad \mathsf{let}\ imp = mk\text{-}Implementation(\textsc{Void}, \{\,\}, \{\,\}, \{\,\}, [\,])\ \mathsf{in} \\ \quad \mathsf{let}\ ie = imps(im, \{\textsc{Void} \mapsto imp\})\ \mathsf{in} \\ \quad \mathsf{let}\ ce = \mu\,[\lambda x \cdot codeImps(im, x, ie)]\ \mathsf{in} \\ \quad \mathsf{let}\ g = \{1 \mapsto mk\text{-}Object(\{\,\}, \textsc{Void})\}\ \mathsf{in} \\ \quad \mathsf{let}\ ex = codeExp(e, ce, ie)\ \mathsf{in} \\ \quad \mathsf{let}\ mk\text{-}ExpResult(g', l, v) = ex(g, \{\,\}, 1)\ \mathsf{in} \\ \quad mk\text{-}MethodResult(g', v) \end{array}$$

5 Soundness of the Type System of School

In this section we present a formal proof that School is statically typed, according with the definition stated in section 1. As most definitions and proofs along this section will need to refer to the type and implementation environments (*TypeEnv* and *ImpEnv*) of a program P, we will assume they are available under the names TE and IE, respectively. According, all definitions and proofs must be understood "with respect to given environments TE and IE". More formally, we assume that the whole section is inside the following scope:

$\mathsf{let}\ ty = \textit{mk-Type}(\{\,\}, \{\,\})\ \mathsf{in}$
$\mathsf{let}\ TE = types(specs(P), \{\textsc{Void} \mapsto ty\})\ \mathsf{in}$
$\mathsf{let}\ imp = \textit{mk-Implementation}(\textsc{Void}, \{\,\}, \{\,\}, \{\,\}, [\,])\ \mathsf{in}$
$\mathsf{let}\ IE = imps(impl(P), \{\textsc{Void} \mapsto imp\})\ \mathsf{in}$
$checkTypes(TE) \wedge checkImps(TE, IE) \ \Rightarrow\ \ldots$

First some definitions. We say that an object identifier i *satisfies* a type t if and only if the type associated with the class of i is a subtype of t:

$$satisfy : ObId \times TypeName \times GlobalMemory \to \mathbf{B}$$

$$\begin{aligned}
&satisfy(i, t, g) \ \ \triangleq \\
&\quad \mathsf{if}\ i = \textsc{Nil} \\
&\quad \mathsf{then\ true} \\
&\quad \mathsf{else\ let}\ t' = type(IE(class(g(i))))\ \mathsf{in} \\
&\qquad subtype(t', t, TE)
\end{aligned}$$

Notice the assumption that Nil satisfies any type. An important property of the above definition is that, as *subtype* is transitive, whenever i satisfies a type t, it also satisfies any supertype of t.

A local memory is *consistent* with a variable environment if and only if all its variables have values satisfying their types:

$$local\_cons : LocalMemory \times VarEnv \times GlobalMemory \to \mathbf{B}$$

$$local\_cons(l, ve, g) \ \ \triangleq \ \ \mathsf{dom}\ ve \subseteq \mathsf{dom}\ l \wedge \forall n \in \mathsf{dom}\ ve \cdot satisfy(l(n), ve(n), g)$$

A global memory is *consistent* if and only if all its objects have states which are consistent with the variable environment for their classes. In other words, all instance variables contain objects with "appropriate" types.

$$global\_cons : GlobalMemory \to \mathbf{B}$$

$$\begin{aligned}
&global_cons(g) \ \ \triangleq \ \ \forall o \in \mathsf{rng}\ g \cdot \\
&\qquad \mathsf{let}\ ve = vars(IE(class(o)))\ \mathsf{in} \\
&\qquad local_cons(state(o), ve, g)
\end{aligned}$$

An important property of a program is that it never changes the class of an object. We can express this fact with the following predicate:

$$keep\_classes : GlobalMemory \times GlobalMemory \to \mathbf{B}$$

$$keep\_classes(g, g') \ \ \triangleq \ \ \mathsf{dom}\ g \subseteq \mathsf{dom}\ g' \wedge \forall i \in \mathsf{dom}\ g \cdot class(g(i)) = class(g'(i))$$

