Three Discussions on Object-Oriented Typing

Jens Palsberg
palsberg@daimi.aau.dk

Michael I. Schwartzbach
mis@daimi.aau.dk

Computer Science Department
Aarhus University
Ny Munkegade
DK-8000 Arhus C, Denmark

August 1991

1 Introduction

This paper summarizes three discussions conducted at the ECOOP'91 W5 Workshop on "Types, Inheritance, and Assignments" Tuesday July 16, 1991 in Geneva, Switzerland, organized by the authors.

Participants at the workshop were: Birger Andersen, Andrew Black, Gregor Bochmann, Gilad Bracha, Simon Brock, Brian Brown, David Carrington, Bruce Conrad, Elspeth Cusack, Jeremy Dick, Rainer Fischbach, Elio Giovanetti, Andreas Hense, John Hogg, Rick Holt, Urs Hölzle, Norm Hutchinson, Eric Jul, Jørgen Lindskov Knudsen, Christian Laasch, Serge Lacourte, Doug Lea, Karl Lieberherr, Ole Lehrmann Madsen, Boris Magnusson, Rick Mugridge, Birger Møller-Pedersen, Jens Palsberg, Oskar Permvall, Markku Sakkinen, Michael I. Schwartzbach, Alan Snyder, Clemens Szyperski, Andrew Watson, and Alan Wills. Most participants contributed a short position

paper; they are collected in [1].

The three discussions were entitled "Classes versus Types", "Static versus Dynamic Typing", and "Type Inference". All these topics were assumed to be volatile and controversial; indeed, a broad range of diverging opinions were represented. However, much superficial disagreement seemed to be rooted in confusions about terminology. When such issues were resolved, there appeared a consensus about basic definitions and the—often incompatible—choices that one is at liberty to make. This clarification, which we hope to have described below, was the most important achievement of the workshop.

In our summary we have attempted to organize the topics and arguments into a succinct and readable format. In particular we sometimes emphasize points of agreement or divergence that were only implied at the workshop. We hope that this style will result in a coherent overview of this research area within the available spare. Simile apologies apply to the absence of references and direct quotations.

This summary is not intended to serve as an introductory survey, and it will certainly not perform that task in a satisfactory manner. We direct ourselves to the active researcher in this field.

Acknowledgement: The authors thank Andrew Black, Elspeth Cusack, Ole Lehrmann Madsen, Alan Snyder, and Peter Wegner for comments on a draft of this paper.

2 Classes versus Types

Initially, this topic seemed very controversial—in particular when specialized to subclassing versus subtyping. However, definite progress towards a consensus was made during the discussion.

2.1 The Rôle of Types

A common understanding of the rôle of types can be acieved by viewing them as predicates on objects. An object x has type T whenever x satisfies the predicate corresponding to T. This implies that, in general, an object

will have many (incomparable) types.

There is full agreement that an object is an encapsulated state, and that classes describe objects with the same implementation. Furthermore, it is clear that subclassing is a technique for reusing object descriptions (classes). There is some disagreement about the exact semantics of subclassing: which transformations on classes should be possible, and how should they be specified? Inheritance is generally considered a fundamental subclassing mechanism, but it has many, somewhat divergent, definitions.

To fully appreciate the need for types, one must consider reuse of individual objects. The *use* of an object consists of sending it messages, such as

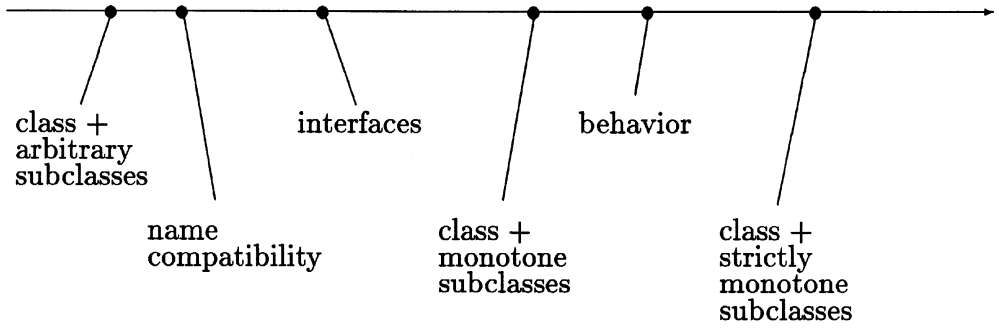
```
x.m(y)
```

Here x is the object, m is the message selector, and y is the argument, which is also an object. When the method corresponding to m was implemented, it is likely that the argument was intended to be an instance of some specific class. The *reuse* of an object is concerned with supplying other—perhaps unforeseen—arguments.

