A Parallel Object-Oriented Language with Inheritance and Subtyping

Pierre America
Frank van der Linden
Philips Research Laboratories
Eindhoven, the Netherlands

July 12, 1990

Abstract

This paper shows that inheritance and subtyping can be introduced advantageously into a parallel object-oriented language, POOL-I. These concepts are clearly distinguished, because they deal with different aspects of programming. In this way several problems traditionally adhering to inheritance can be solved. The language POOL-I is a parallel object-oriented language with a strong typing scheme which includes genericity and dynamic binding. A novel and particularly powerful mechanism offers the possibility to manipulate and analyse types dynamically.

1 Introduction

The parallel object-oriented language POOL [1, 4] has been designed to support writing programs for large-scale parallel machines, even without shared memory. At this moment one such machine, the 100-node DOOM (Decentralized Object-Oriented Machine) [6, 8] is already executing programs written in POOL. In this paper we discuss the language POOL-I, which is the latest member of the POOL family of languages. We shall pay particular attention to the way in which subtyping and inheritance are incorporated in POOL-I.

In POOL parallelism is introduced by giving each object a *body*. This body is a local process that the object executes in parallel with other objects. At specific places during the execution of the body the object is willing to communicate with other objects. The communication is by synchronous message passing. A message consists of a method call to another object. The object that sends the message is blocked until the receiver answers the message, executes the corresponding method, and returns the result to the caller. Both sending and receiving a message take place explicitly (by send statements and answer statements). For many objects, no specific body is specified. In this case they execute the *default body*, which answers all messages one after the other, in the order in which they arrive. It turns out that in POOL-I there is only a weak interaction between inheritance and subtyping on the one hand and parallelism on the other, see sections 2 and 5. Therefore, in this paper we do not deal extensively with aspects of parallelism.

At any moment during the execution of a program, new objects may be created. This is done by executing a routine called new. A *routine* is a kind of procedure different from a method. It is not associated with a specific object but with a class and it can be called by any object in the system without sending messages. Every class has a routine with the name new that creates a new object. Objects are never destroyed explicitly in POOL (but they may be removed by a garbage collector).

At this point we wish to draw attention to the very important distinction between types and classes. A *type* is a collection of objects that share the same externally observable behaviour. That means that in deciding whether an object belongs to a type, the only important aspects are which messages the object answers (and sends), in which order, and which relationships exist between the arguments and results of those messages. On the other hand, a *class* is a collection of objects that have exactly the same internal structure, that is, the same instance variables, the same body, and the same methods. One could say that a class describes how its instances are built, and a type describes how its elements can be used. Distinguish-

This work was done in the context of ESPRIT Basic Research Action 3020: *Integration*

ing these concepts is just as vital as distinguishing between the specification and the implementation of an abstract data type. Of course, the internal structure of an object determines its external behaviour, and therefore class membership determines type membership, but not the other way round. This distinction between types and classes is reflected in the distinction between subtyping and inheritance: subtyping is an inclusion relationship between types, and inheritance is a form of code sharing between classes. For more details, see [2, 3].

In the following sections we concentrate first on subtyping. Section 2 introduces the basic concepts of types and of the subtyping relationship. Section 3 discusses generic types and their interaction with subtyping. Then, in section 4 we present the powerful mechanisms for dynamic type manipulation present in POOL-I. Next we direct our attention towards inheritance, which is dealt with in section 5. Finally, section 6 draws some conclusions from our work.

2 Types and subtypes

As already said above, a type is a collection of objects that have the same behaviour, in so far as that behaviour can be observed by sending messages. In POOL-I, every variable and every parameter and result of a method or routine is typed, which means that it can only refer to objects that belong to a certain type. By looking at these types, the reader of a program can determine the scope of applicability of a method and the range of possible results, and a compiler can ascertain the absence of a certain class of programming errors.

One could say that a type is a specification of the behaviour of its elements. However, it does not necessarily specify this behaviour in complete detail, but it can let certain possibilities remain undecided. Thus there may be another type that is more specific: a subtype. We say that a type σ is a subtype of the type τ when each object of type σ is also an object of type τ. The subtype relationship is important in two situations: assignments and parameter passing.