Now let us see what we can say about methods. The intuitive property of a correct method is that, whenever it is called with appropriate parameters (including Self), it returns an appropriate result. Moreover, one can expect that a correct method keeps the consistency of the memory.

$$
\begin{aligned}
&\mathit{met\_sat} : \mathit{Method} \times \mathit{Arity} \times \mathit{ImpName} \to \mathbf{B}\\
&\mathit{met\_sat}(m, \mathit{mk\text{-}arity}(\mathit{result}, \mathit{parms}), \mathit{in}) \quad \triangleq\\
&\quad \forall g \in \mathit{GlobalMemory}, s \in \mathsf{dom}\, g, p \colon \mathit{ObId}^* \cdot\\
&\qquad \mathit{global\_cons}(g) \wedge (\forall i \in \mathsf{inds}\, \mathit{parms} \cdot \mathit{satisfy}(p(i), \mathit{parms}(i), g)) \wedge\\
&\qquad \mathit{subclass}(\mathit{class}(g(s)), \mathit{in}, \mathit{IE}) \wedge m(g, s, p) \neq \bot \ \Rightarrow\\
&\qquad \mathsf{let}\ \mathit{mk\text{-}MethodResult}(g', v) = m(g, s, p)\ \mathsf{in}\\
&\qquad \mathit{global\_cons}(g') \wedge \mathit{keep\_classes}(g, g') \wedge \mathit{satisfy}(v, \mathit{result}, g')
\end{aligned}
$$

The test $m(g, s, p) \neq \bot$ indicates that we only care about terminating methods; whenever a method call does not terminate, we can say nothing about it. The condition

$$\mathit{subclass}(\mathit{class}(g(s)), \mathit{in}, \mathit{IE})$$

indicates that Self can be of any subclass of the class wherein the method is defined (in).

Lemma 4 *Whenever a method satisfies an arity a, it satisfies any super-arity of a.*

$$\mathit{subArity}(a, \mathit{sa}, \mathit{TE}) \wedge \mathit{met\_sat}(m, a, \mathit{in}) \ \Rightarrow\ \mathit{met\_sat}(m, \mathit{sa}, \mathit{in})$$

Proof: Define $\mathit{mk\text{-}Arity}(\mathit{parms}, \mathit{result}) = a$ and $\mathit{mk\text{-}Arity}(\mathit{parms}', \mathit{result}') = \mathit{sa}$. From the definition of $\mathit{subArity}$, we know that

$$\mathit{subtype}(\mathit{result}, \mathit{result}', \mathit{TE}) \wedge \mathit{subtype}(\mathit{parms}'(i), \mathit{parms}(i), \mathit{TE})$$

Therefore, we have $\mathit{satisfy}(x, \mathit{parms}'(i), g) \ \Rightarrow\ \mathit{satisfy}(x, \mathit{parms}(i), g)$ and $\mathit{satisfy}(x, \mathit{result}, g) \Rightarrow\ \mathit{satisfy}(x, \mathit{result}', g)$. The above implications, together with the definition of $\mathit{met\_sat}$, close the proof. ▌

We say that a class satisfies a type if it provides methods for all operations exported by the type, and each method is correct with respect to the corresponding arity:

$$
\begin{aligned}
&\mathit{class\_sat} : \mathit{Class} \times \mathit{TypeName} \times \mathit{ImpName} \to \mathbf{B}\\
&\mathit{class\_sat}(c, t, \mathit{in}) \quad \triangleq\\
&\quad \mathsf{let}\ \mathit{sg} = \mathit{sig}(\mathit{TE}(t))\ \mathsf{in}\\
&\quad \mathsf{dom}\, \mathit{sg} \subseteq \mathsf{dom}\, c \wedge \forall f \in \mathsf{dom}\, \mathit{sg} \cdot \mathit{met\_sat}(c(f), \mathit{sg}(f), \mathit{in})
\end{aligned}
$$

Lemma 5 *Whenever a class satisfies a type t, it satisfies any super-type of t.*

$$\mathit{subtype}(t, \mathit{st}, \mathit{TE}) \wedge \mathit{class\_sat}(c, t, \mathit{in}) \ \Rightarrow\ \mathit{class\_sat}(c, \mathit{st}, \mathit{in})$$