If the language is untyped, then *any* argument is legal; in short, we have unlimited possibilities for reuse. We operate, however, at our own peril. Our expectations when implementing the method m need not be adhered to, and it is a common experience that unwanted behavior may result.

Types will be needed to *restrict* the potential reuse of objects. We only want *disciplined* reuse of objects. As mentioned, the types will be predicates on objects. Formal parameters to methods will have associated types, and all actual parameters must satisfy the corresponding predicates.

There is a wide spectrum of such predicates that have been suggested. Some place emphasis on the implementation, others on the specification. Informally, they can be arranged along the following line; from left to right the predicates become *more expressive*, in the sense that they consider more and more aspects of objects.



Some explanation is required. When classes are used as types, then the predicate properly requires that the object is an instance of a particular class *or* any of its subclasses. Clearly, this definition depends on the particular choice of subclassing mechanism.

By *arbitrary* subclasses we mean that methods may be added, deleted, or redefined; this results in an almost trivial predicate, since any collection of methods corresponds to some subclass. A step up, we can require name compatibility, i.e., the existence of a particular set of named methods. The next step is to interface types, which consider not just the named methods required by the object, but also include the types of the arguments of those methods (recursively). Going up we encounter *monotone* subclasses, where we can only add methods or redefine method bodies; clearly, this will also preserve the interface. Next, we encounter the idea of also imposing restrictions on the *behavior* of the required methods, typically by specifying pre- and post-conditions; the more subtle ideas of behavior employed by various process calculi could be an alternative. Finally, our diagram shows *strictly monotone* subclasses, where methods can only be added; here, we even require that the specified behavior must have a particular implementation.

Under closer scrutiny, the indicated line would probably be discovered to have a branching structure. There is, however, at least one interpretation under which it is linear. If we partition the (imagined) collection of all objects according to whether they have the same set of types, then we observe strictly finer petitions as we move from left to right.

2.2 Subtyping

Subtyping is a relation on types—typically a partial order—such that if T_1 is a subtype of T_2, then any object of type T_1 is also an object of type T_2. This allows us to exploit the polymorphism of objects.

The exact definition of subtyping depends on the definition of type. However, there is a general soundness criterion that must be obeyed. A method implementation is protected by the types of its formal parameters. This protection is exploited by the language to grantee certain invariants about the dynamic behavior of programs e.g. the absence of certain run-time errors. A sound notion of subtyping must guarantee that sufficient protection is provided by the subtypes of the types of the formal parameters. The major notions of subtyping are listed in the following table.

When a type is:	Subtyping becomes:
class + subclasses	subclassing
name compatability	more methods
interface	conformance
behavior	weaker preconditions stronger postconditions

With classes as types, it is hardly surprising that subtyping becomes subclassing; after all, that choice is sound by definition. Note, though, that there are other implementation types besides a class + its subclasses. Finite sets of classes have also been suggested; in this context, which arises when closed programs are considered, subtyping is simply set inclusion. Even a single class can be a useful type because it fixes both the behavior and the implementation; in this case, subtyping is trivially equality.

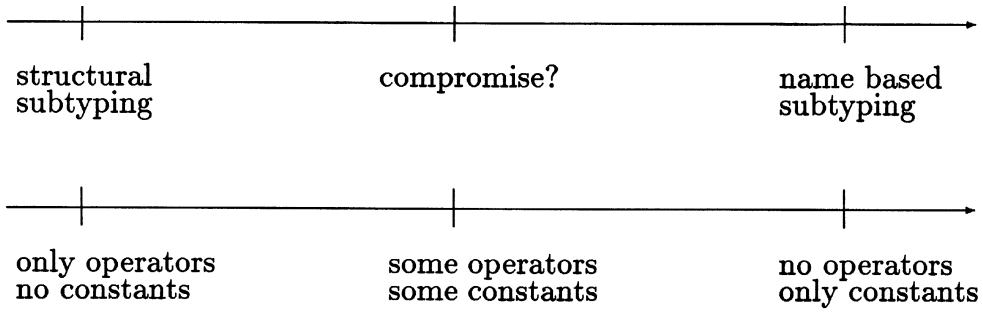
With specification types, subclassing and subtyping need not be related at all; indeed, there is no pressing reason even to have subclassing. For name compatibility, the subtype must implement more methods. For interfaces, the subtype must *conform* to the supertype—a recursive generalization of the requirement for name compatibility that usually involves the notion of *contravariance*.