In POOL the behaviour of objects is determined by the messages they can answer. The messages correspond to methods, which can be distinguished by their names, the number and types of the arguments, and the type of the result, if present. This collection of method names with their parameter and result types comprises the *signature* of the type.

Example 1: The type `Person`.

```
TYPE Person
METHOD get_name () :  String
METHOD get_address () :  String
END Person

TYPE Employee
METHOD get_name () : String
METHOD get_address () : String
METHOD get_salary () : Int
END Employee
```

We see that the type `Employee` is a subtype of `Person`: An object is an element of type `Person` precisely if it has the methods `get_name` and `get_address` with no parameters and with `String` as their result type. All objects of type `Employee` have these methods, so they are also elements of the type `Person`. Objects of type `Employee` are required to have an additional method `get_salary`, and therefore `Person` is not a subtype of `Employee`.

We have already seen that POOL-I makes a distinction between types and classes: A class is a collection of objects that look exactly the same on the inside. We say that a class *implements* a type σ when each object of the class has type σ. There may be many classes that implement a certain type and many types that are implemented by a certain class. In fact, if a class implements a type, it automatically implements all the supertypes of this type. The following class implements the type `Person`:

Example 2: The class `My_Person`.

```
CLASS My_Person
NEWPAR (name, initial_address : String)
VAR address : String := initial_address

METHOD get_name () : String
BEGIN RESULT name      END get_name

METHOD get_address () : String
BEGIN RESULT address   END get_address

METHOD put_address (a : String) : My_Person
BEGIN address := a; RESULT SELF   END put_address

END My_Person
```

Each instance of the class `My_Person` has one instance variable, `address`, and two so-called `new`-parameters, `name` and `initial_address`. An object can change its instance variables at any time (by an assignment), but the `new`-parameters are fixed at the creation of the object. As their name suggests, they are the parameters of the routine `new` of the current class, which can be called by an expression like `My_Person.new("Laura", "Rome")`, the result of which is a newly created and initialized object of class `My_Person`.

Since the class My_Person has the methods get_name and get_address with the correct parameter and result types, it implements the type Person. In fact, POOL-I automatically associates with every class a type with the same name, which can be specified by collecting the headers of all the methods defined in the class. In this way, saying that the *class* My_Person implements the type Person amounts to the same thing as saying that the *type* My_Person is a subtype of Person. For the convenience of the reader of a program, this fact can be made explicit in the class definition (in which case the compiler can also check it).

Another convenient thing is that these simple put and get methods are included automatically when requested by certain keywords. Thus the above class definition is completely equivalent to the following one:

```
CLASS My_Person < Person
NEWPAR (name GETTABLE, initial_address : String)
VAR address GETTABLE PUTTABLE : String
                              := initial_address
END My_Person
```

There are more things that can be used to distinguish types. For instance, not all methods may be answerable at any moment, or the results may depend on the order of calling. The designer of the type may want to write a specification that gives more information than the signature only. Ideally, such a specification language should at least have the power of first-order predicate logic [3]. However, it is inherently impossible for a compiler to check the subtyping relationship if the specification language is so powerful. Therefore instead of first-order logic the specification of a type in POOL-I is augmented with a collection of *properties*, which are just identifiers.

Example 3: The type Int_Stack (integer stack) and the class AIS (array integer stack).

```
TYPE Int_Stack
PROPERTY LIFO %% Last in, first out
METHOD get () : Int
METHOD put (Int) : Int_Stack
END Int_Stack

CLASS AIS
VAR a := Array(Int).new(1,0)

METHOD get () : Int
BEGIN IF a@ub = 0
      THEN RESULT NIL
      ELSE RESULT a@high %% also decreases ub
      FI
END get
METHOD put (n: Int) : AIS
BEGIN a@high := n; %% increase ub and place n
                   %% at high end
      RESULT SELF
END put
PROPERTY LIFO
END AIS
```

Here we see that the class AIS implements the type Int_Stack, because it has the required methods get and put with the right argument and result types and the property LIFO. As far as the compiler is concerned, there is no special meaning associated with the identifier LIFO; in particular it has no means to check that the item that has been entered into the stack last is the one that will be retrieved first. This interpretation exists only in the mind of the programmer, who should explain it to the readers (and users) of his types and classes by means of comments. Ideally, such a property identifier serves as an abbreviation for a formal specification of some aspect of an object's behaviour. The only point where the compiler pays attention to these properties is in determining whether one type is a subtype of another one.

