Designing an Object-Oriented Programming Language with Behavioural Subtyping

Pierre America
Philips Research Laboratories
P.O. Box 80.000
5600 JA Eindhoven
The Netherlands

Abstract

This paper describes the design of the parallel object-oriented programming language POOL-I. We concentrate on the type system of the language and specifically on the aspects of subtyping and genericity. POOL-I is the first language we know of that includes subtyping and inheritance as completely separate language mechanisms. By decoupling these two, which have been strongly tied together in other statically typed object-oriented languages with inheritance, a much cleaner language design can be obtained and a much more flexible use of both mechanisms can be made in actual programs.

In POOL-I subtyping is based only on the externally observable behaviour of objects. This includes not only their signature (the names of available methods and their parameter and result types) but also more detailed information about this behaviour. We also present a preliminary formalism in which these properties can be specified. Finally we introduce the more advanced features of the POOL-I type system, such as bounded genericity and dynamic type manipulation.

1 Introduction

In many object-oriented programming languages the concept of *inheritance* is present, which provides a mechanism for sharing code among several classes of objects. Many people even regard inheritance as the hallmark of object-orientedness in programming languages [Car88]. We do not agree with this view, and argue that the essence of object-oriented programming is the encapsulation of data and operations in objects and the protection of individual objects against each other (see Section 2). Nevertheless, inheritance is a very important concept and an extremely useful mechanism in structuring large systems.

Unfortunately, there are several aspects of inheritance in programming languages that are not yet understood in sufficient detail. Formal, mathematical models are clearly needed. Several such models have been proposed [Car88, Mit88], but the objects that these models deal with are very simple in nature: In essence they are mathematical entities that do not change their states during the execution of a program. Furthermore, they are completely transparent, in the sense that their internal structure is visible from outside (at least the mathematical models do not deal with any difference between the internal representation and the external view of such an object).

It is clear that the objects that occur in concrete programs, written in concrete object-oriented programming languages, have more complicated properties. On the one hand, these objects *can* change their states, while maintaining their identity. Therefore it is essential to be able to deal with dynamically evolving structures of references ('pointers') between objects. On the other hand,

The work described in this paper was done in the context of ESPRIT Basic Research Action 3020, *Integration*.

object-oriented programming can be seen as a refinement of abstract data structure techniques. The implementor of a class of objects uses a concrete internal representation to provide some more abstract service to its users. In order to do justice to this principle, it is necessary to distinguish between the internal structure of an object and the functionality it provides to the outside world. For a theory that must deal with the full generality of objects as they occur in object-oriented programming languages, more sophisticated techniques are necessary than the ones developed so far for simple kinds of objects.

In this paper we sketch a direction along which such a theory can be developed and we also indicate how this may affect the design of a concrete programming language. The paper is structured as follows: Section 2 gives a very brief introduction to object-oriented programming, defining some terminology. In Section 3 we deal with the important concepts of inheritance and subtyping. We argue that it is very useful to distinguish between two aspects of inheritance: On the one hand we have code sharing, which is involved with the internal structure of the objects, and on the other hand we have functional specialization, which has to do with the objects' behaviour insofar as it is visible from the outside. For the first aspect we shall continue to use the term 'inheritance', whereas we denote the second aspect by 'subtyping'.

In Section 4 we see how subtyping, interpreted along these lines, can be built into a concrete object-oriented programming language by describing the language POOL-I. This is the youngest member of the POOL family of languages, which have been designed to support the development of large, complicated programs for highly parallel machines with or without shared memory. In this paper we will not deal extensively with the aspects parallelism in POOL-I, but rather concentrate on the type system. We illustrate the difference between types and classes and we explain that subtyping will depend not only on the signatures of the different types (the names of the methods and their parameter and result types) but also on the presence of property identifiers, which are intended to give more detailed information on the behaviour of the objects in the type. In any case, the subtyping relationship does *not* depend on the internal structure of the object.

In Section 5 we sketch a formalism that could be used for specifying the external behaviour of objects. This formalism is not yet included in POOL-I, because there is still a lot of work to be done to develop into a generally useful and flexible way of writing specifications. Here it serves as an illustration of the kind of object properties we are interested in and of the requirements we would like to impose on a future, better formalism.

Then, in Section 6, we see how a very flexible, but completely safe type system can be built on the basic principles of POOL-I by including mechanisms like bounded and unbounded genericity and dynamic type manipulation.

2 Objects and classes

In this section we provide a short introduction to the basic elements of object-oriented programming and to POOL in particular. For a more extensive treatment of object-oriented programming, see [Mey88] and [SB86]. For more details on POOL, see [Ame87b] or [Ame89].

In object-oriented programming, we consider a system as a collection of objects. An *object* is an integrated unit of data and procedures that can act on these data (see Figure 1). Following the terminology of Smalltalk-80 [GR83], we use the term *methods* for these procedures. The data of an object are stored in *variables*. Such a variable can contain an element of some basic data type of the language, but it can also contain a reference to another object (in a *pure* object-oriented language, such as Smalltalk-80 and POOL, *all* the data are represented by objects, so that only the second case remains).

In general, objects are dynamic entities: They can be created dynamically and the internal state (comprising the values of the variables) of each object can change during its lifetime. A specific contribution of POOL is that each object also has a *body*, a local process that starts as soon as the object is created and runs in parallel with all the other objects in the system. This is what makes

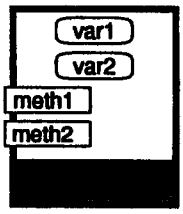


Figure 1: An object.

POOL a *parallel* object oriented language. In this paper, however, we shall not pay much attention to things that are specific to parallelism.

One of the most important principles of object-oriented programming is that the variables of one object cannot be accessed directly by other objects. The only way in which objects can interact is by sending *messages* (see Figure 2). A message is a request for the receiver to execute one of its methods, and such a method *can* access the variables of the object it belongs to. Together, the methods that an object provides constitute a clearly defined interface to the outside world. The fact that every access to an object takes place through this method interface gives rise to a powerful protection mechanism, which protects the data of each object against uncontrolled access from other objects. This mechanism also provides a separation between the implementation of an object (its set of variables and the code of the methods and body) and the behaviour that can be observed from outside.

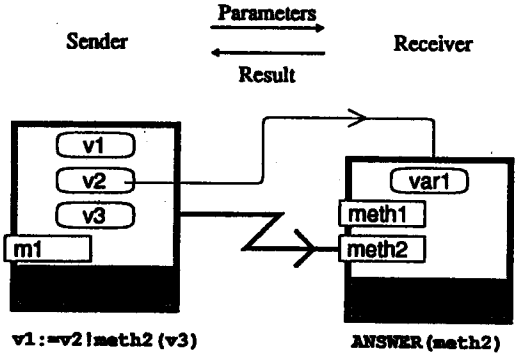


Figure 2: Sending a message.

The author considers this principle of protection of objects against each other as the basic and essential characteristic of object-oriented programming. It is a refinement of the technique of abstract data types, because it does not only protect one type of objects against all the other types, but one object against all the other ones. As a programmer we can consider ourselves at any moment to be sitting in exactly one object and looking at all the other objects from outside.

In order to describe all the objects in a system it is useful to group them into *classes*. All the objects in one class, the *instances* of the class, have the same methods and body and all of them have analogous sets of variables (each object has its own variables, of course, but the names and types of the variables are the same among all the instances of a class). A program in an object-oriented language consists mainly of class definitions, where each class definition contains a number

variable and method declarations. In this way a class definition provides exactly the information at is needed to create new objects. Examples of class definitions will be given below.

The creation of a new object of a certain class is not the task of other objects of the same class, it of the class itself. For this purpose, POOL provides *routines*, a kind of procedures different from ethods. A routine does not sit inside an object (like a method does) but it can be called by any ject that knows it (and even by more than one object concurrently). In fact, a routine is itself an ject. Every class is automatically provided with a routine called **new**, which creates and initializes new object of that class. The parameters of that routine, called the **new**-parameters, are available the object just like instance variables, with the exception that the values of the **new**-parameters annot be changed after the object has been created.

A typical class definition in POOL-I might look like this:

```
LASS My_Employee
EWPAR (first_name, last_name, address GETTABLE: String, init_sal: Int)
AR salary GETTABLE PUTTABLE: Int := init_sal

ETHOD get_name () : String
EGIN RESULT first_name + " " + last_name
ND get_name

ND My_Employee
```

Each object of the class `My_Employee` has an instance variable `salary` and four **new**-parameters, alled `first_name`, `last_name`, `address`, and `init_sal`. The method `get_name` is explicitly defined. t does not take any parameters and it returns the concatenation of first and last name as its result. In ddition, there are a few other methods, which are generated automatically because of the `GETTABLE` nd `PUTTABLE` attributes (a form of syntactic sugar, which provides an abbreviation for some very seful methods). For example, the `GETTABLE` attribute with the **new**-parameter `address` generates he following method:

```
ETHOD get_address () : String
EGIN RESULT address
ND get_address
```

nd the `PUTTABLE` attribute with the instance variable `salary` gives rise to this method:

```
ETHOD put_salary (new_salary : Int) : My_Employee
EGIN salary := new_salary;
     RESULT SELF
ND put_salary
```

Returning `SELF` (an expression that results in a reference to the object that executes it) is just a convention in the case that the method does not have anything more meaningful to return.

