# Types in School

Noemi de La Rocque Rodriguez
Departamento de Computação
UFF – Valonguinho
Niteroi – Brazil
(noemi@inf.puc-rio.br)

Roberto Ierusalimschy
Departamento de Informática
Pontifícia Universidade Católica
Rio de Janeiro – Brazil
(roberto@inf.puc-rio.br)

José Lucas Rangel
Departamento de Informática
Pontifícia Universidade Católica
Rio de Janeiro – Brazil
(rangel@inf.puc-rio.br)

## 1 Introduction

Since Smalltalk, many programming languages have tried to capture the flexibility of object-oriented languages into a static type system. Almost every object-oriented language presents its own concept of type system. Usually, these different type systems are incompatible, in the sense that a correct hierarchy in one language can not be easily translated to another language. Differences arise in points like single versus multiple inheritance (e.g. Beta [KMMPN90] versus Eiffel [Mey88a]), separate hierarchies for types and classes (e.g. Pool [AvdL90] and DuoTalk [Lun89]), possibility of redefinition of signatures of inherited methods (e.g. C++ [Str92] does not allow, while Eiffel does), and many other topics.

One of the reasons for those differences is the lack of a clear definition for the concept of "type" and the role of a "type system" in such languages. In (pure) OOPLs, 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. 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.[2] *Statically typed* languages are those that can ensure the absence of such errors by a type-checking procedure performed during compile-time.

School is a programming language designed to keep the basic semantics of object-oriented languages, mainly Smalltalk, with a secure type system. All values in School are objects. Variables do not contain objects, but references to them, and both assignment and parameter passing manipulate references. All comunication between objects are based on late-binding, and the binding depends solely on the receiver. Moreover, the type system has no influence

[1] [AC91], pp. 104
[2] What is called a "message not understood" error in Smalltalk.

over the semantics of the language.

The main rule that has guided the development of School's type system has been to avoid any construction that can cause a "message not understood error" at run-time; obeying this rule, we have tried to make the language as flexible as possible. In order to do that, School has separate hierarchies for types and classes (specifications and implementations), structural subtyping, and constrained genericity. At the end, we arrived in a terse language that can model most type facilities presented by other OO languages.

This paper explains the type system of School. The next section discusses the separation between types and classes. Section 3 explains the facilities for constrained genericity. In section 4 we show the expressiveness of the language, and how specific facilities from other languages can be modeled in School. As usual, the last section draws some conclusions.

## 2 Separation of Hierarchies

Although already pointed out by some authors (e.g. [AvdL90], [Lun89], [C+89]), the separation of types and classes (and their respective hierarchies) in OOPLs is still neglected in most works on this field. The main idea underlying this concept is the understanding of types as specifications, and classes as implementations. The type of an object is its external appearance, that is, its interface to the outside world. In School, a type declaration states all operations available in objects of that type, and the types of parameters and eventual results of each operation. For example, a point would be declared as:[3]

```
type Point is
  method x () r : Integer
  method y () r : Integer
  method moveBy (p : Point)
end Point
```

On the other hand, the class of an object dictates its internal shape, that is, its structure and the code to handle it. In School, a possible implementation for points is shown below. Notice that, after the class name (`PointCart`), we state the *external type* for objects of that class (`Point`).

```
class PointCart : Point is
  var xPos,yPos : Integer ;    -- instance variables
  method x () r : Integer is r := xPos end
  method y () r : Integer is r := yPos end
  method moveBy (p : Point) is
    xPos := xPos + p.x ;
    yPos := yPos + p.y
  end
end PointCart
```

Many modern non object-oriented languages, like Modula-2 [Wir85] and Ada [Ada83], already provide this separation, leading the way to abstract data types.

Some authors, e.g. [Mey88b] and [Str92], propose the use of inheritance as a way to achieve this separation in object-oriented languages. In these proposals, an abstract type is represented by a "virtual" or "deferred" class, wherein all operations are virtual and no variables are

[3] Because School allows a method to return multiple values, the results have names, like the parameters.

declared. A concrete representation for this type is then implemented as a subclass of it. This solution has some drawbacks. First, it needs a strong programming discipline; a compiler is unable to check if an abstract type is really abstract, or any other mistake related with this construction. Moreover, it mixes up two quite distinct relationships, namely class–subclass and specification–implementation. Finally, although such use of inheritance can simulate abstract data types, it does not separate the specification hierarchy from the implementation hierarchy. This confusion between both hierarchies is the source of several misunderstandings on the subject of inheritance.