Example 4: The type Int_Bag.

```
TYPE Int_Bag
METHOD get () : Int
METHOD put (Int) : Int_Bag
END Int_Bag
```

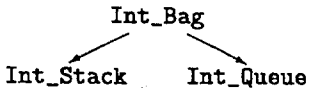
Like a stack, a bag is used to store and retrieve objects. The only difference is that the specification of a bag does not say anything about the order in which elements are retrieved. Therefore the type Int_Bag has the same signature as the type Int_Stack, but it does not have the LIFO property. Therefore the type Int_Stack is a subtype of the type Int_Bag and the class AIS also implements the type Int_Bag.

Example 5: The type Int_Queue.

```
TYPE Int_Queue
PROPERTY FIFO %% First in, first out
METHOD get () : Int
METHOD put (Int) : Int_Queue
END Int_Queue
```

The type Int_Queue only distinguishes itself from the type Int_Stack by having the object property FIFO instead of LIFO: they have exactly the same signature (and they are therefore both subtypes of Int_Bag). Nevertheless, they are different types and neither of them is a subtype of the other one. In particular, the class AIS does not implement the type Queue. We can

express the subtyping relationships in the following diagram:



Example 6: The class `Blocking_Stack` (stack that delays when empty).

```
CLASS Blocking_Stack
VAR a := Array(Int).new(1,0)

METHOD get () : Int
BEGIN RESULT a@high %% also decreases ub
END get

METHOD put (n: Int) : Blocking_Stack
BEGIN a@high := n;  %% increase ub and place n
                    %% at high end
      RESULT SELF
END put

BODY DO IF #a = 0
        THEN ANSWER (put)
        ELSE ANSWER ANY
        FI
     OD
YDOB

PROPERTY LIFO,
         Blocking     %% The method get blocks if
                      %% the stack is empty.
END Blocking_Stack
```

Here we have an example of a class with an explicit body (describing the local process of each instance). In this case it only determines in which order the messages are answered. When the stack is empty, it accepts only put messages. All get messages are delayed until the stack is filled. The behaviour of the body is indicated by the property `Blocking`. Notice that the class `Blocking_Stack` implements the type `Int_Stack`. The class `AIS` could be decorated with the property `Return_NIL_if_empty`, which informs the user that the stack answers promptly, but that `NIL` is returned when the stack is empty. This would ensure that the types `Blocking_Stack` and `AIS` are incomparable, but both are subtypes of `Int_Stack`.

The body is an essential feature determining the behaviour of objects. Therefore, in general, the type of an object depends on its body. But since a formal description of this dependence would be intractable for a compiler, in POOL-I the behaviour of the body is indicated by property identifiers and the programmer has the responsibility to implement the behaviour according to the properties. This is completely analogous to those aspects in the behaviour of methods than are not comprised in their signatures.

We can define the subtyping relationship rigorously as follows:

Definition 1: Subtyping
The type τ is a subtype of σ if

1. The object properties of σ are among those of τ.
2. For each method m_σ of σ there is a corresponding method m_τ of τ, such that
   - m_σ and m_τ have the same name.
   - m_σ and m_τ have the same number of arguments.
   - The ith parameter type of m_σ is a subtype of the ith parameter type of m_τ (the contravariant parameter type rule).
   - Either both m_σ and m_τ have a result type or neither has one.
   - If there is a result type then the result type of m_τ is a subtype of the result type of m_σ (the covariant result type rule).

This is a recursive definition of the subtyping relation, which may have many solutions. We require the *greatest* solution. Informally speaking, this means that τ is a subtype of σ if there is no reason why it should not be a subtype.