Now we can create an instance of the class `My_Employee` by calling the routine **new** associated with this class:

```
e := My_Employee.new("Peter", "Jones", "Oak Lane 11", 500);
```

We can ask for its name or change its salary by sending messages:

```
name := e ! get_name();
e ! put_salary (600);
```

3 Inheritance and subtyping

The basic idea of inheritance, as it appeared in the first object-oriented languages like Simula [DN66] and Smalltalk-80 [GR83], is that in defining a new class it is often very convenient to start with all the variables and methods of an existing class and to add some more in order to get the desired new class. The new class is said to *inherit* the variables and the methods of the old one. Of course this trick can be repeated several times, resulting in a complete inheritance *hierarchy*. We can even allow a class to inherit from more than one existing class, a principle known as *multiple inheritance*. By sharing code among classes in this way, the total amount of code in a system can sometimes be reduced drastically.

This inheritance relationship between classes also suggests another relationship: If a class B inherits from a class A, each instance of class B will have at least all the variables and methods that instances of class A have. It seems that whenever we require an object of class A, an instance of class B would do equally well. Therefore we are tempted to regard instances of class B as *specialized versions* of the ones of class A and to call B a *subclass* of A. It seems that the inheritance hierarchy described above, which is based on the sharing of code describing the internal structure of the objects, coincides completely with another hierarchy, which involves the use of the objects and therefore their externally observable behaviour.

This view has prevailed for a long time in the object-oriented community. However, it is becoming clearer recently that identifying these two hierarchies leads to several problems and that it is useful to separate them (see also [Ame87a, CHC89, Sny86]). The reason for this is that it is not always the case that code sharing automatically leads to behavioural specialization, nor that it is the only way leading to specialization. For example, if we add a new method to a carefully designed set of variables and methods, it is quite possible that the new method invalidates an invariant on which the functioning of the old methods was based. In this way the old methods may start to behave very differently, so that we have code sharing, but no specialization in behaviour. On the other hand, it is well known that is often possible to obtain the same functionality by very different representations. For example, complex numbers can be implemented using Cartesian coordinates or using polar coordinates, and a stack can be implemented using an array as well as using a linked list. In these cases we have not only specialization, but exact duplication of behaviour, while the internal representation, and therefore the code, is completely different.

It turns out that several problems with inheritance, especially multiple inheritance, can be solved if we stop identifying the code sharing mechanism with the specialization hierarchy. To distinguish the two concepts, we shall use in this paper the term 'inheritance' for the mechanism of code sharing as it is present in many object-oriented programming languages, and we shall introduce the term 'subtyping' to denote behavioural specialization. Inheritance, used in this sense, is now a concept that does not need much further explanation. It is already present in some object-oriented programming languages with strong typing (e.g., Trellis/Owl [SCB*86] and Eiffel [Mey88]) as well as in some without strong typing (e.g., Smalltalk-80 [GR83]). It just consists of taking over variables and methods from an existing class in defining a new one, as described above. Let us repeat that by a class we mean a collection of objects that have exactly the same internal structure (variables and methods). In this view, by applying inheritance we get a new class, distinct and disjoint from the class from which it inherits. Therefore we shall not use the term 'subclass' any more.

The concept of subtyping is more difficult to explain. Weakly typed object-oriented languages do not have an explicit notion of types and subtyping at all, while the existing strongly typed languages identify subtyping with inheritance. One of the goals of this paper is to give a definition of types and subtyping that is independent of the internal representation of the objects and therefore independent of inheritance in the sense of code sharing.

As a starting point, note that the types in a program do not only provide some information to the compiler, which can detect certain errors by type checking, but it also constitutes an important part of the documentation for the human reader. Now if we consider the type of a variable or

expression, and therefore the type of the object it denotes, we are not interested in how this object is represented internally, but rather in the possible ways that this object can be *used*. After all, the type of a variable or expression determines a subset of the set of all objects, in such a way that the language mechanisms guarantee that the variable or expression will only refer to objects in this subset. Now the only thing that can be done to the object to which a variable or expression refers (apart from storing it into another variable or parameter) is sending it a message. So the only thing relevant for typing purposes is how an object reacts to messages. The internal details of the object (e.g., the names of its instance variables) are certainly not important. Furthermore, types are static entities that can be manipulated by a compiler, so they should reflect object properties that do not change over time.

Therefore we define a *type* as a collection of objects that have some intrinsic property in common which is externally observable. By 'intrinsic' we mean that the property cannot change during the lifetime of the object, and by 'externally observable' we mean that the property can in principle be observed by sending messages to the object (which is the only way of interacting with it). In this way, we are really talking about the ways in which the object can be used. Note that, because of the word 'intrinsic' above, if one object belongs to a certain type then all the instances of its class belong to that type: all these instances have the same properties at the moment they are created, and the intrinsic properties cannot change during their lifetime.

As an example, let us consider the type `Int_Stack`, which comprises all the objects that have the following behaviour:

> The object will accept `put` and `get` messages, but it will accept a `get` message only if the number of `put` messages already received exceeds the number of `get` messages. A `put` message contains one integer as an argument and returns `SELF` as its result. A `get` message contains no arguments and it returns as its result the integer that was the argument of the last `put` message that has preceded an equal number of `put` and `get` messages.

Note that this property can indeed be observed just by sending messages to the object and *without reference to its internal structure*. By contrast, the property that an object has a variable called `x` does not constitute a type, because it is not observable from the outside of the object.

In other words, a type is essentially the same as a *specification* of the behaviour of its elements. Note that this specification comprises the names of the methods that the objects should have and the types of the parameters and results of these methods. This is often called the *signature* of the type. But our specification give more information about the behaviour of the object under consideration, which is not contained in the signature: it states under which conditions a certain message may be sent to the object (possibly constraining the values of the parameters) and what are the possible values of the result.

Now from this definition of a type it is clear how the notion of subtyping should be defined: We say that a type σ is a *subtype* of a type τ (notation $\sigma < \tau$) if it is always the case that any object belonging to σ will also belong to τ. In terms of specifications: σ is a subtype of τ if for any object the fact that it satisfies σ's specification implies that it satisfies the specification of τ.

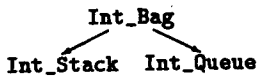
For example, consider the type `Int_Bag` of objects with the following behaviour:

> The object will accept `put` and `get` messages, but it will accept a `get` message only if the number of `put` messages already received exceeds the number of `get` messages. A `put` message contains one integer as an argument and returns `SELF` as its result. A `get` message contains no arguments and it returns as its result some integer such that the number of previously accepted `put` messages having this integer as their argument exceeds the number of previously accepted `get` messages that returned this integer as their result.

The type `Int_Stack` is a subtype of the type `Int_Bag`, since every element of `Int_Stack` is also an element of `Int_Bag`. It must however be admitted that this does not follow in a very trivial way from

the above specifications. Therefore, in the next section we shall develop another way of formulating specifications.

Note, by the way, that `Int_Stack` is a strict subtype of `Int_Bag`, i.e., the subtyping relationship does not hold the other way around, but they nevertheless have the same signature. One could also imagine a type `Int_Queue`, again with the same signature and with the obvious intuitive meaning, which is a subtype of `Int_Bag` but incomparable to `Int_Stack`:



4 Types and subtyping in POOL-I

4.1 Types and classes

In this section we introduce the way in which types are defined in POOL-I and their relationship to classes and other types. We have already seen that a type is a collection of objects that have a certain aspect of the externally observable behaviour in common. One could even say that a type constitutes a specification of that behaviour. The externally observable behaviour is essentially the way in which an object reacts to messages. One of the most important aspects is, of course, whether the object can accept the message at all (i.e., whether it has a method of the appropriate name), and if so, how many parameters (and of which types) it requires and which type of result it returns. This kind of information, which should certainly be included in a type, is called a signature. It is possible for a type definition to consist of a signature only.

Example 1: The type `Person`.

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

The type `Person` denotes the collection of all objects that have, at least, the two methods `get_name` and `get_address`, both with no arguments, and with result type `String`. For reasons of convenience, this does not exclude objects that have other methods in addition to `get_name` and `get_address`.

Recall from Section 2 that a class is a collection of objects with exactly the same internal structure: variables, new-parameters, (code for) methods, and body. This structure is described in a class definition. Now we say that a class C *implements* a type σ if each object of the class C belongs to the type σ.

Example 2: The class `My_Person`.

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

By expanding the methods that are generated automatically by the `GETTABLE` and `PUTTABLE` attributes of the new-parameter `name` and the instance variable `address` (as explained in Section 2) we see that each object of the class `My_Person` has the following methods:

```
METHOD get_name () : String
METHOD get_address () : String
METHOD put_address (String) : My_Person
```

Therefore all instances of the class My_Person belong to the type Person or, in other words, the class My_Person implements the type Person. The fact that the objects of the class My_Person also have a method put_address does not matter for this fact. The fact that a class implements a type can optionally be expressed in the class heading as follows:

```
CLASS My_Person < Person
```

Let us consider another example of a type and a class that implements it.

Example 3: The type Employee and the class My_Employee.

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

CLASS My_Employee < Employee
NEWPAR (first_name, last_name, address GETTABLE: String, init_sal: Int)
VAR salary GETTABLE PUTTABLE: Int := init_sal

METHOD get_name () : String
BEGIN RESULT first_name + " " + last_name
END get_name

END My_Employee
```

Again, expanding the GETTABLE and PUTTABLE attributes as we did in Section 2, we see that every instance of the class My_Employee provides the following methods:

```
METHOD get_name () : String
METHOD get_address () : String
METHOD get_salary () : Int
METHOD put_salary (Int) : My_Employee
```

Therefore the class My_Employee indeed implements the type Employee.

Looking at the types Employee and Person, we see that every object that is an element of the type Employee will certainly have the methods get_name and get_address, taking no parameters and returning a result of type String, so it will also be an element of the type Person. Therefore, the type Employee is a subtype of the type Person. The fact that the type Employee has an additional method get_salary does not prevent this type inclusion; it only ensures that the subtyping relationship does not hold the other way around: Person is not a subtype of Employee.

The consequence of this is that every class that implements the type Employee also implements the type Person. Therefore all the instances of the class My_Employee are elements not only of the type Employee but also of the type Person. By contrast, the instances of the class My_Person are elements of the type Person but not of the type Employee.

Actually, we can also describe this in another way: With each class a type is associated having the same name as the class. The specification of this type consists of the headers of all the methods defined in the class (plus the corresponding properties, see Section 4.4). Therefore the definition of the class My_Person above implies the following type definition.