To say that a type `A` is a *subtype* of a type `B` means that `A` is compatible, from an external point of view, with `B`. Therefore, an `A` object can be used wherever a `B` object is expected. Notice that `A` and `B` do not need to have similar implementations. On the other hand, to say that a class `A` is a *subclass* of a class `B` means that `A` inherits methods and variables from `B`. As they do not need to have compatible interfaces, `A` can freely modify the inherited features. Using the classification proposed in [WZ88], subtyping must have behavior compatibility, or at least signature compatibility, while subclassing is free to adopt cancel compatibility. Therefore, there is no compromise between the flexibility of cancel compatibility and the security of strong typing.

An example may help clarifying that point. A typical use of inheritance is to build a class `Stack` from a class `Array`. However, this use of inheritance is sometimes considered a "bad" use, because a stack does not have most array operations[4]. If one can define `Stack` as a subclass of `Array`, without being a subtype, the problem simply vanishes.

## Structural Compatibility

Usually, in OOPLs, a type is considered a subtype of another one only when it is so declared, directly or through transitivity. The independence between subtyping and subclassing allows a programming language to adopt *structural compatibility*. That means that a type is a subtype of another one if they are compatible in some way.

Following our stated main rule, we want to allow a type `A` to be a subtype of `B` as long as there is no possibility of error when using an `A` object in the place of `B`. As errors occur when a message is sent to an object which has no method for it, we can avoid them with the following definition: a type $A$ is a subtype of $B$ ($A \prec B$) if and only if, for each method $X$ in $B$, with arity $P_{B_1} \times \ldots \times P_{B_n} \to R_{B_1} \times \ldots \times R_{B_m}$, there is a method $X$ em $A$, with arity $T_{A_1} \times \ldots \times T_{A_n} \to R_{A_1} \times \ldots \times R_{A_m}$, where for all $i \leq m$, $R_{A_i} \prec R_{B_i}$ and, for all $i \leq n$, $P_{B_i} \prec P_{A_i}$. The apparent inversion in the last condition is known as the "counter-variance rule", and is needed to assure correctness [CW85]. A formal definition of subtyping, as well as a formal proof that this definition avoids type errors, can be found in [Ier92]. It is important to notice that this definition is not only sufficient to ensure absence of run-time errors, but also necessary. In any language accepting $A$ in place of $B$ with $A \not\prec B$, it is possible to write a routine that generates a run-time error.

As an example of subtyping, consider the declarations:

```
type Point1
  method x () r : Integer
  method y () r : Integer
  method moveBy (p : Point1)
  method copy () p : Point1
end Point1
```

[4] see, for instance, [KM90]

```
type Point2
  method x () r : Integer
  method y () r : Integer
  method moveBy (p : Point)
  method copy () p : Point2
end Point2
```

Then the following relationship holds: Point2 $\prec$ Point, while these do not: Point1 $\prec$ Point (incompatible parameter for method moveBy), and Point $\prec$ Point1 (missing method copy).

The main advantage of structural compatibility is its flexibility. However, this same flexibility is sometimes considered a drawback. The argument is that a type can be subtype of another one by mistake. This problem can be avoided with the use of properties [AvdL90]. A property is just a name attached to a type, intended to represent a characteristic of the type. For instance, a type Stack could have LIFO as one property. A type $A$ is then considered subtype of $B$ only if, besides the above condition, $A$ includes all properties of $B$. Because properties can be simulated through dummy functions, having the property name and without parameters or results, School does not include this facility.

## Inheritance of Classes

While the type hierarchy is automatically deduced from the types, the class hierarchy is built on explicit declarations. When a class is declared a heir to other classes it inherits, by default, all variables and methods from them.