Proof: By $\mathit{checkTypes}(\mathit{TE})$, we know that $\mathit{subSignature}(t, \mathit{st}, \mathit{TE})$. Define $\mathit{sg} = \mathit{sig}(\mathit{TE}(t))$ and $\mathit{sg}' = \mathit{sig}(\mathit{TE}(\mathit{st}))$. If t is a subsignature of st, then $\mathsf{dom}\, \mathit{sg}' \subseteq \mathsf{dom}\, \mathit{sg} \subseteq \mathsf{dom}\, c$. Finally, lemma 4 proves that $\mathit{met\_sat}(c(f), \mathit{sg}'(f), \mathit{in})$. ▌

Our last definition concerns environments. A class environment is correct if all its classes are correct:

$$
\begin{aligned}
&\mathit{classEnvOK} : \mathit{ClassEnv} \to \mathbf{B}\\
&\mathit{classEnvOK}(\mathit{ce}) \quad \triangleq \quad \forall n \in \mathsf{dom}\, \mathit{ce} \cdot \mathit{class\_sat}(\mathit{ce}(n), \mathit{type}(\mathit{IE}(n)), n)
\end{aligned}
$$

For reasons that will become clear in the proof of lemma 9, we do not force correct environments to have classes for all implementations (although they do).

Lemma 6 *Any expression with a correct type, when called in a correct state, returns correct states and a value that satisfies its type. Formally:*

$$\begin{array}{l}
\mathsf{let}\ env = mk\text{-}Env(TE, IE, lve)\ \mathsf{in} \\
\mathsf{let}\ ex = codeExp(E, ce, IE)\ \mathsf{in} \\
typeExp(E, env, in, t) \wedge classEnvOK(ce) \wedge ex(g, l, s) \neq \perp \wedge \\
class(g(s)) = in \wedge global_cons(g) \wedge local_cons(l, lve, g)\ \Rightarrow \\
\mathsf{let}\ mk_ExpResult(g', l', v) = ex(g, l, s)\ \mathsf{in} \\
global_cons(g') \wedge local_cons(l', lve, g') \wedge \\
keep_classes(g, g') \wedge satisfy(v, t, g')
\end{array}$$

In the above formulae, the expression E satisfies type t, and has code ex. The local variable environment is represented by lve, while in is the name of the implementation wherein the expression is written. In the antecedent of the implication, the fourth term assures that Self has a correct implementation, while the last two terms assert the consistency of the global and the local memories. The third term restricts the lemma to terminating expressions; we can say nothing about expressions that do not terminate. The consequent first states the consistency of both memories after the execution of the expression, and then that the result of the expression has the correct type.

Proof: The proof is by cases and induction over the expression structure. First notice that no expression deletes elements from the global memory, nor changes the class of an object. So, it is immediate to conclude that $keep\_classes(g, g')$, and we do not need to worry about this term in the consequent.

SELF In that case, $g' = g$, $l' = l$, so the final memories are correct. We know that $v = s$, and by *typeExp* we have that $subtype(type(IE(in)), t, TE)$. As we also know that $class(g(s)) = class(g(v)) = in$, it is immediate that v satisfies t.

nil In that case, $g' = g$, $l' = l$, and the result $v =$ NIL satisfies any type t.

$mk\text{-}VarExp(n)$ Again, $g' = g$ and $l' = l$, so we only need to prove that the result has correct type. If $n \in \mathsf{dom}\, lve$, then $n \in \mathsf{dom}\, l$ (because l is consistent with lve). But in that case $v = l(n)$, and again by consistency of l we know that $satisfy(l(n), lve(n), g)$. By definition of *typeVarExp*, we have that $subtype(lve(n), t, TE)$, and by transitivity we conclude $satisfy(l(n), t, g)$. If $n \notin \mathsf{dom}\, lve$, then $n \in \mathsf{dom}\, vars(IE(in))$ (otherwise the expression has no type at all), and the type $vt = vars(IE(in))(n)$ is a subtype of t. If we define $st = state(g(s))$, we can infer, by consistency of g, that $local\_cons(st, vars(IE(in)), g)$. Therefore, by definition of *local_cons*, the final result $v = st(n)$ satisfies vt, and therefore also satisfies t.