```
TYPE My_Person
METHOD get_name () : String
METHOD get_address () : String
METHOD put_address (String) : My_Person
END My_Person
```

It is clear that all the instances of the class `My_Person` are elements of the type `My_Person`. Furthermore the type `My_Person` is a subtype of the type `Person`. This is another way of explaining that every object of the class `My_Person` belongs to the type `Person`. In general, one can say that a class C implements a type τ precisely if the type σ associated with C's class definition is a subtype of τ.

4.2 Subtyping in context

The subtype relationship is important in two situations: assignments and parameter passing. An assignment is only allowed if the type of the expression at the right-hand side is a subtype of the type of the variable at the left-hand side. This will make sure that a variable of a type τ can only refer to objects that belong to the type τ.

Example 4: Assignments and subtypes. Suppose that we have the following variable declarations:

```
VAR p: Person
    e: Employee
```

Then the following assignments are allowed:

```
p := My_Person.new("John Smith", "Main Street 5");
e := My_Employee.new("Peter", "Jones", "Oak Lane 11", 500);
p := e;
p := My_Employee.new("Hank", "Baker", "Rock Drive 9", 600);
```

By contrast, the following assignments are illegal:

```
e := p;
e := My_Person.new("Adam Peters", "Hill Street 30");
```

Note that things could indeed go wrong if we would relax this rule. (If the last assignment were allowed, a subsequent statement of the form `e!get_salary()` would send a message to an object of class `My_Person`, which does not have an appropriate method.)

For parameter passing to methods or routines, it it required that the type of each argument expression is a subtype of the type of the corresponding parameter of the method or routine. Again, we illustrate this by an example.

Example 5: Routines and subtypes. Suppose that the following routines are defined:

```
ROUTINE print_person (p: Person)
BEGIN standard_out ! write_String (p!get_name());
      standard_out ! new_line ();
      standard_out ! write_String (p!get_address());
      standard_out ! new_line ();
END print_person

ROUTINE print_employee (e: Employee)
BEGIN standard_out ! write_String (e!get_name());
      standard_out ! write_String (" earns a salary of $");
      standard_out ! write_Int (e!get_salary(), 0);
      standard_out ! new_line ();
END print_person
```

Then the following calls are legal:

```
print_person (p);
print_person (e);
print_employee (e);
```

But the following call is not allowed:

```
print_employee (p);
```

4.3 Parameter and result types

The examples in Section 4.1 suggest that for a type σ to be a subtype of a type τ it is necessary that every method of τ occurs also in σ *in exactly the same form*. However, we can relax that a little. In fact, we can also allow the methods in σ to be in a certain sense more specific, in such a way that it is nevertheless guaranteed that any object of type σ will work correctly when it is used in a context where an object of type τ is required.

Let us first look at the result types: For a method m that occurs in the type τ we require that it also occurs in the type σ and that the result type ρ^σ of m as mentioned in σ is a subtype of its result type ρ^τ as mentioned in τ, or in a formula: $\rho^\sigma < \rho^\tau$. It is clear that this requirement will lead to a correct typing mechanism: In order for an object to belong to type τ, its method m should deliver a result of type ρ^τ, but if it is a member of type σ then its method m will deliver a result of type ρ^σ and if $\rho^\sigma < \rho^\tau$ then this result will also be a member of type ρ^τ. We call this typing rule *covariant*, since the inclusion of the result types of the methods points in the same direction as the inclusion of the parent types σ and τ.

Example 6: The types `Person_Dispenser` and `Employee_Dispenser`.

```
TYPE Person_Dispenser
METHOD get () : Person
END Person_Dispenser

TYPE Employee_Dispenser
METHOD get () : Employee
END Employee_Dispenser
```

Here we have that `Employee_Dispenser` is a subtype of `Person_Dispenser`, since the method `get` returns an object of type `Employee`, which is automatically an element of type `Person`, because `Employee < Person`. If we have a variable pd of type `Person_Dispenser` (in addition to our old variable p of type `Person`, then the assignment

```
p := pd ! get()
```

is always correct. If the variable pd refers to an object of type `Employee_Dispenser`, the result of the message will be an object of type `Employee`, which also belongs to the type `Person`, so that it can safely stored in the variable p.

An important special case of this rule occurs if a method returns `SELF`. In general, each type τ that has a method m returning `SELF` will list τ as a result type. Now if another type σ also has a method m returning `SELF`, so that its result type is σ, then this will be not prevent σ to be a subtype of τ, because of the covariant result type rule. (As we shall see in Section 4.5, a subtyping relationship will always hold unless there is an explicit reason why it should not hold.)

Example 7: The type `Moving_Person` and the class `My_Person`.

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

As explained in Example 2, the class `My_Person` (and therefore also the associated type `My_Person`) has the following methods:

```
METHOD get_name () : String
METHOD get_address () : String
METHOD put_address (String) : My_Person
```

Now because of the covariant result type rule, the type `My_Person` is a subtype of `Moving_Person`, which is another way of saying that the class `My_Person` implements the type `Moving_Person`. (Accidentally, in this case it is also true that `Moving_Person` is a subtype of the type `My_Person`.)

By the way, in POOL it is also possible for a method to have no result type at all, in which case it is used for *asynchronous* communication. We require that if the method m does not have a result type in τ, then it should not have a result type in σ either.

For the parameter types we have the following, *contravariant*, rule: The type π_i^σ of the ith parameter of the method m as mentioned in σ should be a supertype of its ith parameter type π_i^τ as mentioned in τ, or $\pi_i^\tau < \pi_i^\sigma$. (Note that the direction of the inclusion is reversed; this is why this rule is called contravariant.) Again, after a little thought it is clear that this rule is indeed safe: For an object to belong to the type τ it should be able to accept every element of π_i^τ as the ith parameter of its method m, but if it is a member of σ then it will be able to accept every element of π_i^σ, which comprises all elements of π_i^τ.

Example 8: The types `Person_Printer` and `Employee_Printer`.

```
TYPE Person_Printer
METHOD print (p: Person)
END Person_Printer

TYPE Employee_Printer
METHOD print (e: Employee)
END Employee_Printer
```

According to the above rule, we have that `Person_Printer` is a subtype of `Employee_Printer` (not the other way around). We can illustrate this by taking a variable pp of type `Person_Printer` and a variable ep of type `Employee_Printer`. Then the following statements are legal:

```
pp ! print (p);
ep ! print (e);
pp ! print (e);
```

However, the following expression is illegal:

```
ep !  print (p);
```

We can see more clearly that something goes wrong in the last statement by looking at the following class, which is clearly an implementation of the type `Employee_Printer`:

```
CLASS My_Employee_Printer
METHOD print (e: Employee)
BEGIN standard_out ! write_String (e!get_name());
      standard_out ! write_Int (e!get_salary(), 0);
      standard_out ! new_line ();
END print
END My_Employee_Printer
```

Here the method `print` sends a message `get_salary` to its argument `e`, which leads to an error if the argument is an object of type `Person` (e.g., an instance of the class `My_Person`) that does not have a method `get_salary`.

It must be said, however, that the above example does not give a very handy solution for printing objects. A much better design would be to let every object print itself, like in the following fragment:

Example 9: Printables.

```
TYPE Printable
METHOD print (iod: IO_Device): Printable
END Printable

ROUTINE print_array(a: Array(Printable), io: IO_Device)
BEGIN
      FOR i FROM a@lb TO a@ub
      DO a[i] ! print(io)
      OD
END print_array
```

The type `Printable` contains all objects that can print themselves on I/O devices. Subtypes of `Printable` may be `Int`, `Char` and `String`, for example. The types `Person` and `Employee` could also be expanded to include an appropriate method `print`. The routine `print_array` uses this fact to print an array of printables, which may contain instances of different classes that implement the type `Printable`. The routine itself does not need to know what are the details of the `print`-methods of the elements of the argument `a`, since during execution each element of the array simply executes its own `print` method (this is called *dynamic binding*).

4.4 Properties

Up to now we have distinguished types by their signature: Their methods, together with the argument and result types. There is more that can be used to distinguish types. For instance, not all methods may be answerable at any moment or the results of the methods may depend on the order in which they are executed. The designer of the type may want to give a more detailed specification of the behaviour of the object than is included in the signature. In Section 5 we shall explore a formalism to express these specifications. Unfortunately, that formalism has the full power of first-order logic and therefore it is inherently impossible for the compiler to check the subtyping relationship on the basis of specifications in that formalism. Therefore, in the programming language POOL-I, we use only *property identifiers* to denote these specifications. The compiler will not check whether the code of a class really satisfies such a specification; it will just base its decisions about subtyping relationships on the presence or absence of these property identifiers. As far as the programming language is concerned, it is entirely left to the responsibility of the programmer to associate a certain specification (formal or informal) with a property identifier and to verify that each class that mentions a property identifier indeed satisfies the associated specification. This decision in the language design enables us to develop the specification formalism further while the programming language is fixed. At a later stage, it may be possible to construct specialized tools to assist the programmer in administrating the verification conditions or even in proving them formally.

Example 10: 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 t : Int          := 0
    a : Array (Int) := Array (Int).new (1,20)

METHOD get () : Int
BEGIN IF t = 0
      THEN RESULT NIL
      ELSE RESULT a[t]; t := t - 1
      FI
END get

METHOD put (n : Int) : AIS
BEGIN IF a@ub <= t                  %% @ub selects upper bound
      THEN a ! take_size (1, t+20)  %% changes the array bounds
      FI;
      t := t + 1;
      a[t] := n;
      RESULT SELF
END put

METHOD size () : Int
BEGIN RESULT t
END size

PROPERTY LIFO
END AIS
```

Here the compiler (following the POOL-I language definition) will decide that the class AIS implements the type Int_Stack, because it has the required methods get and put and the property LIFO. As far as the language is concerned, there is no special meaning associated with the identifier LIFO, in particular it is not checked at all by the compiler that the item that has been entered last into an instance of AIS 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. The property identifiers are only regarded by the compiler when determining if one type is a subtype of another one.

