Types, Inheritance and Assignments

– A collection of positions papers from the ECOOP'91 workshop W5 Geneva, Switzerland

Jens Palsberg
Michael I. Schwartzbach
(editors)

July 1991

Contents

Preface

This collection contains the positions of most of the 31 participants at the *ECOOP'91 W5 Workshop on "Types, Inheritance, and Assignments.* The workshop is organized in connection with the Fifth Annual European Conference on Object-Oriented Programming, July 15–19 in Geneva, Switzerland. The workshop takes place July 16, 9.00–17.30.

In addition to the 21 submitted position papers, the collection includes an invited paper by Luca Cardelli.

The focus of the workshop is on the premises, results, and aspirations of research in object-oriented type systems. In the Call-for-Participation the following issues were raised.

> The type theory of object-oriented programming is advancing rapidly. Types are required to ensure reliability and efficiency of software, and the presence of inheritance and assignments in object-oriented languages makes typing a challenging problem. This has led to a profusion of approaches, each giving important but often incompatible contributions to the theory. The workshop will seek to relate these approaches, clarify state-of-the-art, and point to major unsolved problems. We will focus on the following five questions: What are appropriate models of classes, types, subclassing, and subtyping? How can updates be typed without loss of type information? To what extent are type systems for functional languages adequate? Should classes and types be different? How can type inference be accomplished?

In view of the present position papers, some more specific questions can be

posed: Are present languages too complicated? Can static typing improve efficiency? Are types just sets of classes? Does separate compilation and concurrency require dynamic typing? What is common to the type systems in object-oriented programming-, database-, and specification languages?
—Aarhus, June 1991

Jens Palsberg   Michael I. Schwartzbach

Typed Foundations of Object-Oriented Programming

Luca Cardelli

In recent years we have seen a flourishing of ideas and techniques both in the design and in the study of typed object-oriented languages. New languages and language features are proposed at every turn, and new semantic models and semantic interpretations closely follow.

While, on one hand, one should be gratified by such richness, I cannot help feeling also a bit embarrassed. New mechanisms are justifiably proposed out of necessity, to remedy deficiencies of existing mechanisms. But, eventually, one reaches a point of diminishing returns, where convenience of additional mechanisms is overshadowed by their added complexity. How many good ideas can there really be?

This kind of exploration should eventually be replaced by consolidation, and *now* may be a good time. In this respect, I would like to strike against two common attitudes. One is the assumption that we understand existing languages well enough, so we can go ahead and create more complex ones. I think it has not been proved yet (nor disproved) that object-oriented programming as currently intended is a "good thing". It is conceivable that even basic features such as self will eventually be considered too subtle and powerful for robust software engineering (or verification), and should be abandoned. Some features will of course survive, possibly becoming more general. Consolidation does not mean oversimplification; necessary distinctions must be made, e.g. between types and classes. But do we need both prototypes and multiple inheritance at once? How can we tell when a language is powerful, as opposed to "just too complicated"?

The other objectionable attitude is of a more technical nature. The semantics of typed o-o programming has been so far explained in terms of denotational models, and here too we have seen a variety of models and an embarrassing richness of interpretation techniques. In all cases, though, a *typed* o-o language is translated into some *untyped* λ-calculus (the language of the model); a typing soundness theorem must then be proven. I think this is a rather indirect and uninformative approach, from a typing perspective, and leads to too many arbitrary choices. Many of the subtle problems we confront these days are in the typing of o-o languages (as well as in their meaning). A proof of typing soundness in a denotational model may show that the type rules of an o-o language are sound, but I don't think it shows *why* they are sound. What are the essential properties of all these models and interpretations that make the type rules sound?

The central question for me is: what is the smallest typed formal system that captures the essence of object-oriented programming? Let's call this hypothetical system *TFS*. I think one should codify the crucial properties of denotational models (or just our plain intuitions) into *TSF*, and then give meanings to o-o languages by a *type-preserving*, *subtype-preserving*, and *meaning-preserving* translation into *TFS*. If we can do this, then we will be able to say that the typing and equational rules of *TFS* capture the essence of typed o-o programming.

I have my share of responsibility for producing overcomplex formal systems, but recently I have been investigating a very simple one, with the aims explained above. This system, called $F <:$, is described in [11] and, just to show its compactness, here is the complete syntax of types:

$A, B ::=$	Types
X	type variables
Top	the supertype of all types
$A \rightarrow B$	function spaces
$\forall(X <: A)B$	bounded quantifications

I am not yet claiming that this is the "right" minimal formal system, but certainly I think it is on the right track. The first indication is that many common constructions, such as *fixed-size records*, can be encoded and their type rules can be derived. More significantly, *extensible records*, for which many complex axiomatizations have been proposed, can also be encoded. (Extensible records were investigated for their relevance to functional and imperative update.) *Record concatenation*, an always troubling subject related to multiple inheritance, can be encoded as well, following Rémy. By adding recursion and higher-order features we can also emulate *F-bounded quantification*, and we can capture all the crucial features of my (rather large) QUEST language.

What needs to be done now is to take some semi-realistic o-o language and attempt to encode it "all the way down" into $F <:$. Early attempts have proven difficult but also rewarding in terms of understanding of o-o features and their typing.

In conclusion, I argue that we should be looking for *typed semantics*, given by translations from o-o languages into small typed λ-calculi. The advantages of this approach would be that (1) the translation process "explains" the type rules of the source language in terms of more fundamental type rules of the target calculus, and (2) the target calculus, being small, has fewer, cleaner and more powerful features, and relatively simple models .

If it turns out that this translation is practically infeasible for some particular o-o language, it may mean that we have the wrong approach, but it may also mean that that language is "just too complicated".

Accessing Variables by Methods in the Conformity Typed Language Ellie

Birger Andersen

Ellie is an object-oriented programming language based on a number of new strong concepts yielding very high language flexibility in order to be generally usable. This has been obtained by allowing definitions of new types and control structures by reusing existing ones and by having a conformity-based type system. As something unique, variables (and other named attributes) are represented by objects providing methods for access of their values.

Delegation and multiple inheritance is supported by the same integrated mechanism called *interface inclusion*. Objects may also define *dynamic interfaces* that may change over time to be used for synchronization.

Furthermore, Ellie has fine-grained objects and fine-grained parallelism as an integrated and natural part of the language. Ellie has been defined in [2] and some facilities are discussed and evaluated in [4].

Variable Objects

In Ellie, variables are represented by *variable objects*, having a number of methods for accessing their values. Variable objects are implicitly created by the existence of declarations of variable names (and other named attributes).

In the creation a variable object requires a parameter object called the *qualifier* which defines that the variable may only be assigned objects conforming to the parameter object.

Variable objects and the technique of accessing variables by methods implemented in ELLIE has many advantages compared to, e.g., C++ and SMALLTALK.

- Assign and read value methods control the access to the value of a variable. This is very useful when processes may access a variable simultaneously. Synchronization mechanisms can be implemented in order to build reliable fine-grained parallel applications.
- The access methods are implicitly defined since variable objects are implicitly defined. The methods are specialized by the qualifier so that type information is available for type checks when accessing variables. Therefore, type checking assignments are like type checking parameters. Methods for dynamic conformity type checks etc. also exist.
- The variable methods may be redefined so what looks like a variable may encapsulate something else than a value, e.g., evaluation methods. This means that real variables may be substituted by objects implementing the same abstract type.
- Variables of an object may be made accessible by other objects by declaring variable methods to be part of the interface of the object. Unlike in C++, the implementation will continue to be inaccessible, i.e., it is still encapsulated. Unlike in SMALLTALK, the access methods are already defined implicitly.

Other Features Concerning Types

ELLIE is semantically and syntactically a simple language but it relies on some sophisticated ideas that all together constitute a very general language.

- First class objects defined by the fundamental and single block structured abstraction mechanism called an ELLIE *object* are used for modeling classes of traditional objects, methods, and blocks. Therefore, variables may refer to typed methods and blocks providing second order programming facilities.

- Conformity type checking and parameter polymorphism analogous to the conformity type system implemented in EMERALD [8, 57] is used for safe, efficient, and flexible typing with static/dynamic checks.

- Functional and operational methods like in EMERALD, are used for separating methods without and with side-effects. Methods are also either *future* or *non-future* methods. A future method forks a process. Such information also define the abstract type of an object.

Current Position

The implementation has shown that the idea of variable objects combined with the conformity type system can be implemented in practice [29]. Some of the powerfulness of these concepts has also been shown by examples [3]. An outstanding question is how ELLIE will perform in the real world? The language may seem too exotic to the programmers? In order to answer such questions, I plan to let a number of graduate students write some real programs.

Types and Polymorphism in Emerald

Andrew Black

The EMERALD programming language has been developed since 1984 as a tool for writing distributed subsystems and applications [8, 34]. It is statically typed, and bases its type checking on an inclusion relation called conformity. It also supports parametric polymorphism, so that it is possible for users to create types like Set.of[element]. All of the operations performed by the EMERALD type-checker at compile-time are also available to the program at run-time; the success of a compile-time check can be viewed as a license to omit the same check at run-time.

More recently, we have been working on a model for the EMERALD type system as a means of obtaining a better understanding of what EMERALD types really are. We are now able to give a type to the constant nil, find the smallest type that conforms to two given types, and type-check polymorphic self-application.

Types and Subtyping

One of our major design goals was that EMERALD objects be *implementation independent*: that objects with the same behaviour be implementable in many different ways, without the cooperation of their clients.

Implementation independence was originally motivated by the need to allow the EMERALD *compiler* to generate different implementations of an object from the same source code, depending on how the object is used. For example, if it is possible to determine by static analysis that a refeence to an object o is never exported from the object that creates o, then there is no need to provide o with the mechanism to deal with incoming remote invocations. However, we soon realised that implementation independence also allows the *programmer* to create multiple implementations of the same abstraction explicitly. For example, matrices can be implemented densely or using sparse array techniques; the interface is the same in both cases, and clients need not care how a particular object is implemented.

Consideration of the consequences of implementation independence and the encapsulation provided by object structure led us to design a type system in which types are sets of operations, not sets of values. With each operation is associated a *signature* that describes the types of its arguments and results. Binding an object of type S to a name declared to be of type T (as occurs during assignment or parameter passing) requires that S be a "subtype" of T, i.e., that objects of type S can be used where objects of type T are expected, or that S can be *substituted* for T. Substitutability means that

1. the operations of S must be a superset of the operations of T;
2. for each operation ϕ supported by T:
   (a) the results of ϕ in S must be substitutable for the corresponding results of ϕ in T, and
   (b) the arguments of ϕ in T must be substitutable for the corresponding arguments of ϕ in S

If the first condition is not met, then there will be some operation θ that may be validly invoked on an object of type T, but which is not supported by an object of type S. The second condition ensures that the first condition is met recursively for operations applied to the arguments and results. Note that part 2(b) is contravariant.

There are many relations between types that have the above properties. For example, POOL's type system has a relation $<$ that requires, in addition to the above properties, that sets of attributes associated with each type be in a subset relationship [1]. EMERALD's conformity relation $\circ\!\!>$ is (by definition) the largest relation that satisfies the substitutability conditions. It might be argued that a smaller relation leads to "safer" programs in some sense; this can be debated at length. However, it is clear that a larger relation will lead to *unsafe* programs, i.e., programs in which it is possible to invoke an operation on an object that does not support it. For us, this is motivation enough to study the conformity relation.

It is easy to model types and conformity between types when all of the operation signatures are constants. It is much harder to find a model that extends to operations that enjoy parametric polymorphism, i.e., where the type of the result of the operation depends on an argument. We have recently developed such a model; both types and operation signatures are represented as functions [9].

Classes and Inheritance

Classes and types are entirely separate in EMERALD. The type system is concerned only with the existence of a certain operation, never with its implementation, or with the implementation of the objects on which it operates. At the language level, there is no notion of class: each object is *autonomous*, by which we mean that it "owns" its own set of operations and "knows" what they are. In the *implementation* of EMERALD "class" objects are present at run-time, one per object constructor on each machine; these classes represent shared implementation detail.

Similarly, while creating new classes by inheritance from existing classes is an important programming technique, it has nothing to do with the type system. The inheritance relationship between two objects' classes is entirely independent of any conformity relationship between their types.

Updating Objects

Mutable objects are distinguished from immutable objects only in that they have update operations that change their abstract state, such as Move on a point object or Enter on a directory object. Since the arguments of the update operations are typed, there is in principle no loss of type information.

In practice, retaining all type information by static methods is infeasible: users require directories that can be used to reference any file, not just a specific type of object such a Textfile. As a consequence, the only static information that we have about the type of the result of a lookup operation on a Directory.of[File] is that it is a File. Recovering the more specific information that it is a Textfile requires a run-time check. However, since the same typing mechanisms are available at run-time and at compile-time, it is easy to integrate this check into the type system. The expression **view** f **as** Textfile has the syntactic type Textfile regardless of the syntactic type of the identifier f; if at run-time the (dynamic) type of the object bound to f does not conform to Textfile, the evaluation of the view expression will cause a checked run-time error.

Implementing dynamic type checking requires that some representation of a type be available at run-time. In EMERALD, types are objects and can therefore be manipulated in the same way as other objects.

On the Specialization of Object Behaviors

Gregor V. Bochmann

(Abstract of a full paper)
Various ordering relations have been used for defining inheritance schemes for object-oriented languages. This paper is not concerned with code sharing, which seems largely an implementation issue, but with properties that are relevant for specifications. In addition to the schemes related to subtyping and relations based on the defined operations of object classes, this paper also considers relations comparing the dynamic behavior of objects, including constraints on the results of operations, the ordering of operation executions, and possibilities for blocking. All these aspects are important for a complete characterization of the allowed behavior of object instances belonging to a given object class. The paper shows that all these aspects can be described in a unified manner, based on a set of allowed "behaviors". This leads to a unified (multiple) inheritance scheme for object-oriented languages covering all the above aspects. The use of these concepts to the design of an object-oriented specification language is also discussed.

