A BEHAVIOURAL APPROACH TO SUBTYPEING IN OBJECT-ORIENTED PROGRAMMING LANGUAGES*

by P.H.M. AMERICA

Philips Research Laboratories, P.O. Box 80000, 5600 JA Eindhoven, The Netherlands

Abstract

It is argued that in object-oriented programming languages a distinction should be made between inheritance, considered as a mechanism by which classes can share code for variables and methods, and subtyping, which expresses specialization in functionality. In contrast to inheritance, subtyping should not be based on the internal structure of the objects and the code they execute, but on that part of their behaviour that can be observed by sending messages to them. A formalism is defined by which one can specify this behaviour, and on the basis of this formalism it is defined when a class implements a type and when one type is a subtype of another type. These definitions are illustrated by some concrete examples.

Keywords: formal methods, inheritance, object-oriented programming, programming languages, specification, type-checking

1. Introduction

Over the last few years the object-oriented style of programming has become so popular that many people expect that it will become the prevalent style of the 1990s. The Smalltalk-80 language and system 1) has played a very important role in this development, and other languages, such as CLOS 2), C++ 3) and Eiffel 4), have shown that object-oriented programming can be used in a wide variety of circumstances. At Philips Research Laboratories an object-oriented language, POOL 5) is being developed for programming a parallel computer, DOOM 6). In Sec. 2 a short introduction to object-oriented

*) The work described in this paper was done in the context of ESPRIT Project 415: Parallel Architectures and Languages for Advanced Information Processing: A VLSI-directed Approach. A condensed version of this paper was presented at the Workshop on Inheritance Hierarchies in Knowledge Representation and Programming Languages, Viareggio, Italy, February 6–8, 1989.
programming is given; for a more extensive treatment see refs. 4 and 7.

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. 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 Sec. 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, 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, 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 in order 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 may be developed. The paper is structured as follows. Section 2 gives a very brief introduction to object-oriented programming, defining some terminology. In Sec. 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
Subtyping in object-oriented programming languages

Fig. 1. An object.

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'.

The rest of the paper presents a semantic basis for the subtyping relationship between classes of objects. In Sec. 4 we develop a formalism for specifying the external behaviour of objects. Then, in Sec. 5, we study how such specifications determine whether one class of objects can be considered as a subtype of another class.

2. Objects and classes

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 fig. 1). Following the terminology of Smalltalk-80\(^1\), we shall 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, 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.

It is very important 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 fig. 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 the behaviour that can be observed from outside.

The author considers this principle of protection of objects against each other as the basic and essential characteristic of object-oriented programming. (This view is not unique, cf. the locality laws of ref. 10.) It is a refinement of the technique of abstract data types, because it not only protects one type of object against all the other types, but also protects one object against all the others. 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 they all 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 of variable and method declarations. In this way a class definition provides exactly the information that is needed to create new objects. Examples of class definitions will be given in figs 3 and 4.

3. Inheritance and subtyping

The basic idea of inheritance, as it appeared in the first object-oriented languages such as Simula 11) and Smalltalk-80 1), 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.

But 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 thus 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 instances 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 refs. 12 and 13). 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 very 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. In contrast, it is well known that it 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. In order 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\textsuperscript{14}) and Eiffel\textsuperscript{4}) as well as in some without strong typing (e.g. Smalltalk-80\textsuperscript{1}). 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. The goal 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 not only provide some information to the compiler, which can detect certain errors by type checking, but also constitute 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.