Example 11: The type Int_Bag.

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

The type Int_Bag has the same signature as the type Int_Stack, but it does not require the LIFO property. Therefore the type Int_Stack is a subtype of the type Int_Bag (but not the other way around) and the class AIS also implements the type Int_Bag. The property LIFO is a means to distinguish the behaviour of stacks from the less specific behaviour of bags.

Example 12: 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 therefore they are 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 Int_Queue. However the following class *does* implement the type Int_Queue:

Example 13: The class AIQ (array integer queue).

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

METHOD get () : Int
BEGIN IF #a = 0                  %% #a: number of elements
      THEN RESULT NIL
      ELSE RESULT a!get_low()  %% take first element and increase lb
      FI
END get

METHOD put (n: Int) : AIQ
BEGIN a!put_high(n);             %% increase ub and place n at high end
      RESULT SELF
END put

PROPERTY FIFO
END AIQ
```

Inserting the property identifiers LIFO and FIFO at the appropriate places will enable the compiler to prevent that a programmer inadvertently uses a queue where a stack is needed, an error that could not be detected by only looking at the signatures.

Sometimes it is natural to associate a certain property with a single method, for example to indicate that a method is free of side-effects or that the operation it performs is commutative or associative. In this case we call it a *method property*. Properties that describe the interplay of two or more methods should pertain to the whole class; to distinguish them from method properties, they are called *object properties*.

For instance we could add a method peek to bags.

Example 14: The type Peek_Bag.

```
TYPE Peek_Bag
METHOD get () : Int
METHOD put (Int) : Peek_Bag
METHOD peek () : Int
PROPERTY Side_effect_free
%% Does not change the internal state.
%% The method peek returns an element that is contained
%% in the bag without removing it.
END Peek_Bag
```

The method property Side_effect_free denotes the behaviour of the method peek, which only delivers a result but does not change the contents of the bag.

4.5 A rigorous definition of subtyping in POOL-I

Now that we have seen all the aspects that play a role in determining whether one type is a subtype of another one, we are ready to give the official definition.

Definition
The type σ is a subtype of τ if

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

This is a recursive definition of the subtyping relation, which may have many solutions. We explicitly require the *greatest* solution, i.e., that solution that says $\sigma < \tau$ for the largest set of pairs $\langle \sigma, \tau \rangle$*. In practice, this amounts to saying that σ is a subtype of τ if there is no explicit reason why it should not be a subtype.

5 Specifying object behaviour

In this section we sketch a formalism that could be used to express the specifications that we would like to associate with property identifiers. Such a formalism should be able to specify the externally observable behaviour of objects. In principle, we could do this by reasoning about the sequence of messages that an object receives and sends, as we have done in the informal specifications in Section 3. However, this technique has two important disadvantages: On the one hand, it has a strong operational flavour, and on the other hand, it is not the most abstract specification (because it distinguishes, e.g., between an empty stack that has just been created and one from which all previously inserted elements have been removed). In addition, for most kinds of structures, reasoning about sequences of messages is not the most natural way to think about them: Intuitively, we think of a stack as an object storing certain data, and we have some abstract notion about the contents of such a stack. Therefore, it is relatively hard to understand specifications of the kind given above and to judge whether they really correspond to our (informally stated) requirements, as the reader may experience from the example specifications in Section 3. For more complex objects, a specification exclusively in terms of sequences of messages is clearly infeasible.

In order to get a specification technique that corresponds more closely to our intuition, we use a technique that is very common in dealing with abstract data types [Hoa72]: We model the abstract conceptual state of an object by an element of some mathematical domain. For example, in the specification of stacks we use as the mathematical domain the set Σ of all finite sequences of integers. More precisely, the internal state of an object of type `Stack` is represented in the specification by a finite sequence s of integers (where the first integer in the sequence is the one that was entered

*It can be shown quite easily that such a greatest solution exists and that it is in fact unique, if one observes that the above definition of the subtyping relation gives rise to a monotonic operator on the lattice of all binary relations on types, ordered by set inclusion. Such an operator then has a unique greatest fixed point, which is the subtyping relation that we want.

into the stack first). Now the methods put and get can be specified by pre- and postconditions as follows:

$$\{\text{true}\}\ \text{put}(n)\ \{s = s_0 * \langle n \rangle\},$$
$$\{s \neq \langle\rangle\}\ \text{get}()\ \{s_0 = s * \langle r \rangle\}.$$

Here s is the sequence representing the current state of the stack, s_0 in the postcondition stands for the value of s before the method execution, and r stands for the result of the get method. Furthermore, the operator $*$ denotes concatenation of sequences, $\langle\rangle$ is the empty sequence, and $\langle n \rangle$ is the sequence having n as its only element. The meaning of such a method specification is that whenever the precondition holds before the execution of the method, then the postcondition holds after the execution.

In general, a specification of a type σ consists of a domain Σ, representing the set of possible abstract states of objects of type σ, plus a set of method specifications of the form $\{P\}m(\bar{p})\{Q\}$, where the precondition $P = P(s, \bar{p})$ describes the state of affairs before the method execution and the postcondition $Q = Q(s, s_0, \bar{p}, r)$ describes the situation after its execution (s always stands for the current abstract state, s_0 for the abstract state before the method execution, $\bar{p}$ for the method parameters, and r for the result). The meaning of such a method specification is that each object of type σ should have a method with name m available such that if the method is executed in a state where the precondition P holds, then after the method execution Q holds.

Now the important question is under what conditions the objects of a given class C are members of a type σ, in which case we say that the class C implements the type σ. These conditions can be formulated as follows: We require a *representation function* $f : C \to \Sigma$, where C is the set of possible concrete states of objects of class C, i.e., the set of possible values of the variables $\bar{v}$ of such an object, and Σ is the set of abstract states associated with the type σ. The representation function f maps the values $\bar{v}$ of the variables of an object of class C to an element s of the mathematical domain Σ that is used in the specification of the type σ. We also need a *representation invariant* I, which is a logical formula involving the values of the variables of the class C. This invariant will describe the set of values of these variables that can actually occur (in general this is a proper subset of the set C). The representation function f should at least be defined for all concrete states for which the invariant I holds.

For the class C to implement the type σ the following conditions are required to hold:

1. The invariant I holds initially, i.e., just after the creation and initialization of each new object.

2. Every method m of the class C (the ones that are mentioned in σ's specification as well as the ones that are not mentioned) should satisfy

$$\{I\}\ m(\bar{p})\ \{I\}.$$

3. For every method specification $\{P\}m(\bar{p})\{Q\}$ occurring in the specification of σ, the class C should also have a method m with parameters $\bar{p}$ of the right number and types. For the moment, we interpret this by saying that the number of parameters and their types should be the same in the specification and in the class, but in Section 5.1 we shall see that we can be more liberal about this. Furthermore this method should satisfy

$$\{P \circ f \wedge I\}\ m(\bar{p})\ \{Q \circ f \wedge I\}.$$

Here $P \circ f$ stands for the formula P where every occurrence of the abstract state s is replaced by the function f applied to the variables and analogously with s_0: $P \circ f = P[f(\bar{v})/s, f(\bar{v_0})/s_0]$. Note that the symbol $\circ$ indeed denotes a kind of functional composition of the function $f : C \to \Sigma$ from concrete states to abstract states and the predicate P, which maps abstract states into truth values ('$\Sigma \to \{\text{t}, \text{f}\}$').

(Because of requirement 2 we could in principle omit the invariant I from the postcondition in the current requirement. However, it remains essential in the precondition.)

As an illustration, consider the class AIS from Example 10 in Section 4.4. It has two variables: a of type Array(Int) and t of type Int. We want to show that this class implements the type Int_Stack. Now we can define a representation function f as follows:

$$f(\mathtt{a},\mathtt{t}) = \langle \mathtt{a}[1],\ldots,\mathtt{a}[\mathtt{t}]\rangle. \tag{1}$$

For the representation invariant we can take

$$I: \quad \mathrm{lb}(\mathtt{a}) = 1 \wedge \mathrm{ub}(\mathtt{a}) \geq \mathtt{t} \geq 0.$$

This invariant I holds for a new object after the initialization of the variables, since $\mathtt{t} = 0$ and a has lower bound 1 and upper bound 20. For the method put of the class AIS we have to show that it satisfies

$$\{I\}\ \mathrm{put}(n)\ \{f(\mathtt{a},\mathtt{t}) = f(\mathtt{a}_0,\mathtt{t}_0) * \langle n\rangle \wedge I\}. \tag{2}$$

If we fill in the definition of f from (1) and simplify we get

$$\{I\}\ \mathrm{put}(n)\ \{\mathtt{t} = \mathtt{t}_0 + 1 \wedge \mathtt{a}[\mathtt{t}] = n \wedge \forall i(1 \leq i < \mathtt{t} \rightarrow \mathtt{a}[i] = \mathtt{a}_0[i]) \wedge I\}, \tag{3}$$

which can be verified from the program text.

The method get should satisfy

$$\{f(\mathtt{a},\mathtt{t}) \neq \langle\rangle \wedge I\}\ \mathtt{get}()\ \{f(\mathtt{a}_0,\mathtt{t}_0) = f(\mathtt{a},\mathtt{t}) * \langle r\rangle \wedge I\}. \tag{4}$$

This can be expanded into

$$\{t \neq 0 \wedge I\}\ \mathtt{get}()\ \{\mathtt{t}_0 = \mathtt{t} + 1 \wedge r = \mathtt{a}_0[\mathtt{t}_0] \wedge \forall i(1 \leq i \leq \mathtt{t} \rightarrow \mathtt{a}[i] = \mathtt{a}_0[i]) \wedge I\}. \tag{5}$$

Again this can be verified from the text of the program.

Furthermore for every method m (that is for put, get, *and* size) we have

$$\{I\}m(\bar{p})\{I\}$$

where $\bar{p}$ stands for the list of parameters of the method m. (For the method put this follows already from (3).) Since all these requirements are fulfilled, the class AIS implements the type Int_Stack.

At this point, it may not yet be clear that the properties that we specify in the above manner can indeed be observed just by sending messages to the objects. However, it should be clear that this notion of implementing a specification does not depend on the internal structure of the objects: If we have a totally different class, e.g., a stack implementation using a linked list, then we can nevertheless show that it implements the specification by choosing an appropriate representation function and invariant. We can illustrate this by giving a different class, IIS (for 'inefficient stack') that also implements the specification Int_Stack.

Example 15: The class IIS.