Types and Inheritance in Object-Z

David Carrington

OBJECT-Z is a formal specification language based on the Z notation developed by Abrial and the Programming Research Group at Oxford. OBJECT-Z provides a class construct to induce additional structure on specifications, an extension that facilitates an object-oriented style (See [14, 22, 23, 24, 25] for an introduction to OBJECT-Z and several case studies).

OBJECT-Z is based on the view that each class represents a type. The primitive mathematical objects that form the basic language are not defined by classes although they could be. At the specification level, there does not seem to be any merit in distinguishing between class and type, while the advantage of simplicity is important.

OBJECT-Z takes a "liberal" view with respect to subclassing in that it does not insist that all subclasses must be substitutable for the parent class. Operations in a descendent class can be extended, redefined or removed. Thus subclasses need not be subtypes. This flexibility is very convenient in a specification context.

There is current research investigating how classes and refinement fit together. Refinement is a relation between objects that offers the ability to substitute one object for another. It provides a convenient framework for viewing the development steps involved in transforming a specification to an implementation. Refinement can be achieved both within the inheritance hierarchy and outside it. For the refinement relation to hold between a class and one of its subclasses, additional constraints must apply to the subclass to make it a behavioural subtype. Investigation of both operational and observational compatibility have been pursued to consider reactive and proactive objects.

With Object-Z, we are primarily concerned with data refinement although procedural refinement into object-oriented programming languages such as Eiffel and C++ is also being studied.

Static Type Inferencing for a Dynamically Typed Language

Bruce A. Conrad

I have been involved in the design and implementation of an object-oriented programming environment which does not use static type-checking, being a derivative mainly of SMALLTALK.

We have constructed some end-user applications using this system, and have noticed a difficulty in delivering an application with only methods which might be used during execution. It appears that some kind of static type inferencing method might allow us to automatically remove from an application those methods which could not be invoked, thus reducing the size of a delivered application.

Objects are identified by literals, variables and message sends. Variables can be either global or local (method temporaries and arguments). Even though the language does not include annotations for types, each literal refers to an object of a certain class, and, during its lifetime, each variable refers to objects of a certain class.

Object types could be identified by their class. Because of polymorphism, this simplistic view needs to be extended. Two ways we have examined are: a type is identified by a single class name meaning that the object will be an instance of that class or any one of its subclasses; or, a type is identified by a set of classes, meaning that the object will be an instance of one of the classes in the set. The potential number of types in the former case is the same as the number of classes in the system; in the latter case, it is exponential in the number of classes in the system.

Our experience has shown that the notion of type and the class inheritance hierarchy are not necessarily related. For example, our File class has methods for sequentially examining the contents of a file, and our Scanner class has a set of methods with the same functionality for examining a string of characters. However, the classes are unrelated by inheritance, except having a common superclass, Object. A compiler object has an instance variable which can be either a File or a Scanner. For this reason, we prefer to view a type as a set of classes, rather than a single class.

We would like to be able to infer the type of the result of message sends. Then we could begin with the startup method and collect a list of the methods (and classes) which might be used by a particular application during its execution.

The type of literals is known statically, by definition. The type of each global variable can be known by examining the class of the initial value of the variable and all assignments to it (many global variables are actually constants, since they never appear on the left hand side of an assignment). For local variables, the initial value is of type Undefined for method temporaries. For formal arguments, the type is the union of the types of corresponding actual arguments.

We can associate with each method a set of type signatures. Given a tuple of receiver and argument types, ex. $(t_0, t_1, \ldots, t_n)$ for a method expecting n arguments, we would like to determine the type of the resulting object. Each method selector would have a function associated with it (indicated by the italicized selector), from $T \times T^n$ to T.

The type signature for primitive methods is defined by the run-time system. For example: $\#class(x) = \mathsf{Class}$. If an object of any type is sent the message $\#class$, the result will be of type Class.

As another example, $\#new(\{\mathsf{Class}\}) = x$, where x is the receiver of $\#new$, typically a constant or particular class. In the case where the receiver of $\#new$ is "self class" in some method of an abstract superclass, the type of "self class new" is the set consisting of the subclasses of the abstract superclass, except those which redefine the method.

For methods giving access to instance variables, existing objects could be examined, as well as assignments to the instance variable. For example:

$\#superclass(\textsf{Class}) = \textsf{Class}$ or $\textsf{Undefined}$.

For other methods, the result could be computed. One way would be bottom up, starting with methods which only send primitive methods. For example: $\#upTo : (\{\textsf{File, Scanner}\}, \textsf{Character}) = \textsf{String}$.

There are simplifying considerations: first, the resulting object of many message sends is simply dropped so that its type is irrelevant; and, second, most of the method selectors in the system refer to unique methods, so that the type of the receiver can be inferred to be the class implementing the method, based on the assumption that we have a working system. The non-polymorphic selectors can be helpful in determining the type information when polymorphism applies. For example, the formal argument in a method, say a_i, has a type which needs to be determined. Arguments cannot be assigned to, so its type will be constant throughout the method. If it is sent $\#x$ and $\#y$, then it must be of type $\{\textsf{Coordinate}\}$, for these methods are defined only in the Coordinate class.

Type and Class

Rainer Fischbach

Type and class are different. Class and type are different notions. There is no simple scheme that relates classes with concrete and abstract types. Class is a syntactical construct, whereas type is a semantical concept.

Frequently—but not in all cases—a class gives rise to a concrete type and binds that type to the signature of an abstract type by means of its interface. An abstract type is not a type but rather a family of types related by a set of morphisms. In particular, it should not be confused with the union of this family, which in most cases—if it is at all a type—is a completely different type. For instance, the union of all groups is not a group!

As a consequence, concrete and abstract types are not in a subset relation! Syntactical constructs that provide for higher forms of abstraction in OO languages, like generic and deferred classes, do not denote types but rather families of types. In particular, the EIFFEL mechanism for automatic covariant redefinition of formal argument types in subclasses inhibits type formation in a deferred class. The only correct use of such class names would be as a bound identifier like "G" in the phrase "let G be a group…" that leads in the statement of a mathematical theorem.

Levels of subtyping are required. Subclassing through inheritance does not produce subtypes in all cases. On the other hand, subtype relation is not limited to the the class inheritance mechanism and could be established by alternative formal means, for instance through embeddings.

Several levels of subtyping should be distinguished. In most programming languages, types are tied to signatures only. In this setting, the subtype relation means conformance of signatures. Notwithstanding the limited expressive power of most programming languages, the notion of semantic conformance as expressed through subspecification or embedding of theories deserves some awareness.

As formal specification receives wider acceptance, a better understanding of these issues should become part of the common knowledge. A point of concern is the widespread use of inheritance in the OO community that is greatly at odds with any notion of semantic conformance.

Type checking becomes difficult. Covariant redefinition of formal argument types and hiding of features in subclasses is a potential source of type errors. An inexpensive way to deal with this situation without giving up static type checking is to constrain the use of polymorphism, as the designers of the EIFFEL derivate SATHER have done.

The only way to reconcile static type checking with the full power of polymorphism would be to calculate the set of dynamic types, that any expression that forms a target or an actual argument of a feature application could assume, and verify if this application is legal for those sets. This seems to be quite costly, but not too costly if much money or even human live is at stake in the case of a software misfunction. Rules for the calculation of those dynamic type sets and limits on the algorithmic complexity of these calculations have to be established.

On Type Inference for Object-Oriented Programming Languages

Andreas V. Hense

Types are essential for the ordered evolution of large software systems [13]. This holds for all programming language styles, be they imperative, functional, logical, or object-oriented. Type inference helps to avoid writing redundant information. In object-oriented programming, one certainly has large evolving software systems, as one of its main virtues is rapid prototyping. Therefore, types are needed for reliability, and type inference is needed for "rapidity". Two features of object-oriented programming make type checking especially hard: *late binding* and *assignments*.

Based on the work of Rémy and Wand [58, 67] we have developed a type inferencer for a small object-oriented language [31]. Our type inferencer works without type declarations. It can thus be seen as an optional test on an otherwise dynamically typed language. The object-oriented language is called O'SMALL and has the following features:

1. *state:* Objects have assignable instance variables, visible only in the declaring class (*encapsulated instance variables*). All variables must be initialized.
2. *classes:* Classes are not first-class objects.
3. *inheritance:* O'SMALL has single inheritance á la SMALLTALK [28] us-

ing pseudo variables self and super. An extension to inheritance with explicit wrappers [30] permitting the modeling of certain cases of multiple inheritance is possible.

4. *parameter passing:* Message parameters are passed and returned by reference. This is consistent with assignments involving only references (no duplication of objects).

For type checking, O'SMALL is translated into a λ-calculus with imperative features. The type checking algorithm uses so called *row variables* [67] rather than subtyping. In contrast to Wand [67] we have *principal types*, for O'SMALL does not have multiple inheritance. We have added the treatment of imperative features: assignments are restricted to their declarative scope. All occurrences of an assignable variable are collected and checked at the end of the scope.

One feature of our type checker is surprising, considering that it works on the λ-calculus level, where the notion of classes does not exist: it recognizes *abstract classes*.

Our type system is best compared to Palsberg's and Schwartzbach's type inferencer [54], because their example language is almost identical to O'SMALL. Their type inferencer is based on an entirely different technique, using subtyping and fixed-point derivation rather than unification. Common to our system is the absence of flow analysis. Their system is more flexible and can check programs that we have to refuse because of the lack of subtyping. But the increased flexibility must be paid with a quadratic expansion of code: all antecedents of a class must be expanded. In our approach every class has to be checked at most once.

One may argue that ML-type inference is DEXPTIME-hard anyway [51], so that a quadratic increase does not matter. But the worst case examples may never occur in practice, and the acceptance of a type checker in a rapid prototype system crucially depends on its performance—also on its flexibility, of course. It remains to be shown how the two type checkers' performance compares in practice.

The comparison of our type checker with the one of ML [20, 66] shows that we are more flexible in the treatment of imperative features (polymorphic references). On the other hand, O'SMALL has language restrictions that ML does not have.

The following questions are open: (1) Can our type checker be substantially generalized? (2) How severe are the restrictions due to the lack of subtyping in practice? (3) What are the advantages of row variables compared to subtyping?

Why static typing is not important for efficiency, or why you shouldn't be afraid to separate interface from implementation

Urs Hölzle

It is commonly believed that the type information provided by type declarations helps compilers to generate more efficient code. To cite from this workshop's Call for Papers: "Types are required to ensure [...] efficiency of software." At first sight it seems obvious why this is true. For example, static type information allows early binding of generic operators: when a PASCAL compiler encounters the expression i + 1, it can compile this into an integer addition or a floating-point addition, based on the declared type of i. In contrast, a LISP compiler usually cannot determine statically which operation to use since the run-time type of i is unknown.

In the remainder of this paper, I will argue that this belief does not hold for object-oriented languages, especially those which separate interface from implementation. For such languages, static type information has almost no efficiency advantages.

Why object-oriented languages are different

Encapsulation is one of the most desirable features of a programming language. Not only does it lead to more modular and maintainable programs, it is also the key to effective code reuse: strict encapsulation ensures that a procedure depends only on the abstract interface of its arguments, and thus that the procedure will work properly with any arguments which correctly implement this interface. As a result, any particular piece of functionality has to be written only once: there is no source-code redundancy.

Most of today's object-oriented languages do not naturally provide true encapsulation (but in some, it can be simulated). For example, C++ allows direct access to instance variables of an object, thus exposing part of its implementation. More importantly, in most languages a subtype must inherit the format of its supertype (i.e., the subtype can only add instance variables but cannot remove them or replace them with functions). Since this representation inheritance is not implied by the mathematical subtyping relationship, interface and implementation are not properly separated in such languages. I will call this form of types *representation types* (as opposed to interface types). Most of today's object-oriented languages have *representation types*; notable exceptions are POOL and TRELLIS/OWL. SELF has full encapsulation but no static typing.

Object-oriented languages achieve encapsulation through the combination of two features: subtyping and dynamic dispatch. Both features have a profound impact of the value of static type information for code generation:

Subtyping dilutes the information content of type declarations: the declaration `v: T` no longer asserts "`v` contains an object of type `T`" but only "`v` contains an object of type `T` or any subtype of `T`."

Dynamic dispatch makes it impossible in general to statically bind a particular function invocation `v.func()` to a specific implementation. By definition, the function actually invoked at run-time depends on the exact type of `v`. Using static type information, the compiler can check the validity of the invocation (i.e., that no "message not understood" error will occur at run-time), but it cannot determine the exact function being called.

Efficiency and static typing

As outlined in the previous section, in an object-oriented language, objects can only be manipulated by invoking functions defined in their interface, and every such function application `v.func()` is (conceptually) a dynamically-dispatched call. Thus, the call frequency of any program will be extremely high since every operation, no matter how trivial, is dynamically-dispatched.

In fact, calls will be so frequent that any implementation which actually performs them will be unacceptably slow, no matter how efficient the method dispatch is. An example from SELF (which provides full encapsulation) will illustrate this claim: if every function application is compiled into a dynamically-dispatched call, programs run up to several hundred times slower than their C counterparts. A simple calculation shows that the number of calls performed is so high that the programs would still run several times slower than C even if all calls were ideally fast (2 cycles/call).

Static type information can eliminate only a small fraction of these dynamically- dispatched calls.[1] Thus, any statically-typed language with interface types would suffer from the same problems, even though the dispatching speed might be better. But if this is true, how can "good" languages (namely those with interface types) ever be practical? The answer is simple: the compiler must be able to optimize away most of the calls. Optimization techniques which can eliminate many calls are used in the SELF compiler (see e.g. [15, 16, 32]) and could be adapted to statically-typed languages as well [37]. In the resulting code, most calls are inlined so that dispatching speed is no longer crucial, and statically-typed languages hold no significant performance advantage over dynamically-typed languages.