To avoid name clashes, and also to increase the flexibility of the mechanism, a class can rename inherited features. Methods, by default, keep their names when inherited. If the programmer does not want to inherit a given method, it can be cancelled renaming it to "_", which represents a null name. Variables, on the other hand, by default are renamed to a null name, and therefore are not accessible for local methods; the programmer is free to change that. As an example, if a class P needs the methods moveBy, renamed to changePos, and x from class Point, and access to variable xPos, it would look like:

```
class P : Atype is
  heir to Point with xPos ,   -- makes xPos accessible
                     moveBy as changePos ,
                     y as _ ;   -- cancel y
               -- x is inherited by default
               -- yPos is invisible by default
 ...
end P
```

Notice that all name conflicts between inherited features must be solved through renames. It is an error to inherit two homonym features.

As already noted, a class definition specifies an external type for the class. This type defines the interface of objects of the class, in the sense that objects created from a class are typed with its external type, and therefore only methods included in this type can be correctly applied over the object. We can also define an *internal type* for a class, denoted as typeOf("class-name"), formed by the headers of all methods of the class.

In order to keep independent hierarchies, the external type of a class has no relationship with the external types of the class superclasses. The only constraint between a class and its type is that the class must offer definitions for all methods declared in its type[5]. It makes no difference whether those methods are inherited or locally defined.

Most OOPLs provide a pseudo-variable that allows the code of a method to refer to the object executing the method. This pseudo-variable is called *self* in Smalltalk, *current* in Eiffel, *this* in C++; in School it is also called *self*. The correct typing of this pseudo-variable poses some difficulties. When a method is inherited, and self has a new type in the heir, the method can become ill-typed. In order to avoid that, the type of self in the new class must be a subtype of the original type of self. In most OOPLs, as the type and class hierarchies are the same, the above requirement is always satisfied. In School, with separate hierarchies, there must be a constraint to avoid this problem.

The most general solution is to allow the programmer to explicitly declare the type of self. If a class `C`, where self has type `SelfC`, inherits from a class `SC`, where self has type `SelfSC`, the following would have to hold: `typeOf(C)` $\prec$ `SelfC`, `typeOf(SC)` $\prec$ `SelfSC`, and `SelfC` $\prec$ `SelfSC`. In spite of its generality, the above solution seems too complex. To avoid this complexity, we have adopted a more restrictive solution in School: the type of self is always the internal type of the class. That implies that the internal type of a class must be a subtype of the internal types of its superclasses, although their external types are still completely independent. This decoupling between the external type of an object and the type of self gives rise to some interesting facilities, which will be discussed in section 4.

# 3 Constrained Genericity

The polymorphism seen so far, resulting from subtyping, can be called *vertical* polymorphism. This section explains generic types, the facility provided by School to support *horizontal* polymorphism.

Generic types in School are somewhat similar to generic types in many other languages, like Ada, Eiffel, or CLU. Any type can have *type parameters*. As a typical example, a generic stack could be declared like:

```
type Stack[T] is
  method push (v : T)
  method pop () v : T
  method is_empty () e : Boolean
end Stack
```

When the type is used, real parameters must be provided. To declare a stack of integers and a stack of stack of integers we write:

```
var si : Stack[Integer] ,
    ssi : Stack[Stack[Integer]]
```

The subtype relationship is not defined for generic types, but only for its concrete instances. For instance, to know whether `Stack[Point]` $\prec$ `Stack[Point2]` holds[6], we take the signature of `Stack[Point]`, which is the signature of Stack changing `Point` for `T`, and then compare it with the signature of `Stack[Point2]`. Our approach can be contrasted with Eiffel, where `X` $\prec$ `Y` always implies `A[X]` $\prec$ `A[Y]`, independently of the definition of `A`. Unfortunately, Eiffel's approach can lead to run-time errors, as shown in the example below.

[5] That is, if a class `C` has external type `T`, we must check that `typeOf(C)` $\prec$ `T`.