```
CLASS IIS
VAR t : Int := NIL
    r : IIS := NIL
    c : Int := 0

METHOD put (n : Int)
BEGIN
    IF r == NIL
    THEN r := IIS.new ()
    ELSIF c > 0
    THEN r ! put(t)
    FI;
    t := n; c := c + 1
END put
```

```
METHOD get (
BEGIN
   IF c > 0
   THEN RESU
        t :=
        c :=
   ELSE RESU
   FI
END get
END IIS
```

This imple
to store the re
invariant is

For the repres

Here the impo
to by the vari
(Note that we
also belong to
While it is
of the internal
they specify c
case, but it de
should not co
a given object
elements that
the type Int_

5.1 Spec

Now that we k
verify that a c
impose on th
τ. At first, it
in τ's specific
such that the
these circums
we send such
that initially
precondition
from σ will h
However,
ematical dom
type Int_Bag
nonnegative i
the number n
put and get

4.4. It has two variables:
class implements the type

(1)

les, since t = 0 and a has
3 we have to show that it

(2)

$a_0[i]) \wedge I\}$, (3)

$I\}$. (4)

$= a_0[i]) \wedge I\}$. (5)

ave

d put this follows already
its the type Int_Stack.

cify in the above manner
, it should be clear that
structure of the objects:
linked list, then we can
propriate representation
IS (for 'inefficient stack')

```
METHOD get () : Int
BEGIN
   IF c > 0
   THEN RESULT t;
        t := r ! get ();
        c := c - 1
   ELSE RESULT NIL
   FI
END get
END IIS
```

This implementation uses the variable t to store the top element of the stack and a variable r to store the rest of the elements. The variable c counts the number of elements. The representation invariant is

$$c \geq 0 \wedge (c > 0 \rightarrow \neg(r == \mathtt{NIL}))$$

For the representation function g we take the following definition:

$$g(t, r, c) = \begin{cases} \langle\rangle & \text{if } c = 0 \\ \langle t \rangle * r.s & \text{if } c > 0 \end{cases}$$

Here the important thing to notice is that we cannot access the internal details of the object referred to by the variable r. Therefore we refer to its abstract state, a sequence of integers, denoted by $r.s$. (Note that we could declare the variable r as having type Int_Stack, because all elements of IIS also belong to the type Int_Stack.)

While it is clear now that our specifications, when interpreted in the above way, are independent of the internal structure of the objects, it is a more difficult problem to decide whether the properties they specify can really be observed just by sending messages to such an object. This is not always the case, but it depends on the adequate choice of the mathematical domain. The elements of this domain should not contain too much information: It should be possible to determine the abstract state of a given object by sending a sequence of messages to it. Moreover, the domain should not contain elements that cannot actually occur as the abstract state of an object. In the above specification of the type Int_Stack, these conditions are clearly satisfied.

5.1 Specifications and the subtyping relationship

Now that we have seen how we can write specifications to define a type and how we could, in principle, verify that a class implements a type, we go on to answer the question which requirements we should impose on these specification of a type σ and a type τ in order to be sure that σ is a subtype of τ. At first, it seems sufficient to require that for every method specification $\{P\}m(\bar{p})\{Q\}$ occurring in τ's specification there should be a method specification $\{P'\}m(\bar{p})\{Q'\}$ in the specification of σ such that the latter implies the former, which can be expressed by $P \rightarrow P'$ and $Q' \rightarrow Q$. Under these circumstances we can indeed use any element of σ whenever an element of τ is expected: When we send such an object a message listing the method m, using it as an element of τ guarantees that initially the precondition P will hold. By the implication $P \rightarrow P'$ we can conclude that the precondition P' in σ's specification also holds. Then after the method execution the postcondition Q' from σ will hold and this again implies the postcondition Q that is required by τ.

However, in general we must assume that the type τ has been specified using a different mathematical domain T than the domain Σ used in σ's specification. As an example, let us take the type Int_Bag. It is convenient here to use as the domain T the set of functions b from integers to nonnegative integers which are nonzero for only a finite number of arguments (with the intuition that the number n is $b(n)$ times present in the bag represented by b). Then we can specify the methods put and get as follows:

$$\{\mathbf{true}\}\ \mathbf{put}(n)\ \{b(n) = b_0(n) + 1 \wedge \forall i(i \neq n \rightarrow b(i) = b_0(i))\},$$

$$\{\exists i(b(i) > 0)\}\ \mathbf{get}()\ \{b_0(r) = b(r) + 1 \wedge \forall i(i \neq r \rightarrow b(i) = b_0(i))\}.$$

Now the above way of showing that `Int_Stack` is a subtype of `Int_Bag` will not work because of the difference in domains.

Therefore in order to show that a type σ is a subtype of the type τ, we require the existence of a function $\phi : \Sigma \rightarrow \mathrm{T}$, called *transfer function*, that maps the mathematical domain Σ associated with σ to T, the one associated with τ. This time we do not need an extra invariant, because we can assume that Σ has been chosen small enough to exclude all the values that cannot actually occur. We now require that for every method specification $\{P\}m(\bar{p})\{Q\}$ occurring in τ's specification there should be a method specification $\{P'\}m(\bar{p})\{Q'\}$ in the specification of σ such that

1. $P \circ \phi \rightarrow P'$.
2. $Q' \rightarrow Q \circ \phi$.

Again $P \circ \phi$ can be obtained from P by replacing the abstract state of τ (in our example: b) by ϕ applied to the abstract state of σ (in our example: s) and analogously for the old values of the abstract states (b_0 and s_0).

Using this definition we can show that `Int_Stack` is actually a subtype of `Int_Bag`. We define the transfer function ϕ as follows: If s is a finite sequence of integers then $\phi(s)$ is that function b from integers to nonnegative integers that maps an integer i to the number of times that i occurs in the sequence s, or formally:

$$\phi(s)(i) = \#\{\, k \mid s_{(k)} = i \,\}.$$

We then have to prove the following implications:

- For the method `put`:
  1. $\mathbf{true} \rightarrow \mathbf{true}$
  2. $s = s_0 * \langle n \rangle \rightarrow$
     $\phi(s)(n) = \phi(s_0)(n) + 1 \wedge \forall i(i \neq n \rightarrow \phi(s)(i) = \phi(s_0)(i))$
- For the method `get`:
  1. $\exists i(\phi(s)(i) > 0) \rightarrow s \neq \langle\rangle$
  2. $s_0 = s * \langle r \rangle \rightarrow$
     $\phi(s_0)(r) = \phi(s)(r) + 1 \wedge \forall i(i \neq r \rightarrow \phi(s)(i) = \phi(s_0)(i))$

The reader can easily verify these implications for himself.

Note that the transfer function ϕ need not be injective. It is often the case (also in this example) that the elements of T contain less information than the elements of Σ. In our example, in a bag, the information on the order in which the elements have been inserted has been lost. Therefore several stacks, differing only in the order of their elements, are mapped onto the same bag.

On the basis of the above definitions we obtain the following desirable property:

Theorem.
If a class C implements a type σ and σ is a subtype of τ, then C implements τ.

We can prove this as follows: Suppose that the class C implements the type σ. Then we have a representation function $f : C \rightarrow \Sigma$ from the set C of concrete states of objects of class C to the mathematical domain Σ associated with the type σ. Furthermore we have a representation invariant I, which holds for each newly created and initialized instance of C, and which is preserved

by each method in the class C. Finally, for each method specification $\{P\}m(\bar{p})\{Q\}$ that occurs in the specification of σ we know that the class C has a method m and that this satisfies

$$\{P \circ f \wedge I\}\, m(\bar{p})\, \{Q \circ f \wedge I\}.$$

Now let us further suppose that σ is a subtype of τ. Then there is a transfer function $\phi : \Sigma \to \mathrm{T}$ and for each method specification $\{P\}m(\bar{p})\{Q\}$ in τ's specification there is a corresponding method specification $\{P'\}m(\bar{p})\{Q'\}$ in σ's specification such that

$$P \circ \phi \to P' \qquad \text{and} \qquad Q' \to Q \circ \phi. \tag{6}$$

In order to prove that C implements τ, we can take the same representation invariant I, since it holds initially for each object and it is preserved by each method of the class C. For the representation function $g : \mathcal{C} \to \mathrm{T}$ we take the composition $\phi \circ f$ of the old representation function $f : \mathcal{C} \to \Sigma$ and the transfer function $\phi : \Sigma \to \mathrm{T}$. Now for every method specification $\{P\}m(\bar{p})\{Q\}$ in τ's specification we must prove that the class C has a method m and that it satisfies

$$\{P \circ \phi \circ f \wedge I\}\, m(\bar{p})\, \{Q \circ \phi \circ f \wedge I\}. \tag{7}$$

From the fact that σ is a subtype of τ we know that there is a corresponding method specification $\{P'\}m(\bar{p})\{Q'\}$ in σ's specification such that (6) holds. From this we can conclude

$$P \circ \phi \circ f \wedge I \to P' \circ f \wedge I \qquad \text{and} \qquad Q' \circ f \wedge I \to Q \circ \phi \circ f \wedge I. \tag{8}$$

Furthermore from the fact that C implements σ we know that C has a method m satisfying

$$\{P' \circ f \wedge I\}\, m(\bar{p})\, \{Q' \circ f \wedge I\}. \tag{9}$$

By the well-known consequence rule, using (8) and (9) we can conclude that (7) holds, which proves the theorem.

Up to this point, we have considered the case where for each method the types of the parameters and the result are equal in the subtype σ and the supertype τ. Now we can generalize this to incorporate the contravariant parameter type rule and the covariant result type rule. This generalized definition requires that for each method m in the specification of τ there should be a method m in the specification of σ such that

- The number n of parameters of m should be equal in σ and τ.
- For each parameter p_i of m, let π_i^τ be its type in τ's specification and π_i^σ its type in the specification of σ. Then it should be the case that $\pi_i^\tau < \pi_i^\sigma$. (The inclusion is in the other direction than the inclusion $\sigma < \tau$, which is the reason for the term 'contravariant'.) It follows that we have a transfer function $\psi_i : \Pi_i^\tau \to \Pi_i^\sigma$ that translates abstract states of the type π_i^τ into abstract states of π_i^σ.
- If the method m in τ's specification returns a result of type ρ^τ then it should also return a result in the specification of σ, and for its type ρ^σ we require $\rho^\sigma \rho^\tau$. Therefore we have a transfer function $\chi : R^\sigma \to R^\tau$ for translating the corresponding abstract states. On the other hand, if m does not return a result in τ it should not return a result in σ either.
- Now suppose that the method specification in τ has the form

  $$\left\{P(t, p_1^\tau, \ldots, p_n^\tau)\right\}\, m(p_1^\tau, \ldots, p_n^\tau)\, \left\{Q(t, t_0, p_1^\tau, \ldots, p_n^\tau, r^\tau)\right\}$$

  and that in σ it has the form

  $$\left\{P'(s, p_1^\sigma, \ldots, p_n^\sigma)\right\}\, m(p_1^\sigma, \ldots, p_n^\sigma)\, \left\{Q(s, s_0, p_1^\sigma, \ldots, p_n^\sigma, r^\sigma)\right\}.$$

  Then we require that the following implications hold:

1. $P(\phi(s), p_1^\tau, \ldots, p_n^\tau) \rightarrow P'(s, \psi_1(p_1^\tau), \ldots, \psi_n(p_n^\tau))$.
2. $Q'(s, s_0, \psi_1(p_1^\tau), \ldots, \psi_n(p_n^\tau), r^\sigma) \rightarrow Q(\phi(s), \phi(s_0), p_1^\tau, \ldots, p_n^\tau, \chi(r^\sigma))$.

Here $\phi : \Sigma \rightarrow \mathrm{T}$ is the transfer function mapping abstract state of σ into abstract states of τ. The transfer functions $\psi_i : \Pi_i^\tau \rightarrow \Pi_i^\sigma$ and $\chi : R^\sigma \rightarrow R^\tau$ have been introduced above.

If we generalize in the same way the requirements for a class to implement a type, then the theorem above can be shown to hold again.

It should be clear that the above only constitutes a sketch of a formalism that could serve to express those aspects of object behaviour that are not captured by a signature. The above example specifications (`Int_Stack` and `Int_Bag`) are relatively simple, and it is not yet clear how well the formalism is suited to specify more complicated objects. Of course, parallelism is not dealt with at all. Moreover, the above specifications describe a complete type, and it is not yet clear how they should be split into individual properties (to be associated with the property identifiers) that describe independent aspects of the behaviour (associating a property identifier with a single method specification will not work: the `LIFO` property, for example, cannot be described in this way).

Nevertheless, the formalism illustrates several important points. First of all, it makes sense to include more information in a type than only a signature: As soon as subtyping is allowed and objects of different classes can belong to the same type (which is true for all well-known object-oriented languages with typing and inheritance), a type that consists only of a signature does not describe how its elements react to messages in full detail. Therefore, in order to be able to construct correct programs, the signature information must be complemented with other information that describes the object behaviour in more detail. In our example, the signature information is not enough to express that we want a stack; additional information is necessary to exclude other data structures, like queues.

The second point is that the specification should describe the object behaviour without referring to the concrete internal structure of the objects. This is necessary to allow a specification of a stack that is satisfied by both the class `AIS` and the class `IIS` (and several others). However, this does not mean that the specification should not refer to an internal state at all. By specifying the object behaviour in terms of an *abstract state*, into which all relevant concrete states can be translated, it is possible to give a very natural specification, which is at the same time independent of the concrete internal structure of the object.

6 Genericity

Now that we have seen, in Section 4, the basic principles of classes, types, and subtyping in POOL-I, and after the interlude of Section 5, which showed how the behaviour of object could be described formally by specifications, we go on to see a number of derived concepts that are present in POOL-I. These concepts are instances of the general idea of *genericity*, which means that we leave some types unspecified in order to have the flexibility fill them in later as we wish.

6.1 Simple genericity

In Section 4.4 we introduced stacks for integers. Of course, stacks are not only useful for storing integers. There are many applications where we want to have stacks of other types. We would like to have a single definition that captures all these different kinds of stack. Generic types provide a way to do this because they allow us to leave the type of the elements stored in the stack open, as a parameter.

Example 16: The generic type Stack.

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

The generic type Stack has a type parameter C, which denotes the type of elements that are stored in the stack. For instance, the type Stack (Person) is the type of stacks that stores persons. In itself, Stack is not a type (there are no objects of type Stack), but it becomes a type if it is provided with a type parameter (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.

Example 17: The generic class AS (array stack).