The length restrictions of this paper do not allow a discussion of particular optimization techniques. However, the following observation may show why the claim is plausible: the generated code contains (relatively infrequent) type tests which test for particular *implementation* types (not interface types!). These tests (or equivalently, dispatches) "guard" sequences of code which are specialized for the particular implementation types; in these code se-

[1]The only calls that could be optimized at link time are calls where the receiver is of a type which has only one implementation and no subtypes. In other words, this is the only situation where the compiler can statically determine the implementation type corresponding to the interface type.

quences, the implementation type of every operand is known. Since type declarations only provide interface types (not implementation types), these type tests cannot be eliminated by the type information obtained from type declarations.[2]

The point I want to make is that any object-oriented language with a type system separating interface from implementation will lead to implementation challenges which are very similar to those found in the implementation of SELF. To achieve good performance, compilers will have to rely heavily on inlining. To inline a call, the implementation type of its receiver must be known. Unfortunately, this implementation type *cannot* in general be computed from the program text alone, and thus statically-typed object-oriented languages suddenly find themselves on equal footing with dynamically-typed languages like SELF.

Contrary to popular belief, the relatively good efficiency of some statically-typed object-oriented languages (such as C++) is *not* the result of static typing per se but the result of not providing true encapsulation: types are representation types, not interface types. Programs which actually try to use fully abstract data types are much slower because the compiler technology used by most compilers is inadequate for this case. Typically, such languages also contain some non-OO types such as Integer which have a fixed representation and are not part of the normal type hierarchy. This limits code reusq for example, it is not possible to insert integers into a collection of “comparable” objects even though integers implement the comparison protocol.

Conclusion

In an object-oriented language with true interface types, dynamically-dispatched calls will be so frequent that any implementation which actually performs the calls will be unbearably slow, no matter how efficient the dispatch. The only currently known way to achieve good performance is to use optimization techniques similar to those employed by the SELF compiler, and the code generated by such techniques can hardly be improved by static type

[2]Some of those tests could be eliminated at link-time. However, the performance impact is likely to be small since type tests represent a small fraction of execution time (if they don’t, the compiler didn’t do a good enough job anyway).

information.

Thus, efficiency should not be a major motivation to include static typing in a new object-oriented language. As a corollary, language designers who want their languages to have a clean separation between interfaces and implementations need not despair: it is possible to implement languages with “clean” type systems efficiently.

Types vs. Classes, and Why We Need Both

Norman C. Hutchinson

Historically, types have been required to serve two purposes:

- Classification of the entities involved in a computation, and
- Providing "representation independence"; ensuring that the meaning of a program is not dependent on the representations chosen for its values.

If we throw away all of the baggage that the phrases "object-oriented" and "object-based" have accumulated over the last decade, we can see that the fundamental advantage that systems that support objects have over systems that do not is encapsulation. That is, a system that supports objects requires the grouping of data and operations and guarantees that only those operations defined with the data will be allowed access to the data. The encapsulation of objects provides exactly the "representation independence" mentioned above; it ensures that only code that understands the representation used for data will be allowed access to that data.

Objects and types

Accepting the object-oriented philosophy allows us to rethink the question of what we want from our type systems. We already have a mechanism for enforcing encapsulation, what we need is a mechanism for the classification of objects. There are two major forms of classification that we might desire:

- Classification based on implementation. The class systems that have evolved since SIMULA address this need very nicely. One can define a subclass of an existing class as a refinement: either extending or modifying the behaviour of the superclass.

  Such a classification scheme is of interest to the programmer of a collection of classes because it allows her to reuse code, ensure that objects behave in a consistent way, etc. It is also of interest to the compiler writer because the information about how objects are implemented can be exploited to generate smaller objects and faster code.

- Classification based on the abstract invocation protocol implemented by the object. By this I mean that each "client" of an object expects the object to implement a particular collection of operations, and any supplied object that implements all of the required operations meets (at least syntactically) the requirements imposed by that client.[3]

  Example of such requirements abound. A window manager expects a particular protocol from each window under its control (move, resize, refresh, terminate). A file system expects its directories to implement add, lookup, delete, and list.

  One can simulate this in a traditional object-oriented system by creating abstract superclasses that define "dummy" implementations of the operations and then subclassing to get each of the various implementations. There are at least two important problems with this approach:

  - You must have the insight to do this in advance of the need, since adding superclasses to existing objects is not generally possible.
  - I believe that this kind of classification is fundamental, and we must directly address the need rather than simulating it using mechanisms that were designed to solve a different problem.

[3]We could strengthen this form of classification by requiring that the object's *semantics* appropriately satisfy the demands of the client. While this is obviously desirable, I believe this to be outside of the scope of type systems.

Position

I believe that in order to fulfill their full potential, object-oriented systems must address both of these forms of classification. I therefore believe that we need to be talking about two notions of typing for object-oriented languages. I therefore believe that *class*, which has historically referred to classification based on implementation should continue to address this need, and that *type* should be used for classification at the abstract level, separate from implementation.

There are a number of issues that must be addressed by further research.

Subclass vs. subtype Does creating a subclass imply that it must be or should be a sub-type? Without additional restriction, a subclass may not be a sub-type since the subclass may redefine the types of arguments or results to an operation. Whether languages should force a subclass to also be a subtype is not so clear.

Type inference Type inference can be done at both levels, for different purposes. Type inference at the abstract level can free the programmer from the tedium of specifying all the type information. Type inference at the concrete level provides the compiler with additional information to aid in optimization.

Implementation If typing is done at an abstract level, then the compiler gets no information (in general) about the implementations of the objects that are being manipulated. How can one efficiently implement method lookup under these circumstances? The methods used in untyped languages can surely be applied, but can one approach the efficiency of the single level of indirection achievable in languages where typing is based on classes?

The Emerald programming language has been exploring these notions for the past several years, and has partial answers to some of the questions, but much more work needs to be done.

Types and Classes in Cocoon

Christian Laasch and Marc H. Scholl

Our primary goal in the COCOON project is to integrate the modeling facilities of object-oriented data models with a strongly-typed set-oriented extension of relational algebra that allows optimization of processing strategy. So we developed an (object/function) model that is sketched in the next section, it separates types from classes. Afterwards we briefly describe our generic query and update operations that allow static type checking.

Cocoon - An Object-Oriented Data Model

The COCOON model as described in [61, 62] consists of objects and functions (see also [69, 21]), but separates types, that include all compile-time information from classes. It is a core object model, meaning that we focus on the essential ingredients necessary to define a set-oriented query and update language.

Objects are instances of either predefined types (e.g. bool, real) or abstract types.

Abstract types are denoted by a set of function labels (in square brackets), e.g. Person == [name, age, sex]. Naming types is simply meant as an abbreviation.

Functions are the only way to retrieve and change the encapsulated properties of objects. They are described by a name and signature, they are the interface operations of type instances. The implementation is specified separately. We use the term *functions* in the general sense including *retrieval functions* as well as *methods*, that is, functions with side-effects. Besides functions we also use 'set' as type constructor.

Subtyping. The subtyping relation ($\preceq$) between abstract type expressions is defined by the inclusion of the function-label sets:

$$[\ldots f \ldots]_1 \preceq [\ldots f \ldots]_2 :\Longleftrightarrow \{\ldots f \ldots\}_1 \supseteq \{\ldots f \ldots\}_2$$

Therefore objects can be instances of several types. The subtype relation of constructed types (τ) can be inferred by following subtyping rules (the consequence is valid, if the premise can be deduced):

$$[\text{SETS}]\frac{\tau_1 \preceq \tau_2}{\{\tau_1\} \preceq \{\tau_2\}} \quad [\text{FCNS}]\frac{\tau_1^{dom} \preceq \tau_2^{dom}, \tau_1^{rng} \succeq \tau_2^{rng}}{\tau_2^{dom} \rightarrow \tau_2^{rng} \preceq \tau_1^{dom} \rightarrow \tau_1^{rng}}$$

Therefore types are regarded as ideals and the set inclusion between atomic types corresponds to the inclusion of their function sets.

Classes are a special kind of abstract objects: they represent (typed) sets of objects. A class C itself is an instance of the meta type 'class' that associates a type, the $member\_type(C)$, to all objects in the set $extent(C)$. The extent of a class includes all objects that are instances of the member type and fulfill the necessary and sufficient class predicate ($suffp(C)$). Due to the separation of types and classes, there may be any number of classes for a particular type: for instance, more than one as the result of selection operations, see below, or none, if we are not interested in maintaining an explicit extent of that type.

Subclassing. A partial reflexive order between classes ($\sqsubseteq$) is defined as follows:

$$subc \sqsubseteq supc :\Longleftrightarrow$$
$$(member\_type(subc) \preceq member_type(supc)) \wedge (suffp(subc) \Rightarrow suffp(supc))$$

Notice that the explicit separation of subtype and subset relationship alleviates the problem of deciding whether a class c_1 is a subclass of c_2 or not, because there is no need to check predicate subsumption (which is in general undecidable), if ($member\_type(c_1) \preceq member\_type(c_2)$ is not true. We use an incomplete decision procedure for positioning a class in a class lattice (resp. testing the predicate subsumption) guaranteeing that the determined position is correct. However, there may be cases where the class could have been placed further down the lattice. There-fore, our notion of subclassing meets the common sense, but divides two independent relationships.

Generic Operations

We use a set-oriented algebra, where the inputs and outputs of the operations are sets of objects. Hence, query operators can be applied to extents of classes, set-valued function results, query results, or set variables. Even though classes represent polymorphic sets, type checking of our language always refers to the unique member type of the involved sets. As query operations we provide set operations (union, intersect), selection of objects (select), and two type changing operators (project, extend). The pick operation chooses one object of a set. The effects of each operator are defined sepakately for type and extent. (Only union, intersect, and pick have an effect on both.)

Another argument for separating subtype and subset relationship among classes is the classification of query results (needed for views definitions, for example) [60]. Classification of results, that uses the object preserving semantics of our operations, improves clarity of the class lattice, and can be used for optimization. Already a single combination of select and project—as usual in relational algebra (resp. in each SQL-statement)—results in a subset and a supertype. Therefore the input is neither a subclass of the result nor vice-versa: we cannot connect the result to the input in a mixed class hierarchy. If, however, we separate the two concepts, two relationships hold—but in opposite direction. Therefore it is possible to position the result type and set close to the input counterparts in the two lattices.

Besides query operations we provide a set of generally applicable *generic update operations* that can also be used to define *type-specific* update operations. Included are operations to assign values to variables, classes, and functions and also for object evolution, i.e., besides creating and deleting objects also adding and removing types to/from them.

Extending the C++ Type System to support Annotations

Doug Lea

Current work by myself and colleagues continues exploration of type systems that can support predicate-based extensions required in order to directly integrate OO formal methods into OO languages. Much of the framework was presented in a draft description of ANNOTATED C++ (A++) [17]. A++ is a superset of C++ enabling programmers to embed specifications via declarative constraints within C++ classes and functions.

General features of our evolving approach with respect to OO type issues include constructs that are incompatible with the underlying C++ type system, but are laid on top of C++ in a way that preserves much the class structure, if not the type structure, of the language. These include:

1. Separation of subtyping and subclassing, in order to remove issues of inheritance and code reuse from those of behavioral guarantees.

2. A contravariance- and conformance-based type system similar to that of EMERALD.

3. Integration of predicative types (behavioral constraints) with standard subtypes. Subtypes may be defined by adding constraints (predicates) in addition to adding or redeclaring methods.

4. Integration of type-checking and constraint verification, in part by relinquishing static checkability guarantees.

5. Constructs that allow programmers to state that an object may *change* the type(s) it conforms to as the result of state changes. This may be seen as a generalization of the assignment issue in OO programming.

Our work is very much applied, and oriented toward construction of a usable annotation system. We are interested in determining the relation between this system and other type models for OO languages.

Experience with Types and Classes in the Guide Language

Emmanuel Lenormand and Michel Riveill

As part of the Esprit COMANDOS project, the GUIDE project investigates a distributed operating system architecture, which could be used as a basis for large scale applications, e.g. software development environments or advanced document processing systems. Such applications involve many components organized in complex structures, persistent data and data sharing. In this light, object-orientation has been chosen, for its ability to fulfill these requirements [26]. A high-level language support is also needed to ensure maintainability and ability to evolve for the system. Thus, in order to provide a better integration of the system and applications, an object-oriented programming language has been designed, which is dedicated to the expression of distributed applications. This language, which is also called GUIDE [35, 19], presents some characteristics concerning its types and classes, which we are going to discuss.

Types and classes in Guide

The types/classes system of GUIDE presents two main characteristics: first, the hierarchies of classes and types are separated, and second, the inheritance mechanism is constrained by conformance rules.

Two hierarchies. As in modular languages, it has been decided to separate, in the GUIDE language, the interfaces of the abstract structures which represent the types, and their implementations, which represent the classes. This choice involves different interesting abilities, among which facilities for *programming in the large* (provide to the programmer the specifications of the modules he wants to use, without having to deal with implementation details) and *modularity.* Modularity is enhanced by this separation since information remains hidden to the programmer, who only accesses the features defined in the types. While these advantages may still be present in some object-oriented languages (such as EIFFEL) which do not have separate definitions of types and classes, the separation of these notions provides additional gains: ability to define several different implementations of a type -this provides a great flexibility, at the possible expense of static binding- and conceptual clarification.

Conformant inheritance. The type system of GUIDE provides conformance rules, which are the basis of the static type checking of GUIDE programs. These rules are the classical ("contravariant") ones as they are defined in [12], and they are respected over the type hierarchy. Thus, a type which inherits from another one, must conform to it. This implies a conformant inheritance mechanism for the type hierarchy. Then the choice has been made, to extend this mechanism to the class hierarchy. Typically, a class implements a type and the condition which must be verified for two classes to be in a valid inheritance relation can be expressed as follows: if class A implements type TA and class B implements type TB, B may inherit from A if and only if TB conforms to TA.

As pointed out in the last paragraph, the separation between types and classes does not forbid them to keep related in some way. The link which exists can be called the implementation link and defined as follows:

A class implements one and only one type. A type can be unimplemented, or implemented by one or more classes, in diflerent ways.

So, a relationship between the types and classes graphs can be derived. The fact that classes inheritance obeys to conformance rules strengthens this correspondence, as each link between a class and its subclass corresponds to a conformance link between the types they implements.