[6] `Point` and `Point2` were defined in section 2.

```
var p : Point , p2 : Point2 ,
   sp : Stack[Point] , sp2 : Stack[Point2]
...
sp := sp2 ;  -- School does not allow that, but Eiffel does.
sp.push(p) ;
sp2.pop().copy() ;  -- may cause a run-time error, because
                    -- "p" may not have a "copy" operation.
```

Classes, like types, can also have type parameters. When a class can be instantiated for any real parameter, we have what is called *unconstrained genericity*. However, a class sometimes may need to apply operations over an object with a generic type. As an example, sets and bags need an equality operation over their (generic) elements. In such cases, the class will not work for types without that operation, and therefore the language needs a mechanism to express that constraint. It is interesting to note that, besides sequences and sequence-like structures, most useful generic classes need constraints.

To express a constraint over a type parameter the programmer uses the clause `where`. This clause states the operations that a type must have to be a valid real parameter, and the arity of these operations. Greater flexibility is provided by the fact that the parameter itself can be used inside those arities. With this facility, a class implementing a set may begin as:

```
class SetImp[T] : Set[T]
  where T has method equal (v : T) r : Boolean end
is
   ...
```

The above definition states that any valid type parameter for `SetImp` must have an `equal` operation, receiving one argument of that same type and returning a boolean. Any type satisfying a `where` clause is said to *conform* to it.

It is interesting to see how conformity relates to subtyping. For instance, suppose the type `T` declared with the same signature required in the where clause of sets,

```
Type T is method equal (v : T) r : Boolean end T
```

and a type `String`:

```
Type String is
  method equal (v : String) r : Boolean
  method findSubString (subS : String) index : Integer
  method concat (s : String) newS : String
  ...
end String
```

String conforms to the set parameter, but it is not a subtype of `T`. In fact, there is no single type which is a supertype of all types that conform to the set parameter. However, we still can simulate conformity using subtyping and genericity. Declare a type `T1` like:

```
Type T1[T] is method equal (v : T) r : Boolean end T1
```

Now, a type `X` conforms to the set parameter if and only if $\texttt{X} \prec \texttt{T1[X]}$. In fact, the above correspondence gives us the formal definition of conformity.

## 4 The Expressiveness of School

This section shows how the type facilities of School can be used to model some situations found in object oriented programming. As we will see, most of those situations need specific facilities in other OOPLs, sometimes weakening the type system.

Our first issue is visibility control. Smalltalk, the first paradigm of OOP, already recognized the need of *private* methods, but provided no facilities for that. Classes in Eiffel have export lists naming what is visible outside the class, while C++ allows each member of a class to be declared as *public*, *private* or *protected*. Public members are freely accessible, private members are visible only inside the class, and protected members are visible inside the class and its heirs. In School, visibility control is achieved through the separation between types and classes; any method defined in a class and not present in its type is not accessible from the outside. The differentiation between protected and private access, provided in C++, is not supported in School. Instead, the heir class controls what it needs from a superclass, using the *rename* facility.

Some languages offer a finer visibility control. C++, for instance, has the *friend* mechanism, whereby a class allows another class or function to access its private members. School allows finer granularity through a judicious use of the pseudo-variable self. The example below illustrates that use.

```
type A is
  method M1 ()
end A

type A1 is
  method M2 ()
end A1

type B2 is
  method M (a : A1)
end B2

Class CA : A is
var b : B2
  method M1 () is ... end
  method M2 () is ... end
  method M3 () is b.M(self) end
end CA
```

Because self has the internal type of a class, the self of class `CA` is compatible with type `A1`. Therefore, the call `b.M(self)` grants method `M` access to the otherwise private method `M2`.

A much more controversial issue is the counter-variance of subtyping in method parameters. Consider the example below:

```
type Point is
  method equal (p : Point) b : Boolean
  method X () x : Integer
  method Y () y : Integer
end Point
```

```
type ColoredPoint is
  method equal (p : ColoredPoint) b : Boolean
  method X () x : Integer
  method Y () y : Integer
  method color () c : Color
end ColoredPoint
```

In our type system, `ColoredPoint` is not a subtype of `Point`. Otherwise, the following program would be legal:

```
var p1,p2 : Point;
    cp : ColoredPoint;
    ...
    p1 := cp;  -- assumed valid
    If p1.equal(p2) then ...
```

The problem here is that the method `equal`, from `ColoredPoint`, is being called with a `Point` as its parameter, breaking the type security. If this method applies `color` to its parameter, an execution error will occur.