```
CLASS AS (C)
VAR t : Int       := 0
    a : Array (C) := Array (C).new (1,20)

METHOD get () : C
BEGIN IF t = 0
      THEN RESULT NIL
      ELSE RESULT a[t]; t := t - 1
      FI
END get

METHOD put (c : C) : AIS
BEGIN IF a@ub <= t                    %% @ub selects upper bound
      THEN a ! take_size (1, t+20)    %% changes the array bounds
      FI;
      t := t + 1;
      a[t] := c;
      RESULT SELF
END put

PROPERTY LIFO
END AS
```

Again, AS in itself is not a class (there are no instances of it), but it becomes a class when it is parameterized by an arbitrary type (it is possible to create instances of AS(Person)). Note that a generic class is parameterized by a type, not by a class: we are not referring to the internal structure of the objects in the parameter type, but to the way in which they can be used.

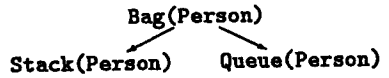
We say that the generic class AS implements the generic type Stack. This means that for each value of the parameter C, AS(C) implements Stack(C). For example, AS(Person) implements Stack(Person). Notice that the type Int_Stack is equivalent to the type Stack(Int), i.e., each object of type Stack is also an object of type Stack(Int) and the other way around. Similarly we have that the class AIS is an implementation of the type Stack(Int) and that AS(Int) implements Int_Stack.

Example 18: The types Bag (generic bag) and Queue (generic queue).

```
TYPE Bag (C)
METHOD get () : C
METHOD put (C) : Bag (C)
END Bag

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

We cannot say that the Stack and Queue are subtypes of Bag, since they are not types but generic types. However, for each value of the parameter C we have that Stack(C) and Queue(C) are subtypes of Bag(C). For instance, We have the following diagram of subtypes.



What happens when we substitute two parameters that are subtypes of each other? For instance, is Stack(Employee) a subtype of Stack (Person)? We see that the answer is no: The method put requires an argument of type Employee in the first case and an argument of type Person in the second case. This is not allowed by the contravariant parameter type rule. The subtyping relationship the other way around conflicts with the covariant result type rule. Therefore in general we do not have a subtyping relationship between different instantiations of the same generic class. However, there are cases where such a relationship holds.

Example 19: The types Dispenser and Receptor.