Further work on the GUIDE language is now focused on multiple inheritance.

Indeed, GUIDE supports only simple inheritance, and, in the perspective of a possible extension, we wonder particularly how its model would react to a multiple inheritance mechanism, for both types and classes, and what model is suitable for this mechanism. An other point concerns the actual separation of type and class hierarchies; the class inheritance mechanism respect conformance rules, whereas there is no theoretical reason for this [18], and it could be interesting to forget this restriction. The question is: to what extent is it interesting?

Experience with the Guide language.

The GUIDE language is used by programmers in Bull-IMAG laboratory and in several locations in Europe, particularly members of the COMANDOS project, so the features mentioned above have been tested and thoroughly evaluated. Indeed, over 100,000 lines of GUIDE source code have been written, in various application programs.

Concerning the separation of the hierarchies, the double declaration of methods in a type and in the class which implements it, seems to be quite redundant for the programmer. To that extent, appropriate editorial tools would be appreciated. Yet, this little drawback should not hide the gain the separation of type and class hierarchies generates.

An important feature which is allowed by this choice is declaring types as access filters, i.e., introducing some control on access to types, as in the following example.

```
TYPE ChanIn IS
   METHOD input;
END ChanIn.

TYPE Chan SUBTYPE OF ChanIn IS
   METHOD output;
END Chan.

CLASS ClassChan IMPLEMENTS Chan IS
   METHOD input; // implementation of input
```

```
    METHOD output; // implementation of output
END ClassChan.

canal: REF ChanIn;
canal:=ClassChan.New // canal is implemented by class ClassChan
canal.input; // valid instruction
canal.output; // illegal instruction - output is not part of the ChanIn type
```

The possibility of declaring several implementations for a type has also been used and appreciated, in order to take in account some kind of heterogeneity, as shown below.

```
TYPE Window IS
    height: Integer;
    width: Integer;
    METHOD resize(IN h,w: Integer); END Window;
```

```
CLASS MyWindow
IMPLEMENTS Window IS
    CONST hmax: Integer=600;
    CONST wmax: Integer=400;
    METHOD resize(IN h,w:INTEGER);
        BEGIN
          IF (h<=hmax) THEN
            height:=h;
        END;
        IF (w<=wmax) THEN
          width:=w
        END;
    END resize;
END MyWindow.
```

```
CLASS YourWindow
IMPLEMENTS Window IS
    CONST hmax: Integer=500;
    CONST wmax: Integer=500;
    METHOD resize(IN h,w:INTEGER);
        BEGIN
          IF (height+h<=hmax) THEN
               height:=height+h;
        END;
        IF (width+w<=wmax) THEN
          width:=width+w;
          END;
      END resize;
   END YourWindow.
```

Then you can choose the implementation you want for a Window variable.

```
       window: REF Window;
       window:=MyWindow.New; or window:=YourWindow.New;
```

The respect of the conformance rule along the type hierarchy is necessary to ensure a useful and simple static type checking. The extension of this rule to the classes inheritance is quite natural from the programmer's point of view, since in most cases the class hierarchy copies exactly the type hierarchy. Yet, as mentioned above, there is apparently no reason why it should be so, and a less restricted class inheritance mechanism should also be convenient, as it does not disallow what GUIDE provides for the moment.

The Demeter Model for Classes and Types

Karl Lieberherr

I focus on the following question: What are appropriate models of classes, types, subclassing, and subtyping? Instead of using the term "type", I use the term "alternation class". Therefore I talk only about classes in the following.

An appropriate model for classes should satisfy the following conditions:

1. A set of classes efficiently defines a set of legal objects.

   This rule is important for the debugging of object-oriented data models. It allows to check whether an object can be "expressed" by a given set of classes. By efficient we mean that an object can be checked for legality by an algorithm of low polynomial complexity.

2. A set of legal objects efficiently defines a set of classes.

   This rule just expresses the intuition that classes are natural abstractions of objects and therefore classes should be computable efficiently from a representative set of objects. Efficiently again means that the problem is solvable by an algorithm of low polynomial complexity.

3. Objects have a succinct description as sentences which contain essentially only information about "primitive" objects.

   This rule makes sure that objects can be easily described for debugging the structure of object-oriented data models and of programs.

4. The model allows object-oriented programming with graphs by expressing a group of collaborating classes as paths in a class graph and by propagating code to the classes along the paths.

Our experience indicates that the above properties are important and we have invented the DEMETER model which satisfies all of them. The DEMETER model is based on a mathematical structure called a class dictionary graph. A class dictionary graph defines a set of legal objects through relationships between construction and alternation classes. Construction classes are instantiable classes which are used to create objects and alternation classes define abstract classes which define disjoint unions of construction classes. A class dictionary graph needs to satisfy a structural specification and two simple axioms: the Cycle-Free Alternation Axiom and the Unique Label Axiom.

To allow short descriptions for objects, class dictionary graphs are extended with terminals to define languages. A class dictionary graph with terminals is called a class dictionary. A printing procedure of a few lines defines how objects are printed as sentences. The set of all legal tree objects in their printed form is the language defined by a class dictionary. To allow a fast transformation of a sentence into any object, a class dictionary needs to satisfy also the Bad Cycle Axiom and two LL(1) rules. Under those conditions, the printing function is a bijection between objects and sentences and it has an inverse which is naturally called a parsing function. The parsing function is easily implemented by a recursive-descent parser and it is heavily used for debugging the structural aspects of object-oriented data models.

Once a class dictionary is debugged, we proceed by defining functions for it. A function with the same name is typically defined for a group of collaborating classes. We define such a group by a propagation pattern which specifies several paths in the class dictionary. Each class on a path gets a function with a specified interface propagated to it; also, a default body is provided which can be overridden by a user-defined function. Programming with class dictionaries and propagation patterns shortens many programs and has other significant advantages, e.g. resilience to change, over the traditional approach. Propagation patterns provide tool support for the Law of DEMETER.

The efficient abstraction of classes from a set of object examples is easily accomplished in our model. The details are described in [6, 40].

The papers [38, 46, 45, 43, 44, 47, 42, 40, 41, 39, 6] contain more information on the DEMETER model.

Classes as First-Class Types

Ole Lehrmann Madsen and Birger Møller-Pedersen

This position paper presents the notion of classes and types in the Beta [36] programming language which is based on the Scandinavian approach to object-orientation. It is argued that the type system of a language should be based upon the class/subclass mechanism.

Classes intended for modeling classification hierarchies

The point of view taken in the Scandinavian approach to object-oriented programming is that classes are intended to model concepts in the application domain. This leads to the definition of the subclassing mechanism as a means to represent classification hierarchies corresponding to generalization/ specialization. Inheritance of properties, and not just of code, is the main motivation for subclassing. Viewed as a modeling mechanism, subclassing must define a hierarchical type system.

Representing application concepts by means of a separate notion of types (or designing class language mechanisms from a type point of view) will imply that some desirable properties are not adequately modeled. If the only allowed signature of a type or class is that of a set of operations, while instance variables are only for implementation purposes, then the well-known apartment example would be difficult to accomplish. In Beta, properties like that of having part-objects representing the fact that apartments have parts like kitchen, bathroom, etc. are regarded as properties associated with the class Apartment:

```
Apartment: CLASS
      (#
      theBathroom: @ Bathroom;
      ...
   #)
```

where Kitchen and Bathroom are names of classes.

Interface and Implementation of Classes

The fact that the Beta approach is to use the class/subclass hierarchy as a type/subtype hierarchy, does not imply we do not want to distinguish between *interface* and *implementation* of a class. The language has a separate *fragment* mechanism for that, illustrated by a simple example:

```
Stack: CLASS
   (# rep: @<<SLOT rep: ObjectDescriptor>>
      push: PROC
         (# E: @ Integer
            enter E
            <<SLOT Push: DoPart>>
         #);
      ...
   (#
```

The example shows a fragment defining the visible parts of a class stack. The slots may be filled with the representation of the stack and with the implementation of the operations, and different fillings will given different implementations. The fragment as showed above corresponds to signatures in other languages. The following fragment defines the implementation:

```
ORIGIN 'Stack'
```

```
—Rep:ObjectDescriptor—
   (# S: [100] @ Integer; top: @ Integer #)
—Push:DoPart—
   (#
   do E->rep.S[rep.top+1->rep.top]
   (#
   ...
```

Strong Typing, Qualified References, and Assignment

Typing in languages that use classes as types are closely associated with *typing of object references* and the use of these types in *access of attributes* and rules for *reference assignment*. These languages provide a compromise between weak and strong typing since not all operations on an object can be inferred from the typing of the reference.

Most strongly typed languages rely on a combination of compile-time and run-time type checking [50]. BETA is an example of such languages. A large class of type errors are caught at compile-time, whereas others are left to run-time checks. Consider the following class hierarchy:

```
Vehicle: class (# ... #);
Bus: class Vehicle (# ... #);
Truck: class Vehicle (# ... #);
Car: class Vehicle (# ... #);

aVehicle: ^ Vehicle;
aBus: ^ Bus;
aTruck: ^ Truck;
```

The reference aVehicle is typed by Vehicle. This implies that aVehicle may denote instances of class Vehicle and instances of subclasses of Vehicle.

The cost for typed references is a run-time check that will take place at some cases of *reference assignment:*

```
new Bus[] -> aBus[];    {1}
aBus[] -> aVehicle[];   {2}
aVehicle[] -> aBus[];   {3}
aTruck[] -> aBus[];     {4}
```

The assignments in {1} and {2} are obviously valid. The assignment in {3} is legal if aVehicle denotes an instance of Bus. In general this may not be detected at compile time. This implies that a run-time type check will be performed in this case. The assignment in {4} is illegal, since it is not possible for aTruck to denote a Bus object.

Most assignments are as in {2} (or types are equal) and no run-time checking is needed. The ability to weaken the type information on an object as in {3} is very usable in order to write general code like queue and list manipulation etc. In the example above one would typically use the following views: as an element in a queue, as a Vehicle, and as a Bus or Truck.

In most languages *value assignment* is defined as a pure copying of bits. In this way the state of one object may be forced upon another object. Often different object states may denote the same abstract value. It is therefore not always desirable to define the semantics of value assignment as a bitwise copying. The situation is even worse when considering *equality*. Here a bitwise comparison of two objects may not correspond to equality of the abstract values represented by the two objects.

In most work on hierarchical type-systems the distinction between reference semantics and value semantics of assignment and equality does not seem to be explicit. In relation to object-oriented programming this distinction is, however, crucial.

Virtual Classes

In Beta classes may be parameterized by other classes by using virtual classes [49]. Beta has covariance of arguments to procedures in subclasses like Eiffel, but as opposed to Trellis/Owl. Since Beta supports subtype substitutability, a run-time check of arguments is performed in the case of covariance. This run-time check is similar to the run-time check performed when assigned a less qualified reference to a more qualified reference as described in the previous section. The amount of run-time checking may be limited by using type exact references and so-called final virtual bindings.

Summary

The following summary made by Pierre America at TOOLS'91 is useful:

> It is only possible to obtain at most 2 of the following 3 properties:
>
> 1. Subtype substitutability
> 2. Covariance
> 3. Static typing

BETA has given up static typing for a limited amount of run-time checking.

Adjusting the Type-Knob

Boris Magnusson

In this paper we describe a programming style that can be used to get SMALLTALK like type-less programming also in strongly typed languages like SIMULA, BETA, C++ and EIFFEL. The strong typing facilities in these languages can then be utilized as the abstractions and class hierarchy falls into place. This style of programming supports migration of applications from rapid prototypes to type-checked products.

Developing prototypes versus products

The relative freedom of type-less programming languages is often praised by programmers developing so called rapid prototypes. The ability to change, re-work and try a program although not all the abstractions and class hierarchies have been worked out has its benefits. Small changes and additional functionality can often quickly be implemented and tried out although the class hierarchy might not yet be consistent. If the functionality is not found valuable one can discard the code without much loss of effort. On the other hand, if the code is found valuable more work can then be spent to integrate it in a consistent way. Systems developed this way does, however, rarely qualify as production systems. The potential risk of such a system failing with the message "Message not understood", or some other information with the same meaning, can not be neglected. In the process of converting the system to production quality a type-less programming language offers no help in finding type errors.

In contrast to type-less languages we have languages with type rules that can be checked by a compiler. In the field of object oriented programming languages strong typing means that references are qualified, e.g. restricted to reference only object of a certain class and its subclasses. Building applications in such a system often includes repeated modifications and recompilations because of type errors in the program. This is sometimes found tedious since the programmer "knows" that the program will not actually come in a situation where the error will show. The work to satisfy the compiler in such cases can be found unnecessary when the goal is to make an experiment and the code will be thrown away in the end. On the other hand when the compiler is satisfied there are certain errors that can not occur.

The merits of these two approaches to program development and language design has often resulted in that systems are developed twice—once to try out the ideas, get some feedback and get a first sketch of the architecture of the system—and then over again to get a robust implementation. In this paper we describe how this transition can be made in incremental steps within the same language. Parts of the program can be expressed in a type checked manner while other parts remain experimental. Experimental parts can also be added later. In this paper we will show how untyped SMALLTALK like programming can be achieved in a strongly typed language.

The type-less, prototype solution

To illustrate the difference in approach between a language with and without typed references we like to use a small example: Consider a system with the following classes:

```
CLASS Graphical
      virtual: Draw, Move
CLASS Line
      operations: Draw, Move
CLASS Rectangle
      operations: Draw, Move
CLASS Circle
      operations: Draw, Move
```

```
CLASS Cowboy
      operations: Draw, Move, Shoot
```

This can be easily expressed in any O-O language with some syntactic variations. Line, Rectangle, and Circle are all classes with the same set of graphical operations while Cowboy is an unrelated class.

In a language without typed pointers we can declare one pointer and let it reference an object of any of the above classes. We can also call an operation of the class (send the object a message):

```
pointer P;
if ... then P:- new Cowboy else p:- new Circle;
P.Shoot;
P.Draw
```

Two different kind of errors can occur here. In the first statement case the object P might point to a Cowboy in which case everything is OK but if it is say a Circle we will get a “Message not understood” during execution. In the second case the programmers assumption might be that all the objects P will point to are graphical and Draw will thus have a certain meaning. If there, however, happens to be a Cowboy object the result will be a surprise.

