Foundations of Object-Oriented Languages*

Workshop Report

Andrew Black† Jens Palsberg‡

1 Introduction

This paper reports on the workshop on foundations of object-oriented languages that was held 17–18 October 1993 at Stanford University, California, USA. The workshop was organized by Kim Bruce and Giuseppe Longo, and sponsored by ESPRIT and NSF. Local arrangements were done by Dinesh Katiyar and John Mitchell. Participation was by invitation only; the participants are listed in the appendix of this paper.

The purpose of the workshop was both to understand and compare the many models of object-oriented languages, and to create better language constructs and models. The workshop consisted both of presentations and lively discussions about such topics as challenge problems for type systems, relative merits of different language designs and type systems, and encapsulation and modularity.

In the following section we briefly report on the talks given at the workshop. In Section 3 we present a dictionary of object-oriented terminology that was created during the discussions. Finally, in Section 4 we report on the discussions by listing the open problems and benchmarks for type systems that were suggested at the workshop.

2 Presentations

Fifteen of the participants presented recent work or commented on the state of the field. Here follow brief summaries of their talks.

Luca Cardelli and Martín Abadi: *A theory of primitive objects.*

Cardelli and Abadi jointly presented a new formal system that supports both subsumption and override. *Subsumption* is the ability to use a "more powerful" or "more complete" object

*To appear in ACM SIGPLAN Notices.

†Digital Equipment Corporation, Cambridge Research Laboratory, 1 Kendall Square, Building 700, Cambridge, Massachusetts 02139, Internet: black@crl.dec.com.

‡Computer Science Department, Aarhus University, Ny Munkegade, DK–8000 Aarhus C, Denmark; currently visiting 161 Cullinane Hall, College of Computer Science, Northeastern University, 360 Huntington Avenue, Boston, Massachusetts 02115, Internet: palsberg@ccs.neu.edu.

in place of a less powerful or less complete object. *Override* is the ability to replace one method implementation by another. These features are both essential to object-oriented languages, but often they are combined in ways that are unsound, unclear, confused, or ad hoc, or they are not combined at all.

The system described in this talk departed from the authors' previous work by starting not from the λ-calculus, but from a new *object calculus*. Just as objects subsume functions, this new object calculus subsumes the λ-calculus. The system does not support a notion of state.

The system supports shallow subtyping: an object of a subtype can have more methods than objects of its supertype, but the types of the arguments and results of the common methods must be identical. Encoding the object calculus in a recognized typed λ-calculus in a way that preserves the subtypings remains an open problem.

A draft of a 90+ page paper presenting this theory can be obtained by electronic mail from the authors [1]. Shorter versions are to appear as [3, 2].

Giorgio Ghelli: *Objects with rôles.*

Ghelli described *rôles*, a concept from the Fibonacci database programming language. The language has a notion of state and a concept of object identity; it also allows *existing* objects to be extended. For example, if *John* is an object (representing a person), when John enrolls as a student the object might be extended to support new behavior, such as a method *Number* that returns his student number. Let us call this extended object *JohnAsStudent*.

In Fibonacci, *John* and *JohnAsStudent* are considered to be the same object by the object identity test. Nevertheless, *JohnAsStudent* and *John* have different behaviors: the first understands the *Number* message while the second does not.

It is also possible for John to take a part time job, requiring that the object that represents him be extended in another direction, creating *JohnAsEmployee*. Worse, *JohnAsEmployee* might also understand a *Number* method, which returns John's employee number.

Fibonacci deals with these complexities by treating an object as a DAG of rôles; messages are sent to a receiving role, which first tries delegating to subrôles, and then inheriting from superrôles. Thus, the meaning of *p.Number* depends on the rôle type of the identifier *p*.

Jens Palsberg: *Object-oriented type systems.*

Palsberg presented an overview of the theory of types that he has developed jointly with Michael Schwartzbach. The goal of the theory is to to explain and improve existing type systems, not to overthrow them. The work provides a common theory that can be used as a basis for comparisons between languages.

The theory deals with object-oriented languages directly, rather than via the λ-calculus. It follows the traditions of languages like SIMULA, C++, Eiffel, and BETA. Types are treated as sets of classes. Thus, the theory can deal with "class types" (the set of subclasses of a particular class) as well as with more abstract notions of type (the set of classes whose objects support a particular interface.)

The theory and its implications are presented in Palsberg and Schwartzbach's recently published book [14].

Ole Agesen: *Type inference for Self – why and how?*

Agesen described an application [4] of Palsberg and Schwartzbach's theory to the untyped programming language Self. Although Self does not have classes, a similar classification of objects can be created by the transitive closure of the "cloned" relation.

Agesen's system solves a number of practical problems faced by object-oriented programmers, including finding all of the places a particular message is sent to a particular object and its clones, and "tree shaking" (extracting an application from the programming environment).

Giuseppe Castagna: *Second order* ad hoc *polymorphism.*

Castagna presented joint work with Giorgio Ghelli and Giuseppe Longo dealing with generic function languages [7, 8]. Such languages treat messages as functions, and message sending as function application. Castagna, Ghelli, and Longo have developed a model for this approach called $\lambda\&$.