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

TYPE Receptor (C)
METHOD put (C) : Receptor (C)
END Receptor
```

Both types Dispenser and Receptor are supertypes of the type Bag. The first one can accept elements, the other can only return elements. The type Dispenser(Employee) is a subtype of the type Dispenser(Person), because the method get has result types that are in the right subtyping relation (i.e., they satisfy the covariant result type rule). On the other hand the type Receptor(Person) is a subtype of Receptor(Employee). In order to make this explicit to the readers of the program, we are allowed to introduce the attribute COVAR, or CONTRA to denote the dependency on the type on the parameter. These attributes will induce the compiler to check this dependency on the parameter type. Thus the generic types Dispenser and Receptor can be introduced as follows.

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

TYPE Receptor (CONTRA C)
METHOD put (C) : Receptor (C)
END Receptor
```

Here, it is stated explicitly that the generic type Dispenser depends covariantly on its parameter, whereas for the type Receptor a contravariant dependency holds. We have seen that we cannot put such a attribute in the generic type Bag. In the case that we want to be explicit that there does not exist a subtyping relationship between several instances of the same generic type we may add the attribute NOVAR in the type heading. For instance, the type Bag could have been introduced as follows:

```
TYPE Bag (NOVAR C)
METHOD get () : C
METHOD put (C) : Bag (C)
END Bag
```

The types Dispenser(C) and Receptor(C) seem useless at the first glance. Objects of the type Dispenser(C) only distributes objects of type C, apparently without way to be refilled. Objects of the type Receptor(C) only accept objects of type C without any way of using them. However, note that that fact that an object belongs to the type Dispenser(C) only means that it has *at least* the method get, but that it may have more methods. For example, the types Dispenser(C) and Receptor(C) are supertypes of Bag(C), so an object b of the latter type (or a queue, for that matter) may have been passed to one client p as an element of the type Receptor(C) and to another client c as an element of Dispenser(C). In this way p can only put objects of type C into b, and c can only get them out, but together they constitute a perfectly sensible use of this object. This technique, passing an object to a client as a member of a type that mentions only a subset of the possible operations, is a simple, but powerful way of working with *capabilities* (see, e.g., [WLH81]).

6.2 Bounded genericity

The simple generic classes introduced above cannot make any assumptions about the elements of their type parameters, since any type could be filled in for these parameters. In particular, a generic class cannot send any message to an element of its type parameters, since it is not known whether the actual type that will be filled in for the parameter has an appropriate method. It cannot assign such an element of a parameter type to a variable of a different type, since it is not sure that the parameter type is indeed a subtype of the other one. In many applications it is useful to have some more information about the types, for instance, the presence of certain methods or properties. Because of this, the parameters may be *bounded* from above with a given type, which means that only subtypes of the given type may be filled in for the parameter.

Example 20: The types Ordered and Sorter.

```
TYPE Ordered

METHOD less (Ordered) : Bool
       PROPERTY Static          %% result independent of moment of calling.
                Antisymmetric   %% x!less(y) -> ~(y!less(x))
                Transitive      %% x!less(y) & y!less(z) -> x!less(z)
                Linear          %% x!less(y) | y!less(x) | x == y
END Ordered

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

The type Ordered has a method less, with object properties to denote that it is an ordering. The type Sorter accepts as parameters only subtypes of Ordered. Therefore an implementation of this type may use the fact that for objects of type X the method less is available, with the properties Static, Antisymmetric, Transitive and Linear.

Example 21: The class Quick_Sorter.

```
CLASS Quick_Sorter (X < Ordered)

METHOD sort (a: Array (X)) : Array (X)  %% Returns a, sorted with quicksort.
BEGIN RESULT quick_sort(a, a@lb, a@ub)
END sort

INTERNAL METHOD quick_sort (a: Array (X), left, right: Int) : Array (X)
              %% Sort the elements of a between left and right.
TEMP i, j: Int
     d: X
BEGIN i := left;
      j := right;
      d := a[(left + right)/2];        %% elements less than  d  go to the
                                       %% left, others go to the right.
      WHILE i <= j                     %% elements between i and j (bounds
                                       %% inclusive) are not yet considered.
      DO WHILE a[i] ! less(d)          %% use method less of type X.
         DO i := i+1                   %% a[i] is already left.
         OD;
         WHILE d ! less(a[j])          %% use method less of type X.
         DO j := j-1                   %% a[j] is already right.
         OD;
         IF i < j                      %% a large element is left and a small
                                       %% one is right: interchange.
         THEN [a[i], a[j]] := [a[j], a[i]]
         FI
      OD;
      IF left < j                      %% sort the left part.
      THEN sort(a, left, j)
      FI;
      IF right > i                     %% sort the right part.
      THEN sort(a, i, right)
      FI;
      RESULT a
END quick_sort

END Quick_Sorter
```

The class Quick_Sorter implements the type Sorter by using quicksort for sorting. In this example we see the use of the method less. Only because the type parameter X is known to be a subtype of Ordered we know that the method less is available. For instance we may sort an integer array int_ar: Array(Int) in the following way:

```
Quick_Sorter(Int).new() ! sort(int_ar)
```

Example 22: The type Part_Ordered.

```
TYPE Part_Ordered

METHOD less (Part_Ordered) : Bool
       PROPERTY Static           %% result independent of moment of calling.
                Antisymmetric    %% x!less(y) -> ~(y!less(x))
                Transitive       %% x!less(y) & y!less(z) -> x!less(z)
END Part_Ordered
```

The type Part_Ordered, is the type of all partially ordered structures. It is a strict supertype of Ordered. Therefore it is not allowed to fill the place of the parameter type C in the generic type Sorter by a type that is a subtype of Part_Ordered but not of Ordered. It is correct that we should not be allowed to do so because, in general, objects of type Sorter(C), and certainly instances of the class Quick_Sorter(C) are not able to sort arrays if the ordering on the type C is a partial order that is not total. Nevertheless it is possible to define what it means that an array a is sorted according to a partial order, namely a[i] ! less (a[j]) ⇒ i < j (we say that the array is *topologically sorted*). The generic class Top_Sorter below provides such a topological sorting method.

Example 23: The class Top_Sorter.

```
CLASS Top_Sorter (X < Part_Ordered)

METHOD sort (a: Array (X)) : Array (X) %% Returns a, topologically sorted.
TEMP min: X
     idx, lidx : Int
BEGIN %% loop invariant: a[a@lb, i-1] is topologically sorted and there are
      %% no k,l with a@lb <=k < i <= l <= a@ub such that a[l] ! less(a[k]).
      FOR i FROM a@lb TO a@ub
      DO min := a[i];
         idx := i;
         midx := i;
         %% loop invariant: min = a[midx] and for all j with i<=j<=idx
         %% we have ~(a[j] ! less(min))
         WHILE idx < a@ub
         DO idx := idx+1;
            IF a[idx] ! less (min)     %% use the method less of type X
            THEN min := a[idx];        %% update min and lidx
                 midx := idx
            FI
         OD;
         [a[i], a[midx]] := [a[midx], a[i]]
                                       %% put the next minimal element at place i
      OD;
      RESULT 1
END sort

END Top_Sorter
```

Notice that for each linearly ordered X the class Top_Sorter(X) also sorts arrays of type X, because Ordered < Part_Ordered.

6.3 Dynamic type manipulation

The mechanisms for genericity introduced above allow us to delay fixing the type parameters for some time (for example, we can store the generic class Quick_Sorter in a library, and use it in many different programs), but nevertheless the parameters must be filled in before the complete program is compiled. In this section we see that by allowing type parameters in methods and routines, it is possible to fill in these parameters even after the program has already started to run. The parameter types (of the object parameters) and the result type of such a method or routine may be dependent on a generic type parameter. As with generic classes the type parameters of methods and routines can be bound from above by a given type. This enables the method/routine to use additional information about the objects of the type parameter.

In order to distinguish them from object parameters, type parameters are indicated by the keyword TYPE or by an upper bound on the type (or both). For instance, the method headings `m(x < B: TYPE)` and `m(x < B)` are equivalent.

Example 24: The class `Universal_Quick_Sorter`.

```
CLASS Universal_Quick_Sorter

METHOD sort (X < Ordered, a: Array (X)) : Array (X)
BEGIN RESULT quick_sort(X, a, a@lb, a@ub)
END sort

INTERNAL
METHOD quick_sort (X < Ordered, a: Array (X), left, right: Int) : Array (X)
                %% Sort the elements of a between left and right.
TEMP i, j: Int
     d: X
BEGIN i := left;
      j := right;
      d := a[(left + right)/2];        %% elements less than  d  go to the
                                       %% left, others go to the right.
      WHILE i <= j                     %% elements between i and j (bounds
                                       %% inclusive) are not yet considered.
      DO WHILE a[i] ! less (d)         %% use method less of type X.
         DO i := i+1                   %% a[i] is already left.
         OD;
         WHILE d ! less (a[j])         %% use method less of type X.
         DO j := j-1                   %% a[j] is already right.
         OD;
         IF i < j                      %% a large element is left and a small
                                       %% one is right: interchange.
         THEN [a[i], a[j]] := [a[j], a[i]]
         FI
       OD;
       IF left < j                     %% sort the left part.
       THEN quick_sort(X, a, left, j)
       FI;
       IF right > i                    %% sort the right part.
       THEN quick_sort(X, a, i, right)
       FI;
       RESULT 1
END quick_sort

END Universal_quick_Sorter
```

In contrast to objects of type `Sorter(C)`, which can only be used for a single type of arrays, objects of the type `Universal_Sorter` be used for any type of arrays (as long as the element type is ordered). In addition, during 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 determined during program execution. Notice the appearance of the type parameter `X` in both method headings. For instance we may sort an integer array `int_ar: Array(Int)`, using the universal sorter in the following way: `Universal_Quick_Sorter.new() ! sort(Int, int_ar)`.

It is also possible to use a type as a parameter in the creation of a new object. This is called a type new-parameter. Because the new-parameters will only be known when the object is created, they may not appear in routines and in method headers.

Example 25: The class Init_Dispenser.

```
CLASS Init_Dispenser (C)

NEWPAR (E < C, e: E, n: Int)
VAR cts := Array(E).new(1,n)
INIT FOR i TO n DO cts[i] := e OD %% initialize with n copies of e
TINI

METHOD get () : C
BEGIN IF #cts > 0
      THEN RESULT cts@low %% take first element and increase lower bound
      ELSE RESULT NIL
      FI
END get

END Init_Dispenser
```

The generic class Init_Dispenser implements the generic type Dispenser. The objects of this class are initially filled by n copies of an element e of type E, which is a subtype of the parameter C of the class. The element e will be distributed n times to callers of the method get. For instance we can create a dispenser with parameter Person, which contains 10 copies of an employee. The result of the method get will be of type Person.