The qualified references, the production version

In a language with qualified references we can declare pointers that are dedicated to point to objects of a certain class or its subclasses and the problems above are caught during compilation. We have to introduce a special class, “Graphical”, in order to express the relation between three of the classes.

```
CLASS Graphical
      virtual: Draw, Move
Graphical CLASS Line
      operations: Draw, Move
```

```
Graphical CLASS Rectangle
        operations: Draw, Move
Graphical CLASS Circle
        operations: Draw, Move
CLASS Cowboy
        operations: Draw, Move, Shoot

pointer(Graphical) aG;
pointer(Cowboy) aB;
aG :- new Circle;
aB :- new Cowboy;
aG.Draw     —is guaranteed to Draw of a subclass of Graphical
aG.Shoot    —will be reported as an error by the compiler
aG :- aB    —will also be refused by the compiler
```

In a typed language the superclass Graphical will be required to declare the virtual operations Draw and Move in order to use them with a pointer qualified only as Graphical. The operations are then implemented in different ways in the three sub classes as shown above. Graphical is not intended to be instantiated and such classes are often called "abstract", Note that the sole reason to introduce this class is to express this commonality between classes, there is no code-sharing going on here.

Type-less programming in a typed language

Our first step towards a type-adjustable programming style is to show how you can actually program type-less in a typed system. The rules are very simple: (1) Make all classes subclasses (direct or indirect) of one class, say AnyType. (2) D ec are 1 all virtual operations in class AnyType. (3) Declare all pointers as pointers to AnyType objects.

This will result in that all pointers can reference any particular object and that all operations are (from the compilers point of view) available in all classes. The above example will become:

```
CLASS AnyType
```

```
        virtual: Draw, Move, Shoot
AnyType CLASS Line
        operations: Draw, Move
AnyType CLASS Rectangle
        operations: Draw, Move
AnyType CLASS Circle
        operations: Draw, Move
AnyType CLASS Cowboy
        operations: Draw, Move, Shoot

pointer(AnyType) P;
if ... the P :- new Circle else P :- new Cowboy;
P.Shoot;
P.Draw
```

Here we can experience the same kind of errors (and thus achieve the same freedom) as in a genuinely type-less language. P.Shoot may result in a execution error like "Message not understood" and P-Draw may be a surprise if P points to a Cowboy object which is now perfectly legal.

Using this simple programming style lets us mimic a SMALLTALK like program development method. Not all issues have been addressed in the above small example. Class hierarchy—in the above example all classes are subclass of AnyType. In a larger example there will be a deeper class hierarchy in order to achieve code reuse. We will address this below. Instance variables—they are not part of the external protocol in languages like SMALLTALK and we have ignored them above. If instance variables are considered part of the external protocol they can be placed in class AnyType. Most likely they are not and can be placed in the class hierarchy where they make sense (as usual).

Using the style described here one can add classes, pointers and methods with almost no interference with the compiler. Any errors in line of thought or mistakes will show up during execution of the program in very much the same way as in a SMALLTALK system. The above rules lets us develop prototypes and perform experiments in a SMALLTALK like style. When these have been successfully completed we like to take benefit of the type system in order to help us create production quality applications. This can be done through incremental changes to a program developed using the three rules defined above. The modifications to the program will follow three routes: (1) Regrouping the classes into deeper hierarchies. (2) Propagate declarations of virtual operations from AnyType to more specialized classes. (3) Narrowing the pointer declarations to be more specific.

Conclusion

The motivation for this paper has been twofold. The proposed programming style is new an may turn out to be useful. The description of how type-less programming can be achieved in a language with strong typing highlights some of the important differences between these radically different approaches to object oriented software development. The type-less programming style concentrate attention on the organization of the code itself (code reuse). In the programming style using qualified references it is essential to find commonality between classes (modeling).

Typing Issues in Kea

W. B. Mugridge

My interest in this workshop stems from an ongoing research project into the design of Kea, a tightly integrated functional/object-oriented language. I provide a brief background of my work (with Hamer and Hosking) [33] before answering the specific questions of the workshop.

Our research is concerned with providing a programming system which is suited to building a class of applications which have not received wide attention. This class is characterised by the need to model complex, richly interconnected structures that are explored by the user to obtain some information, check requirements, expand a design, etc. The model is essentially static, but where the user makes changes to inputs to explore alternatives. In order to make this easy for the user and the programmer, the system must ensure consistency under change, so that the model that is visible to the user is always correct.

Object-orientation was adopted as an appropriate way of representing categories, structures, and interrelationships between these. Consistency implies the need for a functional language within this object-oriented setting, together with run-time dependency management to propagate the effects of user changes.

Kea is a statically-typed programming language that inherits from the OO paradigm the notions of encapsulation, inclusion-polymorphism, method overriding, and multiple inheritance. From the functional paradigm, Kea inherits higher-order and polymorphic functions, type inference, and lazy evaluation.

As discussed in the conference paper [52], KEA has the novel feature of "dynamic classification". A classification attribute specifies a "cluster": a set of mutually exclusive subclasses [63]. If class A has cluster $\{B, C\}$, dynamic classification ensures that any object of class A will also belong to either class B or C (but not both). In this way, clusters constrain types; for example, the presence of cluster $\{B, C\}$ means that no class can multiply inherit from both B and C.

KEA supports multivariant functions, a typed form of CLOS multi-methods. As with multi-methods, the code chosen for execution (during dispatching) depends on the type of all the function call arguments, rather than just the type of the primary object (self). Unlike CLOS, a form of encapsulation is enforced; a function defined within a class need not be part of the class signature.

Our Approach to Types

As far as the KEA programmer is concerned, classes are equated with types. Equating classes with types at the language level allows the benefits of dynamic classification to be realised. Multivariant functions permit subclassing to be equated with subtyping without violating contravariante constraints and hence they avoid the need to separate type and class at the language level, as compared to the approach of [10, 18].

At the language implementation level, we have a richer notion of types in which type does not equate to class. In order to provide for the typing of functions, least upper bound and greatest lower bounds must be defined; i.e., type expressions form a lattice.

KEA does not support assignment; I cannot comment on the issue of typing of updates.

We are still coming to grips with the subtleties of typing; subclassing introduces many interesting issues. While type systems for functional languages appear to provide a basis for typing object-oriented languages, they need to be extended to avoid various forms of type loss (such as those that arise for multivariant functions from the principal type property).

We wish to avoid explicit typing in Kea programs, including (bounded) parametric polymorphism. We are still exploring the extent to which such type inference can be carried out; certainly the typing of recursive multi-variant functions is more complex than the typing of recursive functions in functional languages (in Kea such typing is a multi-step process rather than the single-step unification process used in functional languages).

If type inference can be carried to object parameters, we can introduce (parametric) polymorphic classes (i.e., generic classes) simply by permitting the programmer to provide less type information in a program. If this becomes feasible, it will be interesting to compare the flexibility of such a system with the "type substitution" of Palsberg and Schwartzbach. Handling types appropriately (and thus avoiding type loss) under separate compilation is an important issue that will need to be addressed eventually.

Subclassing and Subtyping

Jens Palsberg and Michael I. Schwartzbach

We have studied an idealized subset of SMALLTALK in order to get a better understanding of subclassing and subtyping [53, 56, 54, 55]. The project is nearing completion, and in the following we survey its results and limitations.

Why Types?

There are three reasons for introducing types. All are motivated by deficiencies of untyped languages: they can be (1) unreadable, (2) unreliable, and (3) inefficient. Types can remedy this by providing (1) type annotations, (2) a safety guarantee, and (3) information for optimization. We have the pragmatic attitude that anything meeting these requirements deserves to be called a *type*.

For object-oriented languages, a suitable definition of type is *set of classes*. Let S be such a set and $\mathsf{x} : S$ a type annotation. The information it carries is that x can only evaluate to either nil or some instance of a class in S. The safety guarantee can by obtained by verifying for every message send $\mathsf{x.m}(\ldots)$ that all classes in S implement a method m. The optimization is primarily concerned with inlining the call of m whenever all classes in S have the *same* implementation of m.

Our idealized language uses *finite* sets of classes as types. We restrict ourselves to consider entire programs, and then finite sets of classes is an almost universal notion of type; after all, in a particular program any predicate on classes can be expanded to yield a finite set. Note that also interfaces and more general specifications correspond to predicates.

What is Subtyping?

Subtyping in object-oriented languages allows a more flexible typing of assignments and parameter passings compared to PASCAL-like languages. With types being sets of classes, it is natural to allow an object to have any type containing its class. It follows that subtyping is simply set inclusion; for an assignment `x:=y` the type of `y` must be contained in that of `x` (and similarly for parameter passings). Any sound notion of subtyping must respect inclusion of the associated sets.

What is Subclassing?

Subtyping and subclassing are conceptually unrelated. A language may have one of them or both. We view subclassing as being primarily concerned with code reuse. The well-known concept of inheritance can be seen as a textual short-hand, which is particularly useful since the compiled code can be reused, due to dynamic method lookup. In typed languages, it is also important if code is reused in a type-correct manner.

With this simple view, there are no problems with updates or type-loss.

Generalized Subclassing

In our view, the previous formal models of classes and types are inappropriate for studying code reuse. They put the emphasis on class *interfaces*, and tend to ignore or water down the imperative constructs.

It seems unavoidable that a theory concerned with code reuse must employ a model that makes explicit reference to source code. We have modeled classes in a direct manner, as regular trees with nodes labeled by untyped, gapped source code. Surprisingly, this can form the basis for a mathematically attractive theory of subclassing [55].

We have defined a partial order $\lhd$ on classes (i.e., on labeled trees) which is a *generalized* subclassing relation. Specifically, if $sup \lhd sub$ holds, then we know that *sub* can be implemented as a modification of *sup* while preserving type-correctness and reusing compiled code. The implementation is a straightforward generalization of the standard SMALLTALK interpreter, employing a dynamic search for the arguments of new [56].

The order $\lhd$ has the same basic properties as inheritance: it is decidable, has a least element, has finite intervals, preserves the recursive structure of classes, and excludes what we call *temporal cycles* (basically, cyclic inheritance relations).

Class Substitution

It turns out that $\lhd$ has two suborders which form an *orthogonal basis* (corresponding to a particular formal definition). One suborder is exactly realized by inheritance, while the other is realized by a new subclassing mechanism, which we call *class substitution.* Together, inheritance and substitution allow for the programming of all subclasses, and they are independent and. complementary.

Class substitution can be seen to implement a general kind of *genericity* which is superior to e.g. parameterized classes: any class is generic, can be gradually generically-instantiated, and has all of its generic instances as subclasses [53].

Type Inference

It might be considered desirable to obtain a safety guarantee and possibilities for optimizations without requiring the programmer to think about types. This leads to the problem of *type inference*, for which we have suggested a solution for our idealized language [54].

We have designed an algorithm that given an untyped program can compute a sound approximation of the type of every expression, i.e., a superset of the classes to instances of which it can evaluate. The algorithm employs a so-called trace graph and generates a finite set of conditional set inclusions, which is solved by a simple fixed-point computation.

Unlike some previous algorithms, ours can type-check most common programs. It is currently being implemented in a prototype version, which shows hopeful promises about its efficiency and applicability.

Separate Compilation

Our approach seems incompatible with separate compilation, in the sense of introducing new classes into a compiled program. We can, of course, allow separate compilation of method bodies.

When new classes may be introduced, then finite sets are no longer sufficient; after all,. one cannot predict the potential future classes. The solution is to use more general predicates such as $\mathsf{x} :\uparrow \mathsf{C}$, which expands to the infinite set of C and all its subclasses.

Unfortunately, most of our results break down for such (finitely represented) infinite sets. This is probably not coincidental, since languages supporting both inheritance and types such as $\uparrow \mathsf{C}$ invariably resort to some degree of dynamic type-checking (or accept an unsound type system).

Ideas for Types and Inheritance in OOP

Oskar Permvall

I'm a PhD-student writing on OOP, especially on classification and comparison of languages pretending they are object oriented. This includes finding out which language constructs can be considered object oriented and how they are defined. Functional programming is another major interest of mine and my opinion is that it would be fruitful to apply ideas in functional programming to concepts in OOP, e.g. higher order functions. My reason for this is increased expressibility, higher readability, and shorter code among other properties.

In order to define what is meant by being object-oriented one has to decide which properties that are important and how to characterize them. Some subjects connected to the topics of this workshop that I find relevant are the difference between values and objects, the importance of a type system, and the relation between inheritance and aggregation. Each of these will be treated below.

Values and objects

What is an object in OOP? Is the number 1 an object? My point of view is that it is not and I will try to motivate that a (good) OOL should have two type-systems, one for value-types and one for object-types. What then is the basic difference between value and object?

A value is equivalent to the totality of its components and its type is a listing of its component types, i.e., some kind of record, tuple, or list. Value types are similar to algebraic types in functional programming.

An object has more to it than its visible methods. An object contains an identity to distinguish it from other objects. Furthermore, it can contain a program counter, if the class is allowed to contain code. None of these should be visible in the type, i.e., an object is more than the totality of its components and cannot be created by gluing components together. It is natural to select the name of the class to be the type of an object.

The essential point is that a value does not have an identity, which gives that if a certain value is in one or two locations in the memory depending on how it is represented, should not matter to the programmer. Basic examples are all kinds of enumerable types, like numbers, booleans, strings and so on.

Another point is how the language treats equality, values should be tested by structural equality while checking identities is the right thing for objects. If we had that numbers where objects then it could happen that 1=/=1 since they were different instances, i.e., if numbers were objects then there has to be at most one instance of every number and it would carry an identity. Furthermore, if numbers are implemented as objects then it will not be easy to reason about expressions using ordinary algebraic laws.