Other languages allow the relationship of subtyping to hold in the example above, through weakening the type system or introducing dynamic checking. An example of the first case is Eiffel, while the second case is exemplified by [MM90]. The advantages of such approaches is the possibility of reusing the code of a `Point` in the implementation of a `ColoredPoint`. However, this same reuse can be achieved in School. See the code below:

```
type Point'[t] is
  method equal (p : t) b : Boolean
  method X () x : Integer
  method Y () y : Integer
end Point'

class ImpPoint'[t] : Point'[t] is
  -- implementation of Points
end ImpPoint'

class ImpPoint : Point is
  heir to ImpPoint'[Point]
end ImpPoint

class ImpColoredPoint : ColoredPoint is
  heir to ImpPoint'[ColoredPoint]
  -- plus methods specific to this class
end ImpColoredPoint
```

Although a little complex, the above structure can be completely type-checked in compile time.

# 5 Conclusions

The main goal of School has been to provide a flexible type system, without relinquishing the security of static type checking or changing the usual semantics of object-oriented languages.

Therefore, its type system is completely secure, that is, any correctly typed program runs without "message not understood" errors.

Although statically checked, School provides great flexibility. Two mechanisms enhance its vertical polymorphism: one is the independence of subtyping and inheritance, and the other is structural subtyping. With these mechanisms, an object is incompatible with a variable (or parameter) only when there is a good reason, that is, the object does not have all operations specified for that variable. School also has parametric types, which provide another form of polymorphism. The union of a flexible subtyping mechanism with parametric polymorphism gives the language a great expression power, allowing the language to emulate more complex mechanisms presented in other languages.

A very interesting property of School is that types have no influence over the semantics of a program. School has two semantic definitions: a formal one in VDM, and an "operational" one, through the compiler. Both have no reference to types, and even ill-typed programs have a clear semantics. The type-checking procedure can be seen as an independent tool to assert an important property of a program, namely, the absence of type errors at run time.

## References


[AC91] R. Amadio and L. Cardelli. Subtyping with recursive types. In *ACM Conference on Principles of Programming Languages*, 1991.

[Ada83] ANSI. *Ada Programming Language*, 1983. ANSI/MIL-STD 1815A.

[AvdL90] 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.

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

[CW85] L. Cardelli and P. Wegner. On understanding types, data abstraction and polymorphism. *ACM Computing Surveys*, 17(4), 1985.

[Ier92] Roberto Ierusalimschy. A denotational approach for type-checking in object-oriented programming languages. Monografias em Ciência da Computação 12/92, PUC-Rio, Rio de Janeiro, Brazil, 1992.

[KM90] T. Korson and J. D. McGregor. Undestanding object-oriented: A unifying paradigm. *Communications of the ACM*, 33(9):40–60, 1990.

[KMMPN90] B. Kristensen, O. Madsen, B. Mollen-Pederson, and K. Nygaard. Object-oriented programming in the beta programming language, 1990. Draft.

[Lun89] C. Lunau. Separation of hierarchies in Duo-Talk. *Journal of Object-Oriented Programming*, 2(2):20–26, 1989.

[Mey88a] Bertrand Meyer. Eiffel – a language and environment for software enginnering. *The Journal of Systems and Software*, 8(3):129–46, 1988.

[Mey88b] Bertrand Meyer. *Object-Oriented Software Construction.* Prentice Hall, 1988.

[MM90] Ole Madsen and Boris Magnusson. Strong typing of object oriented languages revisited. In *ECOOP/OOPSLA'90 Proceedings*, pages 140–150, 1990.

[Str92] B. Stroustrup. *The C++ Programming Language*. Prentice-Hall, second edition, 1992.

[Wir85] N. Wirth. *Programming in Modula-2*. Springer-Verlag, third edition, 1985.

[WZ88] P. Wegner and S. Zdonik. Inheritance as an incremental modification mechanism or what like is and isn't like. In *ECOOP'88 Proceedings*, pages 55–77, 1988. LNCS 322.