```
d := Dispenser(Person).new(Employee, e, 10);
p := d ! get();
```

6.4 Dynamic type analysis

In order to make more effective use of the possibilities of dynamic type parameters, a mechanism for dynamic type analysis is present in the language. This mechanism has the form of a case statement, which compares a type σ with another type τ and $\sigma < \tau$ then a corresponding piece of code is executed. Within this piece of code we can make use of the fact that $\sigma < \tau$. For instance we may execute all methods mentioned in the type τ in objects of type σ.

The method get of the class Init_Dispenser(C) above always returns elements of type C, even though the dispenser object may have been initialized with elements of a more specific type E. Therefore, the special features of the type E cannot be used, even if the type is given in the routine new. For instance, with the dispenser made above the fact that it is filled with employees is not used. However, by using dynamic type analysis, we can make use of the fact that we have stored elements of a subtype, as shown below.

Example 26: the class Init_Dispenser, continued.

```
METHOD s_get (Y: TYPE) : Y
BEGIN CASE E OF
      Y THEN                    %% E is a subtype of Y
         IF #cts > 0
         THEN RESULT cts@low
         ELSE RESULT NIL        %% Return NIL if the dispenser is empty.
         FI
      ELSE RESULT NIL           %% Return NIL if the type Y was wrong.
      ESAC      •
END s_get
```

```
METHOD refill(Y: TYPE, e: Y, n: Int) : Int
BEGIN CASE Y OF
      E THEN                                  %% Y is a subtype of E
         FOR i TO n DO a@high := e OD;        %% insert e n times into a.
         RESULT n                             %% Answer that n elements are added.
      ELSE RESULT 0                           %% Answer that no elements are added.
      ESAC
END refill
```

The result type of the method `get_s` is equal to the parameter type `Y`. If this type is a supertype of the new-parameter `E` we can return an element, because it is of the right type. For instance, we can send the following messages to the dispenser `d`, created above:

```
e := d ! get_s(Employee);
p := d ! get_s(Person);
i := d ! get_s(Int);
```

Both `e: Employee` and `p: Person` get an object of the right type as result of the method. The variable `i: Int` will get the value `NIL`, because `Int` $\not>$ `Employee`.

The method `refill` can be used to enlarge the contents of the dispenser. The dispenser is implemented using an array `Array(E)`. Therefore the new elements should be of a subtype of `E`. For instance, we may add new elements of type `My_Employee`, but not of type `Int`. However, both the following methods are correctly typed, when `i`, `j` and `k` are of type `Int` and `me` is of type `My_Employee`.

```
i := d ! refill(My_Employee, me, 25);
j := d ! refill(Int, k, 109);
```

As a result `i` has the value 25, because 25 times `me` is inserted to the `Dispenser`. However, `j` will get the value 0, because no integer can be inserted to the dispenser `d`.

A different kind of case statement can check whether a type provides a certain method. The method name is determined dynamically in the form of a string.

Example 27: Adding a field over an array.

```
ROUTINE add_Int_field (T: TYPE, a: Array(T), f: String) : Int
TEMP sum: Int := 0
BEGIN CASE T METHOD "get_" + f OF
      m () : Int
         THEN FOR i FROM a@lb TO a@ub
              DO sum := sum + a[i] ! m()
              OD;
              RESULT sum
      ELSE RESULT NIL
      ESAC
END add_Int_field
```

This routine adds the result of the method `get_f` for all elements of array `a`. If the method `get_f` does not exist for type `T`, or it does not deliver an integer no adding is done and the result `NIL` will be returned. Note that the check whether `get_f` is present, and of the right type is only done once per call of the routine `add_Int_field` instead of once for every object in the array. Example of usage:

```
s := add_Int_Field(Employee, all_employees, "salary")
t := add_Int_Field(Int, a, "salary")
```

The variable `s` gets the sum of the salaries of all employees, listed in the array `all_employees`. The variable `t` will get the value `NIL`, since the type `Int` does not have a method `get_salary(): Int`.

7 Conclusions

In an earlier paper [Ame87a], we have already argued that it is useful to distinguish between a concept that we call *inheritance*, which deals with code sharing among classes, and a concept of *subtyping*, which has to do with specialization in behaviour of objects which can be observed from outside. In this terminology, inheritance is concerned with the internal structure and implementation of the objects and subtyping with their use.

We have seen that a type can in fact be identified with a *specification* of the behaviour of an object: the type comprises all objects that satisfy the specification. Such a specification should abstract from the internal details of the object and concentrate on that part of the behaviour that can be observed from outside by sending messages to the object.

In this paper we have defined a way to specify such a type formally. The best way to do this is *not* to reason about the sequence of the messages that are sent to this object, but to introduce an *abstract state* for each object: a mathematical entity that represents the object at a specific point during its life. Then every method can be specified by expressing its effect on that abstract state by pre- and postconditions. Our types are not only concerned with the signatures of the objects (names of methods and their parameter and result types), but they incorporate more detailed information about the values that are transmitted.

For a given class definition, we have defined under which circumstances this class *implements* a type, which ensures that every instance of the class is an element of the type. This is done by defining a *representation invariant* that describes the possible concrete states of the objects in the class and a *representation function* that maps these concrete states into the domain of abstract states of the type. Finally we have defined what conditions must be satisfied for one type to be a subtype of another one. Here we use a *transfer function* that maps each abstract state of the subtype to an abstract state of the supertype.

Certain aspects are not yet dealt with here. One of these is how to prove formally that a method of a certain class is correct with respect to a pair of pre- and postconditions. To solve this problem, the theory developed in [Ame86] can provide a useful basis. However, quite a lot of work is still required to fill in the details.

We have shown that subtyping and inheritance can be included in the language POOL-I. Both notions of subtyping and inheritance can be treated separately and they can be seen as independent options for the programmer, which avoids many complications in the language design. In this paper we have concentrated on subtyping, but it turns out that inheritance also becomes much simpler if it is decoupled from subtyping [AvdL90].

Subtyping in POOL-I not only involves signatures, but also properties of the types. These properties are identifiers that are associated with specifications of the behaviour of objects of the type. It is presently not feasible to let the compiler check these specifications, so the compiler only checks the presence of the property identifiers; associating the property identifiers with specifications and verifying that a class satisfies such a specification is left to the conscientious programmer. In this way we can develop the specification formalism further without having to change the language definition.

Acknowledgement

Frank van der Linden has contributed greatly to this work by intensive discussions and by his active work on the design of POOL-I.

References

[Ame86] Pierre America. A proof theory for a sequential version of POOL. ESPRIT Project 415 Document 188, Philips Research Laboratories, Eindhoven, the Netherlands, October 1986.

[Ame87a] Pierre America. Inheritance and subtyping in a parallel object-oriented language. In *ECOOP '87: European Conference on Object-Oriented Programming*, Paris, France, June 15–17, 1987. Springer Lecture Notes in Computer Science, Volume 276, pages 234–242.

[Ame87b] Pierre America. POOL-T: A parallel object-oriented language. In Akinori Yonezawa and Mario Tokoro, editors, *Object-Oriented Concurrent Programming*, pages 199–220. MIT Press, 1987.

[Ame89] Pierre America. Issues in the design of a parallel object-oriented language. *Formal Aspects of Computing*, 1(4):366–411, 1989.

[AvdL90] Pierre America and Frank van der Linden. A parallel object-oriented language with inheritance and subtyping. In *Proceedings of the ECOOP/OOPSLA Conference*, Ottawa, Canada, October, 21–25, 1990.

[Car88] Luca Cardelli. A semantics of multiple inheritance. *Information and Computation*, 76:138–164, 1988.

[CHC89] William Cook, Walt Hill, and Peter Canning. Inheritance is not subtyping. Report STL-89-17, Hewlett-Packard Laboratories, Palo Alto, California, July 1989. In *Proceedings of the ACM SIGACT-SIGPLAN Symposium on Principles of Programming Languages (POPL)*, January 17–19, 1990.

[DN66] Ole-Johan Dahl and Kristen Nygaard. Simula: An ALGOL-based simulation language. *Communications of the ACM*, 9(9):671–678, September 1966.

[GR83] Adele Goldberg and David Robson. *Smalltalk-80, The Language and its Implementation.* Addison-Wesley, 1983.

[Hoa72] C. A. R. Hoare. Proof of correctness of data representations. *Acta Informatica*, 1:271–281, 1972.

[Mey88] Bertrand Meyer. *Object-Oriented Software Construction.* Prentice-Hall, 1988.

[Mit88] John C. Mitchell. Polymorphic type inference and containment. *Information and Computation*, 76:211–249, 1988.

[SB86] Mark Stefik and Daniel G. Bobrow. Object-oriented programming: Themes and variations. *AI Magazine*, 6(4):40–62, January 1986.

[SCB*86] Craig Schaffert, Topher Cooper, Bruce Bullis, Mike Kilian, and Carrie Wilpolt. An introduction to Trellis/Owl. In *Proceedings of the ACM Conference on Object-Oriented Programming, Systems, Languages and Applications (OOPSLA)*, Portland, Oregon, September 1986, pages 9–16.

[Sny86] Alan Snyder. Encapsulation and inheritance in object-oriented programming languages. In *Proceedings of the ACM Conference on Object-Oriented Programming, Systems, Languages and Applications (OOPSLA)*, Portland, Oregon, September 1986, pages 38–45.

[WLH81] William A. Wulf, Roy Levin, and Samuel P. Harbison. *HYDRA/C.mmp: An Experimental Computer System.* McGraw-Hill, 1981.