A programmer is free to choose if he wants objects or values and design his structures based on that decision. There is nothing that forbids one to implement trees and other structures as values and complex numbers as objects but it will be unusual to do so. For example, in an OO system for graphics it seems clear that a line should be modeled as an object, but a point, is it a value or an object? My view is that it is a value but if it has more attributes than coordinates then I would consider it to be an object. Generally speaking, most concepts seem to have both aspects and it depends on the application which to choose (compare the wave/particle dualism of light in physics).

Values and objects are not on the same level because an object can have value components while a value cannot have objects as components since then it would become an object.

Typing

Types help you to structure your problem and increase readability and maintainability. Types is a complicated notion with several dimensions. Three of these dimensions are weak vs. strong, explicit vs. implicit, and static vs. dynamic. A strongly typed language has a fixed type associated with every variable and this type carries all information about the value stored in the variable. A weakly typed language is close to an untyped language, i.e., you do not know what to expect of a value stored in a variable. See [50] for a discussion of this an related matters. An explicitly typed language has all type information written out, while the compiler computes the necessary type information in an implicitly typed language. A statically typed language has no run-time checks and a dynamically typed language decides if it is well-typed at run-time. These dimensions are continuums and are orthogonal. Concepts such as polymorphism and other hierarchical type systems are within the first dimension.

Here follows some examples from the first dimension. Everyone that has tried to create a flexible list or tree system in PASCAL without using large variant records (which implies that type safety is lost) will agree that it is an example of a too strongly typed language. SIMULA's and BETA's class hierarchy and qualified reference system is one solution to that problem, but we have moved towards the weak end. SMALLTALK is a weakly typed language because it has no types associated with variables. LOOPS and other LISP-based OO systems belong to the same class. If a language is extended with parameterized types, then it does not necessarily change its position in the first dimension.

Having both values and objects implies two different type systems in the language. Classes in the object-world will be replaced by some kind of module in the value-world and these modules will be typed by their signatures rather than their names. The module system of ML can serve as an example of a module system in the value-world.

Inheritance and aggregation

Inheritance is a class operator. It takes a class and a class-piece (should be defined better) yielding a new class. A question well worth to investigate is if there are other operators that it would be useful to introduce into OOP?

Inheritance means different things to different people and this causes some controversy in the 00 community. The inheritance mechanism is used in at least two different ways, as a method for building a conceptual hierarchy and as a language for reusing or aggregating software components. This ambiguity is not good and my view is that inheritance should be restricted to the concept hierarchy and a new (sub-)language should be defined to cope with aggregation and its special needs. Multiple inheritance is often advocated as a solution for combining classes while it is actually an aggregation language that is wanted.

This aggregation language would have operators to combine classes to form new classes and specify sharing, internal communication, exception handling, and so on. However, the major unsolved question is if aggregations are classes or something else? As always, there will be definitions that is in-between, i.e., they can be considered being both a concept and an aggregation.

To introduce a third system of operators for the aggregation language is a bad solution. I have no evidence, but believe that a module language for the value type system would have many similarities with an aggregation language for the object world and therefore can be used as an aggregation language. There will be no problems to keep the semantics apart since building new value types involves no objects and vice versa. The only problem I can predict is that the aggregation language will need more operators, structures and so on than the module language needs.

On treating Basic and Constructed Types Uniformly in OOP

Markku Sakkinen

Objects, values, and types. I agree on most points with the classic paper [48]: both values and objects *and* a clear distinction between them are useful in programming. I would like to have both objects (concrete) and values (abstract) of all types, i.e., the abstract-concrete dichotomy should be independent of type. Especially, there should be objects (variables) of basic data types, as in most conventional, non-OO languages. However, there can be no abstract values of a reference type, nor of any complex type that contains references. Thus the set of object types is a proper superset of the set of value types.

A slight confusion between values and objects usually seems to be coupled with a similar confusion between objects and references. I want to distinguish between any type T and type ref T. I also prefer that aggregate objects need not have only references as their components. In these respects I regard languages like PASCAL, ADA, and C++ as superior to most object-oriented languages.

It is also useful to have immutable (constant) concrete objects. Furthermore, in the presence of references, it is sensible to distinguish between two degrees of immutability. An immutable object is *totally immutable* if and only if either it contains no references, or all references in it are to totally immutable objects. (To make all cyclic cases unambiguous, this would obviously need a little technical refinement.)

How to inherit from a basic type

I will discuss here only "essential inheritance" in the sense of [59], i.e., implying an *is-a* relationship. Since we are not limited to reference semantics, "incidental inheritance" can be seen simply as a special case of aggregation/composition, as outlined in that paper. The same view on "private inheritance" in C++ is taken in [5].

Inheritance (prefixing, type extension) is ordinarily defined for record like types only. Basic types such as Integer are not records; how should the same principle be extended to them? A solution essentially exists already in SIMULA: allow a superclass name, as an alternative to an attribute (component) name, as the second operand of a qua clause! If we define a new class My_integer that inherits from Integer, we can refer to the integer part by self qua Integer, or if you prefer, self.Integer.

Objects of type My_integer can be either mutable or immutable; the value of the integer part must be given in the initialisation at least for immutable objects. If the class were not to be completely encapsulated, literal My_integer values could be denoted by a display notation.

In the interest of symmetry, one could like to refer also to the incremental part ("difference class" instance) of a My_integer object as a whole, e.g., self qua (My_integer–Integer). There is one important point on which I prefer the meaning of class qualification in C++ to that of qua in SIMULA: in C++ it affects the selection of virtual operations. It is a great disadvantage in SIMULA that there is absolutely no way to invoke an overridden superclass version of a virtual procedure or function in the context of a subclass object.

One could inherit from any non-record constructed types (arrays, sets, sequences, . . .) i.n the same way as from basic types, if such type constructors exist in the language at hand. My opinion is that it is nice to have such structured types as language primitives rather than as parameterised classes.

Does this generalise well?

With single inheritance, there is evidently no difficulty when the parent is an "ordinary" (record-structured) class—of course: that is the case we generalised *from*. Even multiple inheritance (MI) is rather simple if the parent classes are pairwise mutually independent, i.e., have no common ancestors. I would regard any name clashes in this case as accidental, to be resolved by superclass qualification.

The "fork-join" MI case, i.e., where parent classes have common ancestors, is more complex. While I did not see any problems in the MI principles of C++ yet in [59], I now agree with [5] that there should always be only one subobject of any ancestor class. I.e., in the C++ model and terminology, base classes in **public** inheritance should in principle always be **virtual**. The proposal of the previous section makes sense un-der this approach: e.g. multi-level superclass qualifications would never be needed.

Of course, all subobjects cannot be contiguous in fork-join MI. This is one argument (efficiency) why it should be possible to forbid, in a class definition, the use of a given inheritance relationship in fork-join MI by subclasses. If class B inherits class A with this restriction, no further class should inherit both B and another subclass of A. There can often be a much stronger, semantic reason for such a restriction besides efficiency: the need for guaranteedly non-shared subobjects.

No magic can help that the handling of virtual operations that are overridden on more than one path from their class of declaration to the most specific subclass, is problematic, as illustrated in the literature [64]. It could be considered to forbid fork-join inheritance also in such cases. On the other hand, method combination schemes as in CLOS can obviously help a lot in typical situations.

Some subtleties

In the "object algebra", we could regard the non-inherited parts of each class definition as forming the *basic* subobjects, and any combination of those could be regarded as a subobject. (If Paul Johnson's *fine-grained inheri-*

tance principle is followed, there will be a lot of basic subobjects in most objects.) **qua** could then have any corresponding class combination as its second operand.

Now, if we invoke a virtual operation using qua, what should happen to self in that invocation? As far as I know, self still denotes the whole object in all common languages. There could be a need for an alternative that would bind self to the qualified subobject. This would be a slight generalisation of here, as recently proposed by Mario Wolczko.

There are well-known complications in comparing and copying objects. One reason why I prefer value semantics to reference semantics is that, when classes are sensibly designed, straight component-by-component comparison can often be "the right thing" in ADA or C++; in referencebased languages almost always, shallow equality is too shallow and deep equality too deep. The same applies to copying. Besides, reference-based languages usually do not offer any equivalent of a PASCAL or ADA *assignment* except for simple types and references.

One typical disadvantage of languages with value semantics (e.g. C++) is that restricted polymorphism (subclass object standing for superclass object) is possible only for references (pointers). However, this defect is avoidable: [37] contains a proposal for extending polymorphism to non-pointer, class-type variables. That proposal could be implemented e.g. by variables with "reference pragmatics" but value semantics.

Types in Ansa DPL

Andrew Watson

ANSA is a distributed computing architecture with an object-based computational model in which clients interact with objects through interfaces, each comprising a set of operations. In order to support the heterogeneity implicit in distributed systems, the computational model type system assigns abstract types to interfaces. Objects themselves cannot be directly manipulated and therefore have no type.

Introduction

The Ansa computational model

ANSA (the Advanced Networked Systems Architecture) is an architecture for distributed computing systems.

ANSA's application programming model (called the *computational model*) is intended to insulate the programmer from the way that his application is distributed over a network of computers. This *distribution transparency* is achieved using an object-based[4] model in which all objects are potentially located on separate network nodes, and only interact by explicitly requesting each other to perform actions on the state that they encapsulate. Since all communication between objects is explicit, potential failures (either of

[4]The term object-based is used in the sense proposed by Wegner [68] to mean that the model has objects, but does not support classes or inheritance.

communication or of objects) have well-defined effects which are incorporated into the model.

Compared to the Smalltalk- object model [27], the Ansa computational model has the following salient features;

- Objects are completely autonomous. They may be created by factory objects, but thereafter have no special relationship with the factory or any other object. There is no object containment, classes or run-time ("reactive") inheritance.
- Each object may have one or more *interfaces*, each of which is a collection of named *operations*, akin to methods. Clients of the object hold references to interfaces (rather than to the object itself), and interact with the object by invoking operations in those interfaces. This allows an object to present different abstractions to different clients. Objects may dynamically create new interfaces.
- Each operation has a predefined set of possible outcomes, called *terminations*, each with an intrinsic name that distinguishes it from the operation's other possible terminations. All terminations have equal status, and each carries a predefined number of result param-eters.

Type system requirements

Support for heterogeneity

Ansa hides heterogeneity (both of hardware and software) from the application programmer. The computational model type system must classify object interfaces solely by the services they perform, not by their implementation. This leads to a system based on abstract types.

Support for trading

Ansa supports dynamic configuration of distributed systems via *trading.* A trader is a broker that accepts and records references to object interfaces providing publically-offered services. Upon receiving from a prospective client a specification of a service that it requires, the trader searches its internal database and returns a reference to a suitable service, if it knows of one. The client then invokes the service using this interface reference, without further interaction with the trader.

Should a client attempt to invoke on a service obtained from the trader an operation that it does not support then an *interaction error* results. To guard against this, and also against the complimentary failure should the service return a termination not understood by the client, the trader accepts from the client a representation of the abstract type it expects the service to possess. The trader checks this against the abstract type of the offered service to ensure that there can be no interaction error when the client comes to use the service.

There are many possible relationships between a pair of types that satisfy this requirement; the weakest (and hence the least restrictive on the programmer) is the type conformance relationship [57]. Briefly, type S conforms to type T provided that every operation that can be performed on T may also be performed on S, and every termination from every operation in S is also present in the corresponding operation on T. Where the operations and terminations have parameters, these must match in number and conform in the appropriate direction.

Support for federation

Ansa supports interconnecting hitherto-unrelated distributed computing systems, allowing interworking that is transparent to the application programmer. This precludes type systems based on programmer assertions of relationships between types; when two systems "meet" for the first time, merging the type relationship databases of the two would require a great deal of (human) effort. This *federation* requires a type system in which the relationships between types are computed automatically as needed (e.g. when trading) rather than explicitly specified in advance.

DPL

Dpl (Distributed Programming Language) is being developed for constructing Ansa distributed applications using the computational model. While computationally complete and capable of being used as a programming language in its own right, Dpl is intended to be used in conjunction with an existing language (such as C).

Dpl is a statically-typed, block-structured object-based language whose design owes much to that of Emerald [57].

Bindings in DPL

Object state is represented as bindings of names to values; these bindings may be either *variable*, meaning that their contents may be changed by assignment, or *constant*, meaning that they may not be updated once established. This distinguishes bindings that are used to name parameters and intermediate results from those that represent mutable object state, thus assisting both programmers and compilers.

Type declarations

Dpl uses few type declarations; interface types are declared explicitly when:

- creating a variable binding, but not a constant binding (since in the former case the programmer may wish subsequently to assign interfaces of various types to the binding)
- declaring the formal parameter list of an operation (since the body of the operation may be compiled separately from programs that invoke it).

Current status

Initial work on the type system was performed by Alastair Tocher (BNR Europe) and Owen Rees (APM) [65], with the latter implementing a type checker as part of the prototype implementation of DPL. This performed static type checks where possible, but the design of the language at that time did not permit the types of all expressions to be determined statically.

After study by Simon Brock (UEA) and the current author the structure of DPL is being altered to remove these unhelpful features of the language, which (amongst other things) allowed the programmer to produce types that change over time and type computations whose outcome depends on data not available until run-time. This redesign is still in progress, but it is expected that a static type system can be constructed which supports all ANSA's requirements.

Model-oriented Type Descriptions and Inheritance

Alan Wills

Types (descriptive of behaviour) and classes (prescriptive of implementation) are clearly different. Inheritance can be seen as the derivation of one description from another—whether of type, class, or whatever. Subtyping is about substitutability, and subclassing is arguably just about inheritance of class attributes.

Automatic type-checkers are limited to checking signatures—sometimes with extra clues about the programmer's intentions from naming, property labels (as in POOL) or a theoretically spurious linkage to inheritance (as in C++). But ideally, subtyping should depend only on proper behavioural descriptions. This becomes practical and interesting if we are interested in formal specification and verification (aside from any questions about whether *that* is a worthwhile endeavour!)

So what is the most practical way to describe a type? What implications does inheritance have for this description?

Model oriented or axiomatic (or both)?