The use of the contravariant rule for the parameter types and the covariant rule for the result type can be understood from the requirement that any object of type τ should fulfil the expectations pertaining to the type σ. In particular, if a caller expects to invoke the method m_σ, the method m_τ should be able to do the job. Therefore m_τ must accept at least all the arguments that m_σ accepts, and the result of m_τ must be acceptable in any context where an object of the result type of m_σ is expected. In fact, this is the only sound rule for the subtyping relationship [9, 11].

3 Genericity

In section 2 we introduced stacks of integers. Of course, stacks are not only useful for storing integers. There are many applications where we want to have stacks of other types. Generic types are a means to model types that are dependent on other types.

Example 7: The generic type Stack.

```
TYPE Stack (C)
PROPERTY LIFO %% Last in, first out
METHOD get () : C
METHOD put (C) : Stack (C)
END Stack
```

The type Stack has a type parameter C, which denotes the type of the elements that are stored in the stack. For instance, the type Stack(Person) is the type of stacks that stores persons. There are no objects of type Stack, but there are objects of type Stack(Person). If s is a (nonempty) object of type Stack(Person), then s!get() is an object of type Person.

Notice that the type Int_Stack is equivalent to the type Stack(Int), i.e., each object of type Int_Stack is also an object of type Stack(Int) and vice versa. Therefore the class AIS is an implementation of Stack(Int). It is also possible to write a generic class AS that implements the generic type Stack, which means that for each type C the class AS(C) implements the type Stack(C).

What happens when we substitute two parameters that are subtypes of each other? We cannot conclude that Stack(Person) is a subtype of Stack(Employee), because the method put requires an argument of type Person in the first case and an argument of type Employee in the second case. Should it be the other way around? No, because this is forbidden by the result type of the method get. Therefore in general we do not have a subtyping relationship between different instantiations of the same generic class. However, in certain cases such a subtyping relationship exists; it may be either covariant or contravariant in the type parameter. This can be made explicit to the reader of a program by writing a keyword COVAR or CONTRA with the type parameter, in which case the compiler will check whether this is correct.

Example 8: The type Dispenser.

```
TYPE Dispenser (COVAR C)
METHOD get () : C
END Dispenser
```

The keyword COVAR before the parameter indicates that the type Dispenser(C) is covariant in the argument C. This is correct since the argument C does not occur at a parameter place of any method. Therefore, e.g., Dispenser(Employee) is a subtype of Dispenser(Person).

The type Dispenser(C) may not seem very useful, but note that it is a supertype of the type Stack(C). It may be the case that some object a has a reference to an object s of type Stack(C), but delivers it to another object b as being of type Dispenser(C). Then b can take elements from the stack s but it cannot fill s itself, because it never knows that s is really of type Stack(C). This is a simple, but powerful way of working with *capabilities* (see, e.g., [14]).

The above generic types are not allowed to use any feature of their parameters, because nothing is known of them. In many applications, however, it is useful to have more information about the parameter types, for instance, the presence of certain methods or properties. To achieve this, the parameters can be *bounded* from above by a given type.

Example 9: The types Ordered and Sorter.

```
TYPE Ordered
METHOD less (Ordered) : Bool
PROPERTY Static          %% result independent of
                         %% moment of calling.
         Antisymmetric   %% a < b -> ~(b < a)
         Transitive      %% a < b & b < c -> a < c
         Linear          %% a < b | b < a | a == b
END Ordered


TYPE Sorter (X < Ordered)
METHOD sort (l: Array (X)) : Array (X)
         %% Returns l, sorted according to less.
END Sorter
```

The type Ordered has a method less, with properties to denote that it is an ordering. The type Sorter accepts as parameters only subtypes of Ordered. Therefore in the implementation it may use the fact that for objects of type X the method less is available, which has the properties Static, Antisymmetric, Transitive and Linear. For instance the types Int and Float have such a method less. Therefore we can create objects of type Sorter(Int) to sort integers and Sorter(Float) to sort floating point numbers.

4 Dynamic type manipulation

Not only types and classes can be generic, but methods may also have type parameters. The other parameter types and the result type may be dependent on a type parameter. As with generic classes the type parameters of the method can be bounded from above by a given type.