Contravariance means that an argument type of a method can only be *gener-*

alized in a subclass, whereas the result type can be specialized. This ensures a statically sound type system. Some argue that contravariance is awkward, because a programmer typically wants to specialize rather than to generalize arguments. An alternative is to use *covariance*, which means that also argument types may be specialized. Such type systems are not statically sound, but can be *dynamically* sound when the compiler inserts appropriate run-time type checks, as discussed in the following section.

For behavior considerations, we must further require that a method in the subtype has a weaker pre- and a stronger post-condition; this is also a form of contravariance.

Subtyping can be structural or based on names, as indicated in the following figure.



At one extreme we find type systems in which subtype relations are always *inferred* from the structure of the types, following simple rules. At the other extreme we find type systems that require a subtype to explicitly mention a supertype in its definition, in order for the two to be related; only transitivity and reflexivity can be inferred.

These differences can be seen to correlate with another phenomenon: the presence or absence of type constructors. In languages with structural subtyping, all types are built as expressions involving a number of type constructors. The subtype rules are associated with these constructors. In languages with explicit, name based subtyping there need not be any type constructors at all; then all types must be given as constants—typically through class definitions.

It is certainly possible to envision a compromise, where a basis of type constants and explicit subtype relations can be extended by a number of type

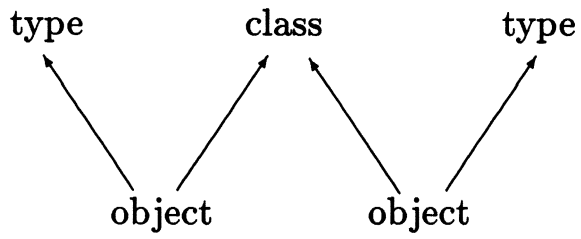
constructors and their derived subtype relations. However, extreme positions are most often adopted.

One can also obtain interesting theories by viewing class definitions as type constructors defined by the programmer; this leads to a notion of structural subclassing which captures some of the best aspects of both approaches.

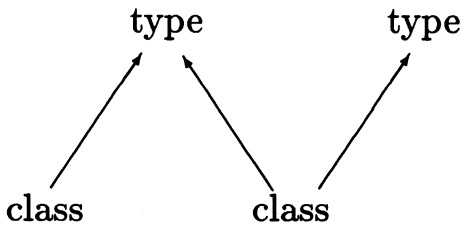
2.3 Specification Types versus Implementation Types

It is significant that the participants agreed on the described conceptual common ground for the various notions of types and subtypes in object-oriented languages. Beyond that, however, there are distinct choices to be made. The fundamental question—whether to separate classes and types—can now be rephrased as a question whether to use specification types or implementation types. Neither alternative is perfect, and it may be that both are needed simultaneously. Some of the major arguments are given below.

When we base our type system on classes, then we require that all instances of a class must have the same type. This is not the case for specification types, where class relationships need have nothing in common with type relationships. For example, even and odd integers can form different specification types; this is only possible for implementation types if they are instances of different classes. In short, the following situation can only be modeled when having separate specification types.



A major problem with implementation types is that we do not allow e.g. two different implementations of a type stack to be substitutable. Furthermore, we cannot reflect if a class implements more than one type. In short, the following picture can only be modeled when having separate specification types.



A major problem with specification types—in the form of interfaces—is that conceptually unrelated types may conform to each other. For example, a type Cowboy with methods draw, move, and shoot conforms to a type Rectangle with methods draw and move. Behavior types may seem to be an appropriate compromise; however, they may make type checking undecidable and offer several technical challenges.

It can be argued that even if classes are types, then the class Cowboy can be programmed as a subclass of Rectangle. This emphasizes that only some of the subclassing relations should be considered as subtyping relations. Some participant feel that the programmer should strive to make the class and type hierarchies coincide, even if this sometimes requires major restructuring of the program; others strongly disagree.

Emphasis on a particular application may point to a more obvious choice between specification and implementation types. In a truly distributed context, information about implementations may simply not be available, and interfaces will have the advantage. On the other hand, if code reuse is the main issue, then implementation types are clearly more appropriate.