The two traditional camps in specification-the property-oriented algebraists (OBJ etc.) and the operation-specifying model-orienteers (Z, VDM etc.)—are not at such opposite and immiscible poles as they might seem.

Both styles begin by defining a type as a signature: i.e., a set of signatures of visible operations. The divergence comes in defining the behaviour. The operation-specifying styles achieve some clarity of notation and simplified proofs by defining just one property per operation; but this is a convenient rather than a mandatory association with the model-oriented approach, and multiple properties per operation are useful when combining inherited and locally-defined properties.

Algebraic specifications define equalities between applicative terms or sequences of operations; whilest *model-oriented specifications* define the effects of operations on a hypothetical set of components (a "model") of the object's state, and vice versa. This contributor favours appropriate technology: algebraic specification is necessary and useful for the simple types like numbers and stacks; model-oriented specification is more readable and intuitive for anything larger.

In any case, some algebraic specifications will always require intermediate existentially-quantified variables [7], which amount to a model. The point of a model is that it shares one description of the intermediate state between several properties, making them easier to read and less errorprone to work with.

Type notation

So a model-oriented type description consists of a signature, model components (hidden variables) and properties. The operations in the signature may be "typed" in the conventional sense, so that the automatic typechecker can do some of your verification; the theorem-prover (wholly or partly human) takes over for the hard stuff in the properties.

Type-descriptions can be combined with class-descriptions—which are signature, variables, and methods—allowing properties to directly document real variables and operations; and supporting the traditional and useful "abstract class", which is both specification and part implementation.

The usual op-spec style divides properties into invariants (static constraints on hidden components) and op-specs (governing state transitions) in the form $pre \Rightarrow post$, where *post* includes references to before and after states. This separation simplifies implementation proofs, especially where the models are different. But you have to be careful that op-specs don't inadvertently impose static constraints on model variables.

Model-oriented subtyping and inheritance

You choose a specification model for convenience and clarity; implementations are chosen for performance and maintainability, and may look entirely different. You have to verify that your implementation behaves the same as your model dictates (but not that it uses the same components). There's no violation of the principle of encapsulation here: the model-oriented specification tells you nothing about the innards of any implementation.

An example is a compiler symbol table, modeled with clarity (and user-defined generic types) as

$$\mathsf{SymbolTable(Ref)} = \mathsf{Stack(Dictionary(Symbol,\ Ref))}$$

but implemented much more efficiently, perhaps as a single dictionary.

A complex development should end up with a simple model at one extreme, an implementation at the other, and perhaps several "reification" stages in between. Since the models should be different, this subtyping will have nothing to do with inheritance, but instead should be documented with reification proofs (even if informal ones).

On the other hand, where the reification *does* happen to be an extension by inheritance, the proofs are a lot easier, if not vacuous.

A type which inherits its properties is bound to be substitutable for its parent; but properties inherited from different parents may conflict with each other and with the locally-added properties, resulting in an unimplementable type. The art of successful model-inheritance is therefore to ensure that the extensions are *conservative*. Essentially, invariants should only constrain their "own" variables, not those belonging to a parent. Name-clashes from multiple parents need to be resolved by renaming unless the variables originate from a common ancestor. Op-specs applying to the same operation can be conjoined; the case where the preconditions are disjoint is trivially non-conflicting.

The model-oriented approach seems, at not too onerous an expense, to offer a good readable style—which is the important thing while program constructors and theorem provers are still people.

Bibliography

[1] Pierre America and Frank van der Linden. A parallel object-oriented language with inheritance and subtyping. In *Proc. OOPSLA/ECOOP'99, ACM SIGPLAN Fifth Annual Conference on Object-Oriented Programming Systems, Languages and Applications; European Conference on Object-Oriented Programming*, 1990. Published in SIGPLAN Notices, 25(11).

[2] Birger Andersen, Ellie language definition report. *ACM SIGPLAN Notices*, 25(11):45–64, 1990. Second edition as DIKU Report no. 91/3, Department of Computer Science, Copenhagen, Denmark, June 1991.

[3] Birger Andersen. Ellie – a general, fine-grained, first class object based language. Submitted to Journal of Object-Oriented Programming, 1991.

[4] Birger Andersen. *Grain-Size Adaption in the Fine-Grained Object-Oriented Language Ellie.* PhD thesis, 1991. DIKU Report no. 91/5, in preparation.

[5] Kenneth Baclawski. The structural semantics of inheritance. Submitted for publication, 1990.

[6] Paul Bergstein. Object-preserving class transformations. In *Proc. OOPSLA '91, ACM SIGPLAN Sixth Annual Conference on Object-Oriented Programming Systems, Languages and Applications*, 1991.

[7] Bergstra, Broy, Tucker, and Wirsing. On the power of algebraic specification of data types. pages 193–204. Springer-Verlag (*LNCS* 118), 1981.

[8] Andrew Black, Norman Hutchinson, Eric Jul, Henry Levy, and Larry Carter. Distribution and abstract types in Emerald. *IEEE Transactions*

on Software Engineering, 13(1):65–76, 1987. Also Technical Report 86-02-04, Department of Computer Science, University of Washington.

[9] Andrew P. Black and Norman Hutchinson. Typechecking polymorphism in Emerald. Technical report, Digital Cambridge Research Laboratory, One Kendall Square, Building 700, Cambridge, MA 02139, December 1990. Technical Report CRL 91/1.

[10] Peter S. Canning, William R. Cook, Walter L. Hill, and Walter G. Olthoff. Interfaces for strongly-typed object-oriented programming. In *Proc. OOPSLA'89, Fourth Annual Conference on Object-Oriented Programming Systems, Languages and Applications.* ACM, 1989.

[11] L. Cardelli, J. C. Mitchell, S. Martini, and A. Scedrov. An extension of system F with subtyping. In *Proc. TACS'91*, 1991.

[12] L. Cardelli and P. Wegner. On understanding types, data abstraction, and polymorphism. *ACM Computing Surveys*, 17(4):471–522, December 1985.

[13] Luca Cardelli. Typeful programming. Technical Report No. 45, Digital Equipment Corporation, Systems Research Center, 1989.

[14] D. Carrington, D. Duke, R. Duke, P. King, G. Rose, and G. Smith. An object-oriented extension to Z. In S. Vuong, editor, *Formal Description Techniques, II (FORTE'89).* 1990.

[15] Craig Chambers and David Ungar. Customization: Optimizing compiler technology for self, a dynamically-typed object-oriented programming language. In *Proc. SIGPLAN'89 Conference on Programming Language Design and Implementation*, 1989. Published as SIGPLAN Notices 24(7), July, 1989.

[16] Craig Chambers and David Ungar. Iterative type analysis and extended message splitting: Optimizing dynamically-typed object-oriented programs a dynamically-typed object-oriented programming language. In *Proc. SIGPLAN'90 Conference on Programming Language Design and Implementation*, 1990. Published as SIGPLAN Notices 25(6), June, 1990.

[17] Marshall Cline and Douglas Lea. The behavior of C++ classes. In *Proc. Symposium on Object-Oriented Programming Emphasizing Practical Applications*, September 1990. Marist NY.

[18] William Cook, Walter Hill, and Peter Canning. Inheritance is not subtyping. In *Seventeenth Symposium on Principles of Programming Languages*, pages 125–135. ACM Press, January 1990.

[19] R. Cooper, H. Nguyen Van, M. Riveill, C. Roisin, and F. Wai. Guide language manual. Technical report, BULL-IMAG, Systèmes, Grenoble, 1990. Num. rapport 3 English Version.

[20] L. Damas and R. Milner. Principle type-schemes for functional programs. In *Ninth Symposium on Principles of Programming Languages*, pages 207–212. ACM Press, January 1982.

[21] U. Dayal. Queries and views in an object-oriented data model. In *Proc. 2nd International Worlcshop on Database Programming Languages*, June 1989.

[22] D. Duke and R. Duke. Towards a semantics for Object-Z. In *VDM'90: VDM and Z!* Springer-Verlag (*LNCS* 428), 1990. D. Bjfirner, C.A.R. Hoare, and H. Langmaack, editors.

[23] R. Duke, P. King, G. Rose, and G. Smith. The Object-Z specification language (version 1). Technical Report No. 91–1, Software Verification Research Centre, University of Queensland, May 1991.

[24] R. Duke, P. King, and G. Smith. Formalising behavioural compatibility for reactive object-oriented systems. In *Proc. Australian Computer Science Conference (ACSC-14)*, pages 11/1-11/11, 1991. Sydney.

[25] R. Duke, G. Rose, and A. Lee. Object-oriented protocol specification. In L. Logrippo, R.L. Probert, and H. Ural, editors, *Protocol Specification, Testing, and Verification, X*, pages 325–338. North-Holland, 1990.

[26] R. Balter et al. Architecture and implementation of Guide, an object-oriented system. To appear in Computing Systems, 1991.

[27] A. Goldberg and D. Robson. *Smalltalk-80—The Language and its Implementation.* Addison-Wesley, 1983.

[28] A. Goldberg and D. Robson. *Smalltalk-80—The Language.* Addison-Wesley, 1989.

[29] Bjarne Hansen. Object activation in a transputer implementation of the Ellie language. Technical report, Department of Computer Science, University of Copenhagen, Denmark, March 1991. M.Sc. Thesis, DIKU Report no. 91/6.

[30] Andreas V. Hense. Denotational semantics of an object oriented programming language with explicit wrappers. Technical Report No. A 11/90, Fachbericht 14, Universität des Saarlandes, June 1990.

[31] Andreas V. Hense. Polymorphic type inference for a simple object oriented programming language with state. Technical Report No. A 20/90, Fachbericht 14, Universität des Saarlandes, December 1990.

[32] Urs Hölzle, Craig Chambers, and David Ungar. Optimizing dynamically-typed object-oriented languages with polymorphic inline caches. In *Proc. ECOOP'91, Fifth European Conference on Object-Oriented Programming*, 1991.

[33] J. G. Hosking, J. Hamer, and W. B. Mugridge. Integrating functional and object-oriented programming. In *Proc. TOOLS Pacific'90 Conference*, 1990.

[34] Norman Hutchinson. *Emerald: An Object-Oriented Language for Distributed Programming.* PhD thesis, Department of Computer Science, University of Washington, January 1987.

[35] S. Krakowiak, M. Meysembourg, H. Nguyen Van, M. Riveill, and C. Roisin. Design and implementation of an object-oriented, strongly typed language for distributed applications. *Journal of Object-Oriented Programming*, 3(3):11–22, 1990, September/October.

[36] B. B. Kristensen, O. L. Madsen, B. Møller-Pedersen, and K. Nygaard. The BETA programming language. In B. Shriver and P. Wegner, editors, *Research Directions in Object-Oriented Programming*, pages 7–48. MIT Press, 1987.

[37] Douglas Lea. Customization in C++. In *Proc. 1990 USENIX C++ Conference*, pages 301–314, 1990.

[38] Karl Lieberherr. Object-oriented programming with class dictionaries. *Journal on Lisp and Symbolic Computation*, 1(3), 1988.

[39] Karl Lieberherr and Paul Bergstein. Incremental class dictionary learning and optimization. In *Proc. ECOOP'91, Fifth European Conference on Object-Oriented Programming*, 1991.

[40] Karl Lieberherr, Paul Bergstein, and Nacho Silva-Lepe. Abstraction of object-oriented data models. In *Proc. International Conference on Entity-Relationship Approach*, pages 81–94. Elsevier, 1990.

[41] Karl Lieberherr, Paul Bergstein, and Nacho Silva-Lepe. From objects to classes: algorithms for optimal object-oriented design. *Journal of software Engineering*, July 1991.

[42] Karl Lieberherr and Ian Holland. Assuring good style for object-oriented programs. *IEEE Software*, pages 38–48, September 1989.

[43] Karl Lieberherr and Ian Holland. Formulations and benefits of the law of Demeter. *Sigplan Notices*, 24(3):67–78, 1989.

[44] Karl Lieberherr and Ian Holland. Tools for preventive software maintenance. In *Proc. Conference on Sofiware Maintenance*, pages 2–13. IEEE, 1989.

[45] Karl Lieberherr, Ian Holland, and Arthur Riel. Object-oriented programming: An objective sense of style. In *Proc. OOPSLA'88, ACM SIGPLAN Third Annual Conference on Object-Oriented Programming Systems, Languages and Applications*, pages 323–334, September 1988. A short version of this paper appeared in IEEE Computer Open Channel, June 1988, pp 79–80.

[46] Karl Lieberherr and A. J. Riel. Demeter: A case study of software growth through parameterized classes. *Journal of Object-Oriented Programming*, 1(3), 1988. A preliminary version of this paper was published in International Conference on Software Engineering, Singapore, 1988, pp 254–264.

[47] Karl Lieberherr and Arthur Riel. Contributions to teaching object-oriented design and programming. In *Proc. OOPSLA'89, ACM SIGPLAN Fourth Annual Conference on Object-Oriented Programming Systems, Languages and Applications*, pages 11–22, 1989.

[48] B. J. MacLennan. Values and objects in programming languages. *Sigplan Notices*, 17(12):70–79, December 1982.

[49] Ole L. Madsen and Birger Møller-Pedersen. Virtual classes: A powerful mechanism in object-oriented programming. In *Proc. OOPSLA'89, Fourth Annual Conference on Object-Oriented Programming Systems, Languages and Applications.* ACM, 1989.

[50] Ole Lehrmann Madsen, Boris Magnusson, and Birger Møller-Pedersen. Strong typing of object-oriented languages revisited. In *Proc. OOPSLA/ECOOP'90, ACM SIGPLAN Fifth Annual Conference on Object-Oriented Programming Systems, Languages and Applications; European Conference on Object-Oriented Programming*, 1990.

[51] H. G. Mairson. Decidability of ML typing is complete for deterministic exponential time. In *Seventeenth Symposium on Principles of Programming Languages*, pages 382–401. ACM Press, January 1990.

[52] W, B. Mugridge, J. Hamer, and J. G. Hosking. Multi-methods in a statically-typed programming language. In *Proc. ECOOP'91, Fifth European Conference on Object-Oriented Programming*, 1991.