Example 10: The type Universal_Sorter.

```
TYPE Universal_Sorter
METHOD sort (X < Ordered, Array (X)) : Array (X)
END Universal_Sorter
```

In contrast to the type `Sorter`, objects of the type `Universal_Sorter` can sort all kinds of arrays with ordered elements according to their ordering method `less`. In addition, in the declaration of variables of the type `Universal_Sorter` the programmer does not have to decide for which types the sorter should act. These types can be computed during program execution. Suppose we have an object `s` of type `Universal_Sorter` and two arrays, the one, `ia`, containing integers and the other, `fa`, containing floating point numbers. Then we can perform `s!sort(Int,ia)` to sort the first array and `s!sort(Float,fa)` to sort the second array.

It is also possible to use a type as a new-parameter. In this case the type is known to the object during its whole life time, and the types of, e.g., its instance variables can depend on this new-parameter.

In order to make use of the different substitutions for the type parameters, the language allows some form of dynamic type analysis. This can test whether a type σ is a subtype of a type τ, where both σ and τ may depend on some type parameters. If the test succeeds, any object of type σ can be considered to be an element of τ.

Example 11: The type case statement.

```
TYPE Has_Int
METHOD get_Int   () : Int
END Has_Int

TYPE Has_Float
METHOD get_Float () : Float
END Has_Float
...
METHOD sum (T : TYPE, a : Array(T)) : Int
TEMP isum : Int := 0,    fsum : Float := 0.0
BEGIN
   CASE T
   OF Has_Int   %% T is a subtype of Has_Int,
                %% so it has a method get_Int
   THEN FOR i FROM a@lb TO a@ub
        DO isum := isum + a[i]!get_Int() OD;
        RESULT isum
   OR Has_Float %% T is a subtype of Has_Float,
                %% so it has a method get_Float
   THEN FOR i FROM a@lb TO a@ub
        DO fsum := fsum + a[i]!get_Float() OD;
        RESULT fsum@Int   %% round only here
   ELSE RESULT NIL
   ESAC
END sum
```

Note the difference between this scheme and another, more common scheme where each object carries its own type, which can be determined dynamically. In the above example, only one type comparison is necessary, and then the corresponding operations can be applied to a large number of objects. If every object carries its own type, then the type must be checked before every individual operation. In addition, note that the POOL-I scheme does not compromise the safety of the capability mechanism (see example 8 in section 3): a type can only be analysed if it is explicitly given, not if only an object is given.

5 Inheritance

Inheritance is a means for code sharing between classes. If a class C inherits from another class C', it gets all the methods, properties, instance variables and new-parameters from C'. Moreover, the initialization expressions of the variables are inherited as well. Only the body, the code that an object executes in parallel with the other objects, is not inherited. Note that, in contrast with most other strongly typed object-oriented languages, the mere fact that one class inherits from another one does not imply anything about a subtyping relationship between the corresponding types.

Example 12: The class `Linkable` (inspired by [12]).

```
CLASS Linkable(T)

NEWPAR (initial: T)      %% The initial contents
                         %% of the linkable

VAR value GETTABLE PUTTABLE := initial
                         %% the contents of the link
    right GETTABLE PUTTABLE : MYTYPE
                         %% the neighbour of the link
END Linkable
```

The class `Linkable(T)` can be used to build linked lists containing elements of type `T`. The `GETTABLE` and `PUTTABLE` attributes provide the type `Linkable(T)` with the following methods:

```
METHOD get_value() : T
METHOD put_value(T) : MYTYPE
METHOD get_right() : MYTYPE
METHOD put_right(MYTYPE) : MYTYPE
```

In the definition of the class `Linkable(T)` we see the occurrence of the type `MYTYPE` as the type of the variable `right`. This type is just another notation for the type that corresponds to the current class definition, viz. `Linkable(T)`. This notation is used with regard to

future inheritance. The class Bi_Linkable(T) below inherits from Linkable(T). Therefore, it gets the keyword MYTYPE at the same places, but now this keyword denotes the type Bi_Linkable(T).