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 Stack, which comprises all the objects that have the following behaviour:

The object will accept \texttt{put} and \texttt{get} messages, but it will accept a \texttt{get} message only if the number of \texttt{put} messages already received exceeds the number of \texttt{get} messages. A \texttt{put} message contains one integer as an argument and does not return a result. A \texttt{get} message contains no arguments and it returns as its result the integer that was the argument
Subtyping in object-oriented programming languages

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 for an object to have 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 gives 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 σ ⊑ τ) 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 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 does not return a 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 Stack is a subtype of the type Bag, since every element of Stack is also an element of Bag. It must be admitted, however, 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 Stack is a strict subtype of Bag, i.e. the subtyping relationship does not hold the other way around, but that they nevertheless have the same signature. One could also imagine a type Queue, again with the same signature and with the obvious intuitive meaning, which is a subtype.
of Bag but incomparable with Stack:

\[
\begin{array}{c}
\text{Bag} \\
\downarrow \\
\text{Stack} \\
\downarrow \\
\text{Queue}
\end{array}
\]

4. Specifying object behaviour

In order to obtain a formal grip on types as defined above, it is necessary to be able to specify externally observable properties 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 Sec. 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, for example, 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 above example specifications. For more complex objects, a specification exclusively in terms of sequences of messages is clearly not feasible.

In order to achieve a specification technique that corresponds more closely to our intuition, we use a technique that is very common in dealing with abstract data types: 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 \( \Sigma \) 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 into the stack first). Now the methods put and get can be specified by preconditions and postconditions as follows:

\[
\{ \text{true} \} \text{put}(n) \{ s = s_0 * \langle n \rangle \},
\]

\[
\{ s \neq \langle \rangle \} \text{get}() \{ s_0 = s \ast \langle r \rangle \}.
\]
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 \( \ast \) denotes concatenation of sequences, \(<\) is the empty sequence, and \( <n> \) 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 \( \sigma \) consists of a domain \( \Sigma \), representing the set of possible abstract states of objects of type \( \sigma \), plus a set of method specifications of the form \( \{ P \} m \{ P \} \{ Q \} \), 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 \( \sigma \) 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.

Note that such a specification should also indicate the types of the parameters and the possible results of the methods. We shall briefly come back to this point later. Moreover, a specification as above is intended to express that the methods put and get must be present, but that additional methods are allowed (the specification indicates a minimal signature). We might have chosen to exclude the possibility of additional methods, but this would be contrary to the usual practice in object-oriented programming.

Now the important question is under what conditions the objects of a given class \( C \) are members of a type \( \sigma \), in which case we shall say that the class \( C \) implements the type \( \sigma \). We do this as follows: We require a representation function \( f \colon \mathcal{C} \rightarrow \Sigma \), where \( \mathcal{C} \) 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 \( \Sigma \) is the set of abstract states associated with the type \( \sigma \). 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 \( \Sigma \) that is used in the specification of the type \( \sigma \). 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 \( \mathcal{C} \)). The representation function \( f \) should at least be defined for all concrete states for which the invariant \( I \) holds.

Now in order for the class \( C \) to implement the type \( \sigma \) the following conditions are required to hold.
CLASS AS
VAR t: Int:=0
    a:Array (Int):= Array (Int).new (1, 20)
METHOD put (n: Int)
BEGIN
    IF a@ub <= t
        t:=t+1; a[t]:= n
    FI;
END put
METHOD get (): Int
BEGIN
    IF t=0
        RESULT NIL
    ELSE RESULT a[t]; t:=t-1
    FI
END get
METHOD size (): Int
BEGIN RESULT t
END size
END AS

Fig. 3. The definition of the class AS.

(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 $\sigma$'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 $\sigma$, the class $C$ should also have a method $m$ with parameters $\bar{p}$ of the right number and types and this method should satisfy

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

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
Subtyping in object-oriented programming languages

CLASS IS
VAR t: Int:=NIL
    r: IS:=NIL
    c: Int:=0
METHOD put (n: Int)
BEGIN
    IF r == NIL
    THEN r:= IS.new ()
    ELSIF c > 0
    THEN r ! put (t)
    FI;
    t:=n; c:=c + 1
END put
METHOD get (): Int
BEGIN
    IF c > 0
    THEN RESULT t;
    t:=r ! get ();
    c:=c - 1
    ELSE RESULT NIL
    FI
END get
END IS

Fig. 4. The definition of the class IS.