$mk\text{-}IfExp(ex, ex', ex'')$ Left to the reader (remember that, by the induction hypothesis, the lemma is valid for all sub-expressions).

$mk\text{-}Assignment(n, ex)$ Let g', l' and v be like in *codeAssignment*, and vt like in *typeAssignment*. By the induction hypothesis, we know that g' and l' are correct states, and that v satisfies the type vt. As vt must be a subtype of t, we have $satisfy(v, t, g')$. Now we must check that the final memories are also correct. There are two cases. If $n \in \mathsf{dom}\, lve$, the final global memory is g', which is OK. The type vt is equal to $lve(n)$. The final local memory is $l' \dagger \{n \mapsto v\}$; as we know that $satisfy(v, lve(n), g')$, we can conclude that this final memory is still correct with respect to lve and g'.

If $n \in vars(IE(in))$, then the final local memory is l', which is correct. Notice that, in *codeAssignment*, when we create the final global memory we do not change the class of any object, and only change the state of s. So, we only need to prove that the final state of this object is correct. The type vt in such case is $vars(IE(in))(n)$. Let us define $st = state(g'(s))$, $st' = st \dagger \{n \mapsto v\}$, and $gve = vars(IE(in)) = vars(IE(class(g'(s))))$. Then, by consistency of g', we can infer that $local\_cons(st, gve, g')$. This fact, together with $satisfy(v, gve(n), g')$, leads us to conclude that $local\_cons(st', gve, g')$. Therefore, the new global memory, $g' \dagger \{s \mapsto mk\text{-}Object(st', \ldots)\}$, is consistent.

$mk\text{-}NewExp(n)$ In this case, the local memory is unchanged, and the global memory receives a new element (by definition of *allot*). So, the consistency of l is maintained. As *Nil* satisfies any type, it is easy to check that $local\_cons(declare(v), v, g')$, for any environment v and memory g'. In special, for $v = vars(IE(n))$, and defining $o = mk\text{-}Object(declare(v), n)$, we have that $local\_cons(state(o), vars(IE(class(o))), g)$. Therefore, the final memory is consistent. Finally, it is immediate to check that this new object satisfies the type t.

$mk\text{-}WhileExp(ex, ex')$ The while expression is defined as a least fixed-point (lfp). The lfp of a functional F, denoted by $\mu[F]$, is given by $\bigcup_{i=0}^{\infty} F^i(\perp)$. Therefore, to prove that the lfp satisfies a property, we are going to prove that all $F^i(\perp)$ satisfy that property, using induction. Our base case gives $ex = \perp$, that trivially satisfies the lemma. In the induction step we must prove that, if x satisfies the lemma, then $F(x)$ also satisfies it. This proof is similar to the proof for an if expression, and again is left to the reader.

$mk\text{-}Block(dec, exl)$ Left to the reader.

$mk\text{-}FunctionCall(r, op, p)$ This is the most complex case. Let us define g', l', g'', l'', vl, v and i like in *codeFunctionCall*. By the induction hypothesis, we know that g', g'', l', and l'' are correct. If $i = \perp$, the whole expression evaluates to $\perp$ and the lemma is trivially true. So, let us see the case that $i \neq \perp$. Let us suppose that rt is the type whose existence is assured in *typeFunctionCall*, and let $mk\text{-}Arity(result, parms) = sig(TE(rt))(op)$. Again by the induction hypothesis, $satisfy(i, rt, g')$, and $\forall p \in \mathsf{inds}\, parms \cdot satisfy(vl(p), parms(p), g'')$.

Define $rc = class(g'(i))$. If $rc \notin \mathsf{dom}\, ce$, all its methods are undefined, and the whole expression evaluates to $\perp$. Otherwise, following the definition of $classEnvOk(ce)$, we have that $class\_sat(ce(rc), type(IE(rc)), rc)$, and by lemma 5, $class\_sat(ce(rc), rt, rc)$. By definition of *class_sat*, we conclude that $met\_sat(ce(rc)(op), sig(TE(rt))(op), rc)$. As we have all the antecedents of *met_sat*, we can conclude that g''' is consistent, and that $satisfy(v, result, g''')$. Finally, as $subtype(result, t, TE)$, we have that $satisfy(v, t, g''')$.