[53] Jens Palsberg and Michael I. Schwartzbach. Type substitution for object-oriented programming. In *Proc. OOPSLA/ECOOP'90, ACM SIGPLAN Fifth Annual Conference on Object-Oriented Programming Systems, Languages and Applications; European Conference on Object-Oriented Programming*, 1990.

[54] Jens Palsberg and Michael I. Schwartzbach. Object-oriented type inference. In *Proc. OOPSLA'91, ACM SIGPLAN Sixth Annual Conference on Object-Oriented Programming Systems, Languages and Applications*, 1991.

[55] Jens Palsberg and Michael I. Schwartzbach. Static typing for object-oriented programming. Computer Science Department, Aarhus University. PB-355. Submitted for publication 1991.

[56] Jens Palsberg and Michael I. Schwartzbach. What is type-safe code reuse? In *Proc. ECOOP'91, Fifth European Conference on Object-Oriented Programming*, 1991.

[57] Rajendra K. Raj, Ewan Tempero, Henry M. Levy, Andrew P. Black, Norman C. Hutchinson, and Eric Jul. Emerald: A general-purpose programming language. *Software — Practice and Experience*, 21(1):91–118, 1991.

[58] Didier Rémy. Typechecking records and variants in a natural extension of ML. In *Sixteenth Symposium on Principles of Programming Languages*, pages 77–88. ACM Press, January 1989.

[59] Markku Sakkinen. Disciplined inheritance. In *Proc. ECOOP'89, Europeun Conference on Object-Oriented Programming*, pages 39–56, 1989.

[60] M.H. Scholl, C. Laasch, and M. Tresch. Updatable views in object-oriented databases. Technical Report No. 150, ETH Zürich, Dept. of Computer Science, 1990.

[61] M.H. Scholl and H.-J. Schek. A relational object model. In *Proc. ICDT '90, International Conf. on Database Theory.* Springer-Verlag (*LNCS* 470), December 1990. Paris.

[62] M.H. Scholl and H.-J. Schek. A synthesis of complex objects and object-orientation. In *Proc. IFIP TC2 Conf. on Object Oriented Databases (DS-4)*, July 1990. Windermere, UK.

[63] J. M. Smith and D. C. P. Smith. Database abstractions: aggregation and generalisation. *ACM Transactions on Database Systems*, 2(2):105–133, 1977.

[64] B. Stroustrup. Multiple inheritance for C++. In *Computing Systems*, volume 2(4), pages 367–395, 1989.

[65] ANSA Team. ANSA reference manual, volume c, part x, chapter 7. Technical report, APM Ltd, March 1989.

[66] Mads Tofte. *Operational Semantics and Polymorphic Type Inference.* PhD thesis, Univ. of Edinburgh, May 1988. CST-52-88 also published as ECS-LFCS-88-54.

[67] Mitchell Wand. Type inference for record concatenation and multiple inheritance. In *LICS'89, Fourth Annual Symposium on Logic in Computer Science*, pages 92–97, 1989.

[68] P. Wegner. Dimensions of object-based language design. In *Proc. OOPSLA'87, Object-Oriented Programming Systems, Languages and Applications*, 1987.

[69] K. Wilkinson, P. Lyngbaek, and W. Hasan. The Iris architecture and implementation. *IEEE Trans. on Knowledge and Data Engineering*, 2(1):63–75, March 1990.

Discussions

The workshop is organized as five discussions, the topics of which are outlined below.

Classes versus Types

This topic has been touched upon by most position papers. There is some concensus that classes describe implementations, whereas the role of types is much more in question. Definitions of types range from types in the λ-calculus, over class interfaces, to simply classes. Naturally, these views carry over to subclassing and subtyping. Do the differences reflect genuine disagreement about the premises of object-oriented programming, or do they merely arise from emphasizing different areas of application?

Static versus Dynamic Typing

Many languages aspire to achieve static typing. However, this ideal is often sacrificed for added flexibility. It has been argued that type systems based on contravariance are too restrictive; others argue that type systems based on covariance require too many run-time checks. There is also some dispute about the motivations for wanting static typing. Efficiency is often brought up, but how strong is the connection? Can one responsibly deliver a product in which “message not understood” may occur?

Type Inference

Do programmers want explicit type information in their programs, or do they prefer the prototyping style of e.g. SMALLTALK? Are types in many existing languages too cumbersome? Can their benefits be obtained through type inference?

Challenge Session

Most type systems are quite hard to compare on a formal basis, since their premises are radically different. A shortcut to reaching an understanding may be to compare the way some specific programming tasks are achieved under the various approaches. We invite participants to present small examples that they feel are important and representative.

The Future?

Cardelli, in his position paper, warns that many languages may be “just too complicated” and advocates that the time is ripe for consolidating some basic ideas. Does this correspond to common attitudes? If so, which are the basic ideas that we can agree upon? What are the future developments that most people would welcome as important and significant?

Names and Addresses

The symbol ☺ means "participant"; □ means "position paper (co)author".

Biger Andersen	Status:	☺ □
Department of Computer Science	Tel:	+45 31 39 64 66
University of Copenhagen	Fax:	+45 31 39 02 21
Universitetsparken 1	E-mail:	birger@diku.dk
2100 Copenhagen		
Denmark		
Andrew P. Black	Status:	☺ □
Digital Equipment Corporation	Tel:	+1 617 621 6633
Cambridge Research Laboratory	Fax:	+1 617 621 6650
1 Kendall Square, Building 700	E-mail:	black@crl.dec.com
Cambridge, Massachusetts 02139		
USA		
Gregor v. Bochmann	Status:	☺ □
Departement d'Informatique et	Tel:	+1 514 343 7484
de Recherche Operationnelle	Fax:	+1 514 343 5834
Universite de Montreal	E-mail:	bochmann@iro.umontreal.ca
Canada		
Gilad Bracha	Status:	☺ □
Computer Science Department	Tel:	+1 801 521 5211
University of Utah	Fax:	
Salt Lake City, UT 84112	E-mail:	gilad@cs.utah.edu
USA		

Simon Brock	Status:	☺
School of Information Systems	Tel:	+44 603 593 164
UEA	Fax:	
Norwich NR4 7TJ	E-mail:	shb@sys.uea.ac.uk
England		
Brian Brown	Status:	☺
Methodologie du Logiciel	Tel:	3080-6382
Bull SA F6/0D/07	Fax:	3080-7078
Rue Jean-Jaures	E-mail:	brian.brown@fecl.bull.fr
F78340 les Clayes-sous-Bois		
France		
Luca Cardelli	Status:	□
Digital Equipment Corporation	Tel:	+1 415 853 2100
Systems Research Center	Fax:	+1 415 324 4873
130 Lytton Avenue	E-mail:	luca@src.dec.com
Palo Alto CA 94301		
USA		
David Carrington	Status:	☺ □
Software Verification	Tel:	
Research Centre	Fax:	+61 7 365 1999
Department of Computer Science	E-mail:	davec@cs.uq.oz.au
The University of Queensland		
Australia		
Bruce A. Conrad	Status:	☺ □
WordPerfect Corporation	Tel:	+1 801 228 7006
1555 North Technology Way	Fax:	+1 801 222 5077
Orem, Utah 84057	E-mail:	conrad@bert.cs.byu.edu
USA		

Rainer Fischbach	Status:	☺ ☐
Hirtenbrimnle 25	Tel:	+49 7478 2149
D-7245 Starzach	Fax:	+49 7478 2148
Germany	E-mail:	rf@garfield.t-informatik.ba-stuttgar

Andreas V. Hense	Status:	☺ ☐
Fachbereich 14	Tel:	+49 681 302 2464
Universitaet des Saarlandes	Fax:	
Im Stadtwald	E-mail:	hense@cs.uni-sb.de
D-6600 Saarbruecken 11		
Germany		

John S. Hogg	Status:	☺ ☐
Mail stop QE 012	Tel:	+1 613 763 7197
Bell-Northern Research Ltd.	Fax:	+1 613 763 5647
P.O. Box 3511, Station C	E-mail:	hogg@bnr.ca
Ottawa, Ontario K1Y 4H7		
Canada		

Urs Hölzle	Status:	☺ ☐
Stanford University	Tel:	+1 415 725 3695
Center for Integrated Systems	Fax:	+1 415 725 6949
RM 042	E-mail:	hoelzle@cs.standford.edu
Stanford, CA 94305-4070		
USA		

Norman C. Hutchinson	Status:	☺ ☐
Department of Computer Science	Tel:	+1 604 224 8188
University of British Columbia	Fax:	
6356 Agricultural Road	E-mail:	norm@cs.ubc.ca
Vancouver, B.C. V6T 1Z2		
Canada		

Eric Jul	Status:	☺
DIKU	Tel:	+45 3139 6466
Universitetsparken 1	Fax:	+45 3139 0221
2100 Kbh	E-mail:	eric@diku.dk
Denmark		

Christian Laasch	Status:	☺ □
Department of Computer Science	Tel:	+41 1 254 7243
Information Systems - Databases	Fax:	+41 1 262 3973
ETH-Zürich	E-mail:	laasch@inf.ethz.ch
CH-8092 Zürich		
Switzerland		

Serge Lacourte	Status:	☺
Bull-IMAG Systèmes	Tel:	+33 7654 4912
2 rue de Vignate	Fax:	+33 7654 7615
38610 Gières	E-mail:	lacourte@mururoa.imag.fr
France		

Doug Lea	Status:	☺ □
SUNY at Oswego	Tel:	+1 315 341 2688
NY CASE Center	Fax:	
NY 13126	E-mail:	dl@oswego.edu
USA		

Emmanuel Lenormand	Status:	□
Bull-IMAG Systèmes	Tel:	+33 7654 4912
2 rue de Vignate	Fax:	+33 7654 7615
38610 Gières	E-mail:	lenorman@mururoa.imag.fr
France		

Karl Lieberherr	Status:	☺ □
Northeastern University	Tel:	+1 617 437 2077
College of Computer Science	Fax:	+1 617 437 5121
360 Huntington Avenue	E-mail:	lieber@corwin.ccs.northeastern.edu
Boston, MA 02115		
USA		

Ole Lehrmann Madsen	Status:	☺ □
Computer Science Department	Tel:	+45 8612 7188
Aarhus University	Fax:	+45 8613 5725
Ny Munkegade	E-mail:	olmadsen@daimi.aau.dk
DK-8000 Aarhus C		
Denmark		

Boris Magnusson	Status:	☺ □
Department of Computer Science	Tel:	+46 4610 8044
University of Lund	Fax:	+46 4613 1021
PO Box 118, S-221 00 Lund	E-mail	boris@dna.lth.se
Sweden		

Gianni Mainetto	Status:	☺
CNUCE - Institute of CNR	Tel:	+39-50-593285
Via S. Maria, 36	Fax:	+39-50-576751
56126 Pisa	E-mail:	gmsys@icnucevm.cnuce.cnr.it
Italy		

W. B. (Rick) Mugridge	Status:	☺ □
Department of Computer Science	Tel:	NZ+9 737 999 x8914
University of Auckland	Fax:	NZ+9 737 934
Auckland	E-mail:	rick@cs.aukuni.ac.nz
New Zealand		

Birger Møller-Pedersen	Status:	☺ □
Norwegian Computing Center	Tel:	+47 2 45 3500
Gaustadalleen 23	Fax:	+47 2 69 7660
P.O.Box 114 Blindern	E-mail:	birger@nr.no
N-0314 Oslo 3		
Norway		

Jens Palsberg	Status:	☺ □
Computer Science Department	Tel:	+45 8612 7188
Aarhus University	Fax:	+45 8613 5725
Ny Munkegade	E-mail:	palsberg@daimi.aau.dk
DK-8000 Aarhus C		
Denmark		

Oskar Permvall	Status:	☺ □
Department of Computer Science	Tel:	+46 4610 8044
Lund University	Fax:	+46 4613 1021
PO Box 118, S–221 00 Lund	E-mail:	oskar.permvall@dna.lth.se
Sweden		

Michel Riveill	Status:	□
Bull-IMAG Systèmes	Tel:	+33 7654 4912
2 rue de Vignate	Fax:	+33 7654 7615
38610 Gières	E-mail:	riveill@mururoa.imag.fr
France		

Markku Sakkinen	Status:	☺ □
Department of Computer Science	Tel:	+35 841 603 016
And Information Systems	Fax:	
University of Jyväskylä	E-mail:	sakkinen@jytko.jyu.fi
PL 35, SF-40351 Jyväskylä		
Finland		

Marc H. Scholl	Status:	□
Department of Computer Science	Tel:	+41 1 254 7243
Information Systems - Databases	Fax:	+41 1 262 3973
ETH-Zürich	E-mail:	scholl@inf.ethz.ch
CH-8092 Zürich		
Switzerland		

Michael I. Schwartzbach	Status:	☺ □
Computer Science Department	Tel:	+45 8612 7188
Aarhus University	Fax:	+45 8613 5725
Ny Munkegade	E-mail:	mis@daimi.aau.dk
DK-8000 Aarhus C		
Denmark		
Alan Snyder	Status:	☺
Hewlett-Packard Laboratories	Tel:	+1 415 857 8764
P.O. Box 10490	Fax:	+1 415 857 8526
Palo Alto CA 94306-0971	E-mail:	snyder@hpl.hp.com
USA		
Clemens A. Szyperski	Status:	☺
Institut für Computersysteme	Tel:	+41 1 254 7318
ETH-Zentrum,	Fax:	+41 1 262 3973
CH-8092 Zürich	E-mail:	szyperski@inf.ethz.ch
Switzerland		
Andrew Watson	Status:	☺ □
APM Ltd	Tel:	+44 223 323010
Poseidon House	Fax:	+44 223 359779
Castle Park	E-mail:	ajw@ansa.co.uk
Cambridge CB3 ORD		
UK		
Alan Wills	Status:	☺ □
Department of Computer Science	Tel:	+44 61 275 6135
The University of Manchester	Fax:	
Manchester M13 9PL	E-mail:	alan@cs.man.ac.uk
United Kingdom		