a kind of functional composition of the function $f: \mathcal{C} \rightarrow \Sigma$ from concrete states to abstract states and the predicate $P$, which maps abstract states into truth values $\Sigma \rightarrow \{t, f\}$. (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 example, assume that we have a class AS (for ‘array stack’) as shown in fig. 3. It has two variables: $a$ of type $\text{Array}(\text{Int})$ and $t$ of type $\text{Int}$. We want to show that this class implements the type $\text{Stack}$. Now we can define a representation function $f$ as follows:

$$f(a, t) = \langle a[1], \ldots, a[t] \rangle.$$  \hfill (1)

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

$$\{I\} \text{put}(n) \{f(a, t) = f(a_0, t_0) * \langle n \rangle \land I\}. \quad (2)$$

If we fill in the definition of $f$ from eq. (1) and simplify we obtain

$$\{I\} \text{put}(n) \{t = t_0 + 1 \land a[t] = n \land \forall i(1 \leq i < t \rightarrow a[i] = a_0[i]) \land I\}, \quad (3)$$

which can be verified from the program text.

The method get should satisfy

$$\{f(a, t) \neq \langle \rangle \land I\} \text{get()} \{f(a_0, t_0) = f(a, t) * \langle r \rangle \land I\}. \quad (4)$$

This can be expanded into

$$\{t \neq 0 \land I\} \text{get()} \{t_0 = t + 1 \land r = a_0[t_0] \land \forall i(1 \leq i \leq t \rightarrow a[i] = a_0[i]) \land I\}. \quad (5)$$

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 eq. (3)). Since all these requirements are fulfilled, the class AS implements the type 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, IS (for 'inefficient stack'), that also implements the specification Stack. Its code is given in fig. 4. 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

$$c \geq 0 \land (c > 0 \rightarrow \neg(r = = \text{NIL}))$$
Subtyping in object-oriented programming languages

For the representation function \( g \) we take the following definition:

\[
g(t, r, c) = \begin{cases} 
\langle \rangle & \text{if } c = 0 \\
\langle t \rangle \ast r \cdot s & \text{if } c > 0 
\end{cases}
\]

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 \( r \cdot s \). (Note that we could declare the variable \( r \) as having type \textbf{Stack}, because all elements of \textbf{IS} also belong to the type \textbf{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 \textbf{Stack}, these conditions are clearly satisfied.

5. The subtyping relationship

Now that we have defined what constitutes a type and when a class implements a type, we go on to answer the question under which conditions a type \( \sigma \) is a subtype of a type \( \tau \): \( \sigma \subseteq \tau \). At first, it seems sufficient to require that for every method specification \( \{P\}m(p)\{Q\} \) occurring in \( \tau \)'s specification there should be a method specification \( \{P'\}m(p)\{Q'\} \) in the specification of \( \sigma \) such that the latter implies the former, which can be expressed by \( P \equiv P' \) and \( Q' \rightarrow Q \). Under these circumstances we can indeed use any element of \( \sigma \) whenever an element of \( \tau \) is expected. When we send such an object a message listing the method \( m \), using it as an element of \( \tau \) guarantees that initially the precondition \( P \) will hold. By the implication \( P \rightarrow P' \) we can conclude that the precondition \( P' \) in \( \sigma \)'s specification also holds. Then after the method execution the postcondition \( Q' \) from \( \sigma \) will hold and this again implies the postcondition \( Q \) that is required by \( \tau \).

However, in general we must assume that the type \( \tau \) has been specified using a different mathematical domain \( T \) than the domain \( \Sigma \) used in \( \sigma \)'s specification. As an example let us take the type \textbf{Bag}. It is convenient here to
use as the domain $T$ the set of functions $b$ from integers to non-negative integers which are non-zero 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:

\[ \{\text{true}\} \text{put}(n) \{b(n) = b_0(n) + 1 \land \forall i (i \neq n \rightarrow b(i) = b_0(i))\}, \]

\[ \{\exists i (b(i) > 0)\} \text{get}() \{b_0(r) = b(r) + 1 \land \forall i (i \neq r \rightarrow b(i) = b_0(i))\}. \]

Now the above way of showing that Stack is a subtype of Bag will not work because of the difference in domains.