Example 13: The class Bi_Linkable.

```
CLASS Bi_Linkable(T)
            %% not a subtype of Linkable(T)

PROPERTY Left_right_id
            %% left  ~== NIL -> left@right = SELF
         Right_left_id
            %% right ~== NIL -> right@left = SELF

INHERIT Linkable(T)

REDEFINE put_right OLD put_right_dangling
            %% put_right is not inherited, but
            %% newly defined. The old version is
            %% available as put_right_dangling.

VAR left GETTABLE : MYTYPE
            %% left neighbour of the link

METHOD put_left (l: MYTYPE) : MYTYPE
           %% put l to the right of the answering
           %% object.
BEGIN left := l;
      IF l ~== NIL
      THEN l ! put_right_dangling (SELF)
      FI;
      RESULT SELF
END put_left

METHOD put_right (r: MYTYPE) : MYTYPE
...
METHOD put_left_dangling (l: MYTYPE) : MYTYPE
...

END Bi_Linkable
```

The class Bi_Linkable(T) inherits everything except the (default) body from the class Linkable(T). The method put_right, derived from the attribute PUTTABLE in Linkable(T), is redefined, but the old method is available under the name put_right_dangling. This is necessary, because we want to have the old version available, e.g., in the definition of put_left. In addition a new variable left and some methods, including the redefined put_right are added to the class Bi_Linkable(T).

The type Bi_Linkable(T) is not a subtype of Linkable(T), nor the other way around. This is due to the parameter type of the method put_right, which is MYTYPE in both cases. This type behaves covariantly during inheritance, whereas we need a contravariant behaviour for the subtyping relationship. In fact, it is a typical phenomenon that with inheritance parameter and result types behave covariantly, whereas for subtyping a contravariant behaviour of the parameter types is necessary. This is an important source of trouble for languages where inheritance and subtyping coincide (see, for example, [10]).

In POOL-I multiple inheritance is also supported, i.e., a class can inherit from more than one other class. The conflict that arises if several of the inherited features happen to have the same name can and must be resolved explicitly by renaming (all but one of) them. Since inheritance is not coupled with subtyping, renaming methods does not lead to problems or to peculiar behaviour (as in [12]). If several of the inherited methods must be combined into a new one, we have the experience that it is simpler and more reliable if the programmer does this explicitly than if this is left to a complicated automatic mechanism, which is hard to understand (as in [7]).

There is no natural and practically useful automatic way to make an old body work well with new methods, or to combine several bodies into a new one. Therefore in POOL-I bodies are not inherited and every class must be given a body of its own. Since most classes only have the default body anyway, this does not lead to much extra work. Moreover, in those cases where an explicit body is necessary, it is well worth the trouble of paying extra attention to it. Thus the interaction between inheritance and parallelism is limited to the obligation for the programmer to provide a body. Because subtyping and inheritance are not combined the new body does not need to be compatible with the bodies of the inherited classes.

6 Conclusions

We have shown how subtyping and inheritance are included in the language POOL. The notions of subtyping and inheritance are treated separately because they deal with different views of objects. Types and subtyping look at an object from the outside, where the only relevant question is in what way the object can be used. Classes and inheritance deal with the internal structure of the object, where it is only important to realize the desired external behaviour by combining the right instance variables, new-parameters, methods, and body. It has long been recognized that the distinction between the external interface of an object and its internal realization is one of the most important aspects of object-oriented programming, but the awareness that this distinction logically implies an analogous distinction between subtyping and inheritance is much less widespread in the object-oriented community [2, 11].

Decoupling inheritance and subtyping gives considerably more freedom to the programmer. For example, it becomes possible to implement a stack by inheriting from the class `Array` without implying that a stack is a special kind of array so that all operations applicable to arrays are also applicable to stacks. It is also possible to group together a heterogeneous collection of objects which have only a part of their behaviour in common (e.g., having a method `print`), without the necessity that all of them have been constructed by inheriting from a fixed single class.