Although, for technical reasons, we have had to consider the situation wherein the class environment ce does not contain a definition for the class of the receiver object (rc), such case never occurs — see lemma 9. Also notice that, as we know that $op \in \mathsf{dom}\, sig(TE(rt))$ and that the class rc satisfies the type rt, we have that $op \in sig(IE(rc))$. Therefore, there is no possibility of "message not understood" errors.

∎

Lemma 7 *Correct methods satisfy their arities.*

let $F = \textit{mk-Function}(h, b)$ in
let $a = \textit{mk-Arity}(\textit{result}(h), \textit{paramType}(\textit{parameters}(h)))$ in
let $mt = \textit{codeFunction}(F, ce, IE)$ in
$\textit{classEnvOk}(ce) \wedge \textit{typeFunction}(F, TE, IE, in) \;\Rightarrow\; \textit{met\_sat}(mt, a, in)$

Proof: Let us define l, g', l' and v according to *codeFunction*. Because the real parameters satisfy their types (hypothesis of *met_sat*), we have that l is consistent with $lve = vd(parms(a))$. Moreover, if the function is correct, its body must be correct, that is,

$$\textit{typeExp}(\textit{body}(F), env, in, \textit{result}(a))$$

(where $env = \textit{mk-Env}(TE, IE, lve)$). By lemma 3, its body is also correct in any subclass of in, that is, $\textit{subclass}(sn, in, IE) \;\Rightarrow\; \textit{typeExp}(\textit{body}(F), env, sn, \textit{result}(a))$. Therefore, the hypothesis in *met_sat* give us all conditions needed to apply lemma 6 over the body of the method (with substitutions $[E \mapsto \textit{body}(F), in \mapsto sn, t \mapsto \textit{result}(a)]$). ▮

Lemma 8 *Correct classes satisfy their types.*

let $I = IE(n)$ in
let $c = \textit{codeImp}(I, ce, IE)$ in
$\textit{classEnvOk}(ce) \wedge \textit{typeImp}(n, TE, IE) \;\Rightarrow\; \textit{class\_sat}(c, \textit{type}(I), n)$

Proof: A class, in order to satisfy its type, must provide methods for all functions of the type, and these methods must satisfy their correspondent arities (see *class_sat*). According to *codeImp*, a class includes all functions declared in its implementation, plus all functions inherited from all superclasses. Moreover, *typeImp* assures that such set of functions has the same signature than the type of the implementation (remember that the field *sig* in an implementation includes the inherited operations). Therefore, a correct class has all operations from its type. Now we must prove that those methods are correct. If the method is inherited from a class *sc*, *classEnvOk* tells us that it is correct in *sc*, and so in any subclass of *sc*. If the method is locally declared, *typeImp* assures that its definition is correctly typed, and applying lemma 7 we conclude that the method satisfies the appropriate arity. ▮

Lemma 9 *The class environment of a correct program is correct. Moreover, it provides definitions for all implementations in a program. Formally:*

let $ce = \mu\,[\lambda x \cdot \textit{codeImps}(\textit{impl}(P), x, IE)]$ in
$\textit{classEnvOk}(ce) \wedge \mathsf{dom}\, IE \subseteq \mathsf{dom}\, ce$

(where P is the program that originates TE and IE — see the beginning of this section).

Proof: Again we have a definition using fixed-points, and again we are going to use the technique used in lemma 6, for expressions **while**. In the present case,

$$F = \lambda x \cdot \textit{codeImps}(\textit{impl}(P), x, IE)$$

- The base case is $ce = \perp$. In this case, the class environment defines no classes at all, and so it trivially satisfies *classEnvOk*. Notice that, if we have defined that a correct class environment must provide definitions for all implementations, we would be unable to prove this step.

- The induction step is that *codeImps* preserves the correctness of class environments. Formally:

  let $ce' = \textit{codeImps}(\textit{impl}(P), ce, IE)$ in
  $\textit{classEnvOk}(ce) \;\Rightarrow\; \textit{classEnvOk}(ce')$

According to our general assumptions, P is correct. Therefore, all its implementations are correct. Our previous lemma assures that correct implementations generate correct classes. Therefore, all classes added to ce are correct, and then ce' satisfies *classEnvOk*.