Therefore in order to show that a type $\sigma$ is a subtype of the type $\tau$, we require the existence of a function $\phi: \Sigma \rightarrow T$, called a transfer function, that maps the mathematical domain $\Sigma$ associated with $\sigma$ to $T$, the domain associated with $\tau$. This time we do not need an extra invariant, because we can assume that $\Sigma$ 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 $\tau$’s specification there should be a method specification $\{P'\}m(\bar{p})\{Q'\}$ in the specification of $\sigma$ such that

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

Again $P \circ \phi$ can be obtained from $P$ by replacing the abstract state of $\tau$ (in our example, $b$) by $\phi$ applied to the abstract state of $\sigma$ (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 Stack is actually a subtype of Bag. We define the transfer function $\phi$ as follows: If $s$ is a finite sequence of integers then $\phi(s)$ is that function $b$ from integers to non-negative 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. $\text{true} \rightarrow \text{true}$
  2. $s = s_0 \ast \langle n \rangle \rightarrow \phi(s)(n) = \phi(s_0)(n) + 1 \land \forall i (i \neq n \rightarrow \phi(s)(i) = \phi(s_0)(i))$

- For the method get

  1. $\exists i (\phi(s)(i) > 0) \rightarrow s \neq \langle \rangle$
  2. $s_0 = s \ast \langle r \rangle \rightarrow \phi(s_0)(r) = \phi(s)(r) + 1 \land \forall i (i \neq r \rightarrow \phi(s)(i) = \phi(s_0)(i))$
Subtyping in object-oriented programming languages

The reader can easily verify these implications for himself.

Note that the transfer function $\phi$ 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 $\Sigma$. 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 $\sigma$ and $\sigma$ is a subtype of $\tau$, then the class
$C$ implements the type $\tau$.

We can prove this as follows. Suppose that the class $C$ implements the type
$\sigma$. Then we have a representation function $f: \mathcal{C} \rightarrow \Sigma$ from the set $\mathcal{C}$ of concrete
states of objects of class $C$ to the mathematical domain $\Sigma$ associated with the
type $\sigma$. 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 $\sigma$ we know that the class $C$ has a method
$m$ and that this satisfies

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

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

$$P \circ \phi \rightarrow P' \quad \text{and} \quad Q' \rightarrow Q \circ \phi.$$  \hspace{1cm} (6)

In order to prove that $C$ implements $\tau$, 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 $g: \mathcal{C} \rightarrow T$ we take the
composition $\phi \circ f$ of the old representation function $f: \mathcal{C} \rightarrow \Sigma$ and the transfer
function $\phi: \Sigma \rightarrow T$. Now for every method specification $\{P\}m(\bar{p})\{Q\}$ in $\tau$'s
specification we must prove that the class $C$ has a method $m$ and that it satisfies

$$\{P \circ \phi \circ f \land I\}m(\bar{p})\{Q \circ \phi \circ f \land I\}.$$  \hspace{1cm} (7)

From the fact that $\sigma$ is a subtype of $\tau$ we know that there is a corresponding
method specification $\{P'\}m(\bar{p})\{Q'\}$ in $\sigma$'s specification such that eq. (6) holds.
From this we can conclude
\[ P \circ \phi \circ f \land I \rightarrow \theta \circ f \land I \quad \text{and} \quad \eta \circ f \land I \rightarrow \psi \circ f \land I. \tag{8} \]

Furthermore from the fact that C implements \( \sigma \) we know that C has a method \( m \) satisfying
\[ \{ \theta \circ f \land I \} m(\bar{p}) \{ \eta \circ f \land I \}. \tag{9} \]

By the well-known consequence rule, using eqs. (8) and (9) we can conclude that eq. (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 \( \sigma \) and the supertype \( \tau \). Now we can generalize this into the following \textit{contravariant parameter type rule}, which requires that for each method \( m \) in the specification of \( \tau \) there should be a method \( m \) in the specification of \( \sigma \) such that