3 Static versus Dynamic typing

There is general agreement that type information in programs serve the following four purposes:

1. readability,
2. correctness,
3. safety guarantee, and

4. efficiency.

There is little controversy about the first point: type annotations up to a point will make programs more readable. Reading a program is a static activity that can be aided only by static type information. For the remaining three points, however, there are excellent reasons to defer some typing to run-time.

The second point, correctness, has relevance for type systems that describe behavior, e.g. by specification of pre- and post-conditions. With systems of this generality even type *checking* can be undecidable. Interactive proof checkers may be the answer, but often it will be much easier to simply check assertions on run-time. This may also serve as a useful debugging mechanism. Finally, supplying proofs of all versions of a program during the development phase will almost certainly be an overwhelming task; run-time checking of assertions may be a useful compromise.

The safety guarantee, mentioned as the third point, concerns the absence of run-time errors; specifically, the error message-not-understood. The type constraints that one must impose in order to obtain such a safety guarantee are, however, too strict for some naturally arising situations.

As mentioned in the previous section, covariance may be preferred to contravariance in practice. This leads to a statically unsound type system, however, because seemingly legal arguments may in fact have a too general type. To obtain a dynamically sound type system, the compiler must insert a run-time type check to ensure that the argument is sufficiently specialized.

A similar situation arises in connection with assignments between unequal types. In most safe type systems the assignment

aSupertype := aSubtype

is type correct, whereas the converse is not. In connection with heterogeneous collections this asymmetry causes a problem. For example, we may construct a list of instances of subtypes of a type Comparable and sort them as instances of Comparable. Afterwards, we want to recover the elements of the list as instances of their original types. We find, however, that this original type information is lost.

There is no easy fix to this problem. A parameterized list type solves the problem for *homogeneous* lists, but when the collection is heterogeneous, there is no single subtype that can be used as actual parameter. The solution is to recover the lost type information dynamically. One can debate whether this should be implicit or explicit in the syntax. One solution is to allow assignments of the form

aSubtype := aSupertype

but to insert an implicit run-time check to verify that the object on the right-hand side is indeed an instance of an appropriate type. Another solution is to explicitly use a mechanism such as

view x **as** T

which yields an instance of type T if x is saliently specialized; if not, then either a run-time error may be invoked or the result may be nil.

Similar problems arise in coercion-based languages, where imperative updates cannot be statically typed without loss of type information—even if bounded parametric polymorphism is employed.

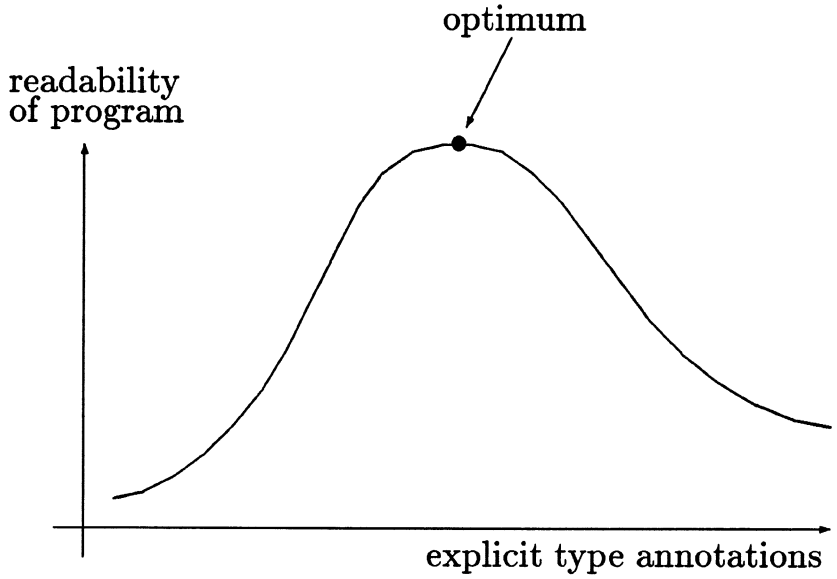
The final point, concerning efficiency, does harbor some controversy. One must distinguish between those type systems that focus on implementation (relating to classes) and those that focus on specification (relating to interfaces or behavior). It seems that specification types can contribute little to the efficiency of language implementations. The impact on efficiency of the safety guarantee—which can be supplied by both kinds of type system—is quite modest. Performance is dominated by the overhead of late binding of methods (dynamic dispatching) and by the multitude of method calls itself. Implementation types can certainly help in this respect, but equally good results seem to be possible by relying exclusively on dynamic type information. It follows that even languages relying on completely static specification typing may require dynamic implementation typing to ensure reasonable performance.