We still have to prove that $\mathsf{dom}\, IE \subseteq \mathsf{dom}\, ce$. As *codeImps* adds definitions for all implementations in its first parameter, and $impl(P)$ contains all implementations in the program P, we have that $\mathsf{dom}\, IE \subseteq \mathsf{dom}\, codeImps(impls(P), ce, IE)$. Therefore, $F^i(\perp)$ provides, for $i \geq 1$, definitions for all implementations in the program, and so does the fixed-point of F. ▮

Finally, we are able to prove our main result:

Lemma 10 *Correct programs run without type errors.*

Proof: According to the definition of *typeProgram*, a correct program has a correct type environment, and a correct implementation environment. Moreover, its main expression is correctly typed. The global memory g has only one object, and it is consistent, because the implementation Void has no instance variables to be wrong. The local memory, empty, is consistent with the empty variable environment for that expression. Lemma 9 assures that the class environment ce is also correct. Therefore we have all conditions to apply lemma 6 to the main expression of the program. ▮

6 Some Extensions to School

In previous sections we have developed a simple programming language, mainly in order to facilitate the proofs. Here we will show how some other facilities can be incorporated into School. Although we do not present complete formalisms for these extensions, we will address some questions about such formalizations.

Structural Subtyping

As we have pointed out in section 3, School uses a kind of name compatibility for its hierarchy. That means that a type is considered a subtype of another one only when there are declarations asserting that. That is the approach followed by some important OO languages, like C++ [10] and Simula [3]. Now we will see how School can incorporate structural subtyping. In this approach, a type is considered a subtype of another one whenever their declarations are compatible.

Following the definitions in section 3, we can see that the main property we want for subtyping is that it must imply subsignatures. Therefore, one can be moved to change the definition of the *subtype* function to:

$$subtype : TypeName \times TypeName \times TypeEnv \to \mathbf{B}$$

$$subtype(t1, t2, te) \quad \triangleq \quad subSignature(t1, t2, te)$$

assuming that *subArity* now uses this new definition for *subtype*. Although very attractive, this definition has one problem. If we define two types like:

```
Type A
  Function X () :  A
End A
Type B
```

```
  Function X () :  B
End B
```

then $subtype(A, B, te)$ is undefined, because of the infinite recursion.

One way to avoid this problem, adopted in [2], is to get the largest solution of the recursive definition, instead of the usual least fixed-point. However, this approach, changing the usual meaning of a recursive equation, is not compatible with the semantics of VDM adopted here. Moreover, it does not conform to our natural idea of a recursive definition. So, here we will adopt a different solution, avoiding the recursion at all. In order to do that, first we define what we call a hierarchy:

$$Hierarchy = TypeName \xrightarrow{m} TypeName\text{-}\mathsf{set}$$

A hierarchy stores, for each type, the set of all its supertypes. Then we adapt the definitions of *subSignature* and *subArity* to use a hierarchy:

$$subArity' : Arity \times Arity \times Hierarchy \to \mathbf{B}$$

$$\begin{aligned} &subArity'(a1, a2, h) \quad \triangleq \\ &\quad result(a2) \in h(result(a1)) \wedge \mathsf{len}\, parms(a1) = \mathsf{len}\, parms(a2) \wedge \\ &\quad \forall i \in \mathsf{inds}\, parms(a1) \cdot parms(a1)(i) \in h(parms(a2)(i)) \end{aligned}$$

$$subSignature' : TypeName \times TypeName \times Hierarchy \times TypeEnv \to \mathbf{B}$$

$$\begin{aligned} &subSignature'(tn1, tn2, h, te) \quad \triangleq \\ &\quad \mathsf{let}\ s1 = sig(te(tn1)), s2 = sig(te(tn2))\ \mathsf{in} \\ &\quad \mathsf{dom}\, s2 \subseteq \mathsf{dom}\, s1 \wedge \forall f \in \mathsf{dom}\, s2 \cdot subArity'(s1(f), s2(f), h) \end{aligned}$$