- The number \( n \) of parameters of \( m \) should be equal in \( \sigma \) and \( \tau \).
- For each parameter \( p_i \) of \( m \), let \( \pi_i^\tau \) be its type in \( \tau \)'s specification and \( \pi_i^\sigma \) its type in the specification of \( \sigma \). Then it should be the case that \( \pi_i^\tau \subseteq \pi_i^\sigma \). (The inclusion is in the other direction than the inclusion \( \sigma \subseteq \tau \), which is the reason for the term 'contravariant'.) It follows that we have a transfer function \( \psi: \Pi_i^\tau \rightarrow \Pi_i^\sigma \) that translates abstract states of the type \( \pi_i^\tau \) into abstract states of \( \pi_i^\sigma \).
- If the method \( m \) in \( \tau \)'s specification returns a result of type \( \rho^\tau \) then it should also return a result in the specification of \( \sigma \), and for its type \( \rho^\sigma \) we require \( \rho^\sigma \supseteq \rho^\tau \). Therefore we have a transfer function \( \chi: R^\sigma \rightarrow R^\tau \) for translating the corresponding abstract states. In contrast, if \( m \) does not return a result in \( \tau \) it should not return a result in \( \sigma \) either.\(^*)\
- Now suppose that the method specification in \( \tau \) has the form
  \[ \{ P(t, p_1^\tau, \ldots, p_n^\tau) \} m(p_1^\tau, \ldots, p_n^\tau) \{ Q(t, t_0, p_1^\tau, \ldots, p_n^\tau, r^\tau) \} \]
  and that in \( \sigma \) it has the form
  \[ \{ P'(s, p_1^\sigma, \ldots, p_n^\sigma) \} m(p_1^\sigma, \ldots, p_n^\sigma) \{ Q(s, s_0, p_1^\sigma, \ldots, p_n^\sigma, r^\sigma) \}. \]

\(^*)\text{In fact, we could generalize this even further by allowing more parameters in } \tau \text{ than in } \sigma \text{ and/or returning a result in } \sigma \text{ but not in } \tau.\)
Then we require that the following implications hold:

(1) $P(\phi(s), p^1, \ldots, p^n) \rightarrow P'(s, \psi_1(p^1), \ldots, \psi_n(p^n))$.

(2) $Q'(s, s_0, \psi_1(p^1), \ldots, \psi_n(p^n), r^\sigma) \rightarrow Q(\phi(s), \phi(s_0), p^1, \ldots, p^n, \chi(r^\sigma))$.

Here $\phi: \Sigma \rightarrow T$ is the transfer function mapping abstract state of $\sigma$ into
abstract states of $\tau$. The transfer functions $\psi_i: T^{\Pi_i} \rightarrow T^{\Pi_i^\sigma}$ 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.

6. Conclusions

In an earlier paper\textsuperscript{12}, 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 preconditions 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 precondition and postcondition pair. To solve this problem, the theory developed in ref. 16 can provide a useful basis. However, quite a lot of work is still required to fill in the details. In contrast, it seems interesting to see how these ideas can be integrated into a concrete object-oriented programming language. Currently it is not realistic to assume that a compiler can automatically verify the conditions of subtyping, because it would need a complete theorem prover. However, it would be possible for the compiler only to check the signatures and to leave the responsibility for the preconditions and postconditions to the programmer. A language along these lines would support very well the modern techniques of object-oriented software development (see, for example, ref. 4).

REFERENCES

1) A. Goldberg and D. Robson. Smalltalk-80, The Language and Its Implementation, Addison-Wesley, Reading, MA, 1983.
Subtyping in object-oriented programming languages


Author

P. H. M. America: Drs. degree (Mathematics and Computer Science), University of Utrecht, The Netherlands, 1982; Ph.D., Free University of Amsterdam, The Netherlands, 1989; Philips Research Laboratories, Eindhoven, 1982–. Since 1984 he is responsible for the design of the parallel object-oriented programming language POOL and he is working on formal semantics and verification techniques for parallel and object-oriented languages.