In conclusion, there is no doubt that static type information is much to be preferred—when it is available. However, realistic programming systems can

rarely stay within this bound. This can be further emphasized by considering systems that are either developed or executed in a distributed context. Here, complete type information may not exist at compile time, or may be unreliable. Such situations require dynamic typing.

4 Type Inference

The most important realization about the issue of type inference is that one does not have to make a definitive choice between explicit or implicit type information in programs. Rather, there is a continuous spectrum ranging from pure type inference to pure type checking. Furthermore, it seems reasonable that the true situation can be illustrated by the following informal diagram.



The graph shows along the x-axis the degree of explicit type annotation required, and along the y-axis a measure of the readability of the resulting programs (both measured in fictitious units). For language designs, one should look for an *optimum*, i.e., a golden compromise between explicit and implicit type information.

One endpoint of the spectrum—the completely untyped program—is a realis-

tic possibility. A completely typed program, however, has not been seriously suggested in any language. Even a very simple program would explode with type annotations, as illustrated by the following example, where type annotations are written as subscripts.

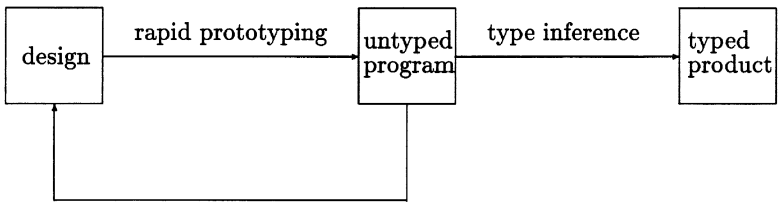
$$
\begin{array}{l}
\textbf{var } \mathsf{x},\mathsf{y}\colon \mathsf{Int} \\
\textbf{var } \mathsf{z}\colon \mathsf{Bool} \\
\vdots \\
\mathsf{z}_{\mathsf{Bool}} :=_{\mathsf{Bool}} (\mathsf{x}_{\mathsf{Int}} =_{\mathsf{Int}\times\mathsf{Int}\to\mathsf{Bool}} (\mathsf{y}_{\mathsf{Int}} +_{\mathsf{Int}\times\mathsf{Int}\to\mathsf{Int}} 1_{\mathsf{Int}})_{\mathsf{Int}})_{\mathsf{Bool}}
\end{array}
$$

Of course, nobody would require this much type annotation. Hence, all compilers will do a modicum of type inference; for example, the type of y + 1 is inferred from the types of y and 1. Overloading of operators for e.g. real and integer values is another common feature that requires some type inference.

However, some redundancy is often accepted; it is even generally advo-cated as a sound engineering principle. In particular, explicit type declarations may serve as a contract between separately developed modules.

An argument against explicit type information is related to rapid prototyping of programs. During the initial development phase it may be too cumbersome to specify all type information. Furthermore, some exploratory program approximations may not even be typable. This explains the success of untyped languages in this area.

There is general agreement, however, that the finished product should be typed—or at least typable. In other words, the kind of insight that is represented by type information should in any case be achieved by the programmer. Thus, the primary rôle of full-scale type inference can be as an important tool aiding in the transformation from prototype to product, as illustrated by the following picture.



There are two technical problems with type inference. One is that type information is generally uncomputable; hence, any sound type inference algorithm will reject some typable programs. The second problem is that a type inference algorithm is unaware of the intentions of the programmer; thus, it may inadvertently accept programs that the programmer would reject. Both problems can be relieved by combining type inference with partial type annotations.

The second problem can even be viewed as an advantage. It can be argued that in this situation the compiler is simply discovering that the written code is more general than the programmer imagined. Disagreement about this point reflects the more basic dichotomy: do types simply reflect abstract properties of programs, or do they express the subjective intentions of the programmer?

A different benefit of explicit type information is the ability to perform *code* inference from types. In simple forms this is a well-known concept; code for value assignment and deep equality can be inferred from the types. It appears that the much richer type structure of object-oriented system can allow more advanced developments in this direction.

References

[1] Jens Palsberg and Michael I. Schwartzbach. *Type Inheritance and Assignments: A collection of position papers from the ECOOP'91 W5 Workshop.* Computer Science Department, Aarhus University. PB-357, 1991.