We say that a hierarchy is consistent when all its relationships satisfy the sub-signature criterion:

$$consistent : Hierarchy \times TypeEnv \to \mathbf{B}$$

$$consistent(h, te) \quad \triangleq \quad \forall t1 \in \mathsf{dom}\, h \cdot \forall t2 \in h(t1) \cdot subSignature'(t1, t2, h, te)$$

Finally, we say that a type is subtype of another one if there is a consistent hierarchy wherein that relationship holds:

$$subtype : TypeName \times TypeName \times TypeEnv \to \mathbf{B}$$

$$subtype(t1, t2, te) \quad \triangleq \quad \exists h \colon Hierarchy \cdot consistent(h, te) \wedge t2 \in h(t1)$$

The above definition avoids recursion, and it captures the intuitive notion of structural subtyping. Moreover, it is an easy task to prove that this definition satisfies our main criterion, that subtyping implies subsignatures. Finally, as this new definition is also transitive, we can prove that all results of section 5 are still valid if we adopt this definition in all type-checking functions of section 3.

Another interesting result is that the above concept of subtyping is maximal, in the sense that it is the largest subtyping relation that avoids "message not understood" errors. Any relation which includes a pair <type,subtype> not accepted by the structural definition results in an unsafe type system.

I/O Facilities

For the sake of completeness, here we show how primitive I/O operations can be included in School.

The main difficulty is the absence of built-in types in our language. To solve that, we will use the same solution adopted for condition expressions, which is the identification of NIL with the boolean False and any value different from NIL with True. Following this approach, all I/O will be done through sequences of booleans.

$$File = \mathbf{B}^*$$

The state of a program now has, besides the main memory, an input and an output file.

$$\begin{array}{rrl} GlobalState :: & mem & : GlobalMemory \\ & input & : File \\ & output & : File \end{array}$$

An input expression, with type VOID, reads a boolean from the input and returns NIL or any value different from NIL (Self, for instance). A write expression takes one argument, evaluates it and appends the appropriate value to the output file.

It should be noticed that any sequential binary data are, in fact, a sequence of boolean values. Therefore, the above commands are enough to access any kind of data. Again we emphasize that our goal is to provide completeness, and not efficiency.

Private Methods

A *private* method is a method that is not visible outside the class where it is defined. Many OO languages offer a special mechanism for this facility, e.g. C++ [10] and Eiffel [17].

School, with its separation between implementations and specifications, presents a natural way to support private methods. Whenever a routine is defined in an implementation but not in its correspondent specification, it is private. However, in order to support this simple idea, we must change a little our language. With the present definition, all "local" functions are called through Self. If a function is not exported, it is not present in the type of Self, and therefore can not be called.

We solve this problem introducing in the language a facility for "early-binding" calls. The syntax can be conventional: just the name of the function with eventual parameters, without a receiver. Only local methods can be called that way. The semantics is also the conventional semantics for early-binding routine calls in conventional languages. With this extension, public routines can be called with late-binding and early-binding, while private routines must be called with early-binding. The relationship between this facility and inheritance is straightforward. Because what we inherit is the denotation of a method, all early-bindings of a method are kept when it is inherited. Even if a called method is redefined, the inherited calling method will still use its old version.

7 Conclusions

We have shown how we can prove the type-correctness of an OOPL, using a pragmatical (but formal) concept of type. Our method deals with most typical features of OOPLs, like multiple inheritance, recursive types, recursive classes, late-binding, etc.

We start with a denotational description of the language. Such description includes a definition of valid programs (concerning type checking, variable declarations, etc), as well as the semantics of valid programs. Then we proceed to prove that correct programs run without "message not understood" errors.

A disadvantage of our approach is that it is too operational. Some object-oriented concepts, like data abstraction, do not have an independent description. Many times, recursion is avoided with the use of indirection, like in the definition of *TypeEnv* (types do not refer to other types, but to names).

On the other hand, this more operational approach is what allows our method to deal with more realistic OOPLs. Although School is an economical language, with no fancy control structures, predefined types or libraries, the inclusion of those features would not change its main character. Apart from this simplicity, School is a quite real OOPL, including most relevant features from languages like Eiffel, Simula, Objective-C [8], or C++. In this sense, School is a more "typical" OOPL than most languages adopted in theoretical approaches, like Quest [6] or FOOP [12].