Furthermore such a decoupling makes it possible to solve several problems in the design of a strongly typed object-oriented language. A well-known problem is the occurrence of name clashes in multiple inheritance. Several complicated mechanisms have been proposed to resolve these clashes, but the simplest and most reliable mechanism, resolving the clashes by explicit renaming, leads to a conflict in subtyping (or to strange behaviour, as in [12]). If inheritance and subtyping are separated, this conflict disappears. Another problem is that inheritance tends to lead to a covariant subtyping relationship between parameter types of methods, whereas the only sound subtyping rule requires a contravariant behaviour. Coupling inheritance with subtyping can only lead to either an unsafe type system [10] or to unconvenient restrictions in inheritance, as in Trellis/Owl [13].

Note that these issues have nothing to do with parallelism. The above considerations are equally valid in a purely sequential object-oriented language. In fact, the POOL-I mechanisms for subtyping and inheritance interact only very weakly with its concepts for parallelism. The impossibility of a natural way of inheriting bodies is not particular to parallelism, but it is related to the difficulty of finding adequate automatic mechanisms for method combination in multiple inheritance.

The basic mechanisms for subtyping and inheritance in POOL-I are very simple, but in combination with the facilities for genericity and dynamic type manipulation they result in a very powerful language, with a flexible but absolutely safe type system. We are presently implementing POOL-I. First we make a sequential implementation, but we are planning to make a parallel implementation as well. Moreover, we intend to develop a formal semantics for POOL-I, based on the the earlier semantics defined for POOL [5], but giving a better insight in the fundamental properties of objects.

References

[1] P.H.M. AMERICA, *POOL-T: A parallel object-oriented language.* In A. Yonezawa, M. Tokoro (eds.): Object-Oriented Concurrent Programming, MIT Press, 199–220 (1987).

[2] P.H.M. AMERICA, *Inheritance and subtyping in a parallel object-oriented language.* In Proc. ECOOP'87, Paris, Springer LNCS 276, 234–242 (1987).

[3] P.H.M. AMERICA, *A Behavioural approach to subtyping in object-oriented programming languages.* Workshop on Inheritance Hierarchies in Knowledge Representation and Programming Languages, Viareggio, Italy, February 6–8, 1989. Also in Philips Journal of Research 44(2/3):365–383, (1989).

[4] P.H.M. AMERICA, *Issues in the design of a parallel object-oriented language.* Formal Aspects of Computing 1(4):366–411 (1989).

[5] P.H.M. AMERICA, J.W. DE BAKKER, J.N. KOK, J.J.M.M. RUTTEN, *Denotational semantics of a parallel object-oriented language.* Information and Computation 83(2):152–205 (1989).

[6] J.K. ANNOT AND P.A.M. DEN HAAN, *POOL and DOOM: The object-oriented approach.* Chapter 3 in P. Treleaven (ed.), Parallel Computers: Object-Oriented, Functional and Logic, Wiley (1989).

[7] D.G. BOBROW, L.G. DEMICHIEL, R.R. GABRIEL, S. KEENE, G. KICZALES, D.A. MOON, *Common Lisp Object System specification.* Doc. 88-003, X3J13 Standards Committee (ANSI Common Lisp) (1988).

[8] W.J.H.J. BRONNENBERG, A.J. NIJMAN, E.A.M. ODIJK, R.A.H. VAN TWIST, *DOOM: A decentralized object-oriented machine.* IEEE-MICRO 7(5):52–69 (1987).

[9] L. CARDELLI, *A semantics of multiple inheritance.* Information and Computation 76:138–164 (1988).

[10] W. COOK, *A proposal for making Eiffel type-safe.* Proc. ECOOP '89, Nottingham, England, July 10–14, 1989, 57–70, Cambridge University Press.

[11] W. COOK, W. HILL, P. CANNING, *Inheritance is not subtyping.* Proc. POPL (1990).

[12] B. MEYER, *Object-Oriented Software Construction.* Prentice-Hall (1988).

[13] C. SCHAFFERT, T. COOPER, B. BULLIS, M. KILIAN, C. WILPOLT, *An introduction to Trellis/Owl,* Proc. OOPSLA '86, Portland, Oregon, September 1986, pp. 9–16.

[14] W.A. WULF, R. LEVIN, S.P. HARBISON, *HYDRA/C.mmp: An Experimental Computer System.* McGraw-Hill (1981).