Along the work, we have also gained a better understanding of how each OO feature interacts with other ones, and how this interaction affects the type system. An interesting example concerns the separation of hierarchies of types and classes. This separation is a very attractive facility, and has been adopted in many works (e.g. [2], [16], [11]). However, our work has demonstrated that, in a statically typed language with **self**, these hierarchies can not be completely independent. Another interesting result is that structural subtyping, as stated in section 6, is the most permissible compatibility check that still assures type-correctness. It is important to note that structural subtyping assumes the antimonotonic rule for parameter passing.[11]

Although we have defined a new language to show our method, the method can be applied to most OOPLs. Unfortunately, a great number of them is not statically typed, and some are not typed at all. We hope this work can help changing this situation.

Acknowledgments

The author would like to thank Noemi Rodriguez for helpful comments on this work. This work has been partially supported by CNPq (The Brazilian Research Council).

References

[1] R. Amadio and L. Cardelli. Subtyping with recursive types. In *ACM Conference on Principles of Programming Languages*, 1991.

[2] P. America and F. van der Linden. A parallel object-oriented language with inheritance and subtyping. *Sigplan Notices*, 25(10), 1990. OOPSLA/ECOOP'90 Proceedings.

[3] G. Birtwistle, O. Dahl, B. Myhrhaug, and K. Nygaard. *Simula Begin*. Petrocelli Charter, 1975.

[4] D. Bobrow, L. DeMichiel, R. Gabriel, S. Keene, G. Kiczales, and D. Moon. *Common Lisp Object System Specification*. ANSI Common Lisp, 1988. Doc. 88–003, X3J13 Standards Committee.

[11] The *antimonotonic* rule states that, when a method is redefined in a subtype, its input parameter types must be supertypes of the original ones.

[5] L. Cardelli. A semantics of multiple inheritance. In D. MacQueen, G. Kahn, and G. Plotkin, editors, *Semantics of Data Types: International Symposium*. Springer Verlag, 1984. LNCS 173.

[6] L. Cardelli. Typeful programming. In *notes of IFIP Advanced Seminar on Formal Description of Programming Concepts*, Petropolis – Brazil, 1989.

[7] L. Cardelli and P. Wegner. On understanding types, data abstraction and polymorphism. *ACM Computing Surveys*, 17(4), 1985.

[8] B. Cox and A. Novobilski. *Object Oriented Programming: an Evolutionary Approach*. Addison-Wesley, second edition, 1991.

[9] S. Danforth and C. Tomlinson. Type theories and object-oriented programming. *ACM Computing Surveys*, 20(1):29–71, 1988.

[10] M. Ellis and B. Stroustrup. *The Annotated C++ Reference Manual*. Addison-Wesley, 1990.

[11] P. Canning et al. Interfaces for strongly-typed object-oriented programming. *Sigplan Notices*, 24(10):457–467, 1989. OOPSLA'89 Proccedings.

[12] J. Goguen and J. Meseguer. Unifying functional, object-oriented and relational programming with logical semantics. In B. Shriver and P. Wegner, editors, *Research Directions in Object-Oriented Programming*. MIT Press, Cambridge, Mass., 1987.

[13] Adele Goldberg and Dave Robson. *Smalltalk-80 : The Language and its Implementation*. Addison-Wesley, 1983.

[14] Cliff B. Jones. *Systematic Software Development using VDM*. International Series in Computer Science. Prentice Hall, second edition, 1990.

[15] B. Kristensen, O. Madsen, B. Mollen-Pederson, and K. Nygaard. Object-oriented programming in the beta programming language, 1990. Draft.

[16] C. Lunau. Separation of hierarchies in Duo-Talk. *Journal of Object-Oriented Programming*, 2(2):20–26, 1989.

[17] Bertrand Meyer. Eiffel – a language and environment for software enginnering. *The Journal of Systems and Software*, 8(3):129–46, 1988.

[18] M. Wolczko. Object-oriented languages. In C. B. Jones and R. C. Shaw, editors, *Case Studies in Systematic Software Development*, chapter 10. Prentice Hall, 1990.