In $\lambda\&$, each function has many branches, each corresponding to a method in an object-oriented language. The branch chosen in response to an application depends on the class of the arguments. The binding is dynamic: it depends on the dynamic class of the argument objects, not the syntactic class of the argument expressions.

The class of a generic function is the set of the classes of the branches that constitute it. To ensure that there is always a "best fitting" branch for a particular invocation, there are some restrictions on the methods that can be combined in a generic function.

Benjamin Pierce: *Concurrent OO language design.*

Pierce described work in progress in Edinburgh to design an object-oriented concurrent language based on firm formal foundations [15, 16]. He is starting with the π-calculus, because this work is going on at Edinburgh, because the π-calculus is small and relatively simple, because it is theoretically well understood, and because it is computationally complete.

To this base he plans to add values, typing, higher-order programming, results and objects.

Values are structured collections of behaviors. Typing applies to channels; each channel has a type describing the shape of the values that it carries.

An object is a record of request channels for some server process. This approach provides selective enabling of methods *via* the guarded choice command of the π-calculus. Pierce believes that typing, subtyping and polymorphism should yield to standard techniques. Inheritance is more problematic, and is a topic for future work. Because there is no method body as such, just fragments of the process body that execute in response to a message, it is not clear what to inherit.

Robert van Gent: *TOIL: a type-safe object-oriented imperative language.*

Van Gent presented joint work with Kim Bruce on TOIL, an imperative object-oriented language that is provably type safe [6, 17]. In TOIL, methods in a subtype can have any subtype of the corresponding method of the supertype; this is deep subtyping (compare with Cardelli and Abadi's shallow subtyping). TOIL can be type-checked in a modular fashion, and inherited methods are type-safe.

TOIL adds several features to Bruce's previous work: references, **nil**, an identity test, and eager evaluation of values imported into objects.

Andrew Black: *What a language designer wants to know about types.*

Black took the position that the language designer is a *consumer* (of ideas), and the type theorist is a *vendor* (of type systems). A question of obvious importance is then: do the vendors understand the consumers? He also made the point that being in the type theory business is like being in the dog-food business: the consumer is not the same as the customer, and care must be taken to satisfy the needs of both.

Black emphasized that the designer of object-oriented languages wants the type theorists to work on objects right from the beginning. Objects subsume records, not the other way round. Similarly, although bounded polymorphism may be theoretically complex, it reflects the need of real programs to adapt to new objects.

Black also mentioned some practical solutions to real problems, such as finding the union of two types that contain methods with the same name but different numbers of arguments (by treating the number of arguments and results as part of the method name), and finding a typing for **nil**. He posed the question: if subtyping is undecidable in all interesting type systems, is there any reason to avoid **type:type**?

William Cook: *ADTs vs OOLs.*

Cook contrasted the concepts of Abstract Data Types (ADTs) and objects in Object-Oriented Languages (OOLs) [9]. He argued that although both are based on the idea of data abstraction, they are fundamentally different. In short, an ADT may be understood as a set *with* operations, whereas objects are sets *of* operations.

In an ADT, abstraction is achieved by hiding the type of the representation. This hiding is often modeled using an existential quantifier. In OOLs, abstraction is achieved by letting the operations themselves represent the data. The interfaces of the operations need not have any direct relation to the underlying representation at all.

One of the consequences of the above is that in an ADT, recursion is seldomly needed in the interface (although it may well be used in the representation). In OOLs, recursion is often essential to the interface. Thus, recursion should be central in models of OOLs.

Dinesh Katiyar: *Interfaces and typed OOP.*

Katiyar presented a type system featuring bounded existential quantification [10, 5, 13]. His experience with modeling systems confirmed the need for such types.

A bounded existential quantification may be understood as both hiding a type, and at the same time stating requirements on of what form the hidden type should be.

Roberto di Cosmo: *Which types for the objects?*

di Cosmo presented and criticized the idea of using a type as the *search key* for finding methods in a class library. The idea is that the search should return only those methods that have the given type. This would be helpful if we don't know the *name* of the method

we are searching for. He had analyzed a commercially available class library containing 2349 methods. Most methods required 0–2 arguments, and it appeared that insufficient functionality was expressed in the interfaces to effectively distinguish them. Thus, to provide efficient method retrieval tools, we may need to reconsider the notion of type for objects and methods.

Francois Bouladoux: *Inheritance and subtyping: Some problems and possible approaches.*

Bouladoux presented a refined approach to subtyping in a framework where objects are modeled as recursive records. In that framework, F-bounded quantification is a well-known approach to typing methods.

Bouladoux showed how to define a pre-order between methods, based on the F-bounds, and used that to refine the subtype ordering. With the refined subtyping relation, a method can be applied in more contexts than before.

Gary Leavens and K. Kishore Dhara: *A model theory for subtyping in imperative OOLs.*

Leavens and Dhara jointly presented work on specification and verification of object-oriented programs [12, 11]. The emphasis was on incomplete specifications and model theory. Once one considers specifications, the meaning of subtyping must be changed to include not just the existence the of corresponding methods, but also their behavior.

The general idea is that a type is described by an incomplete specification; a subtype may have a more complete specification, but it must not contradict any of the properties of the supertype. Leavens' previous work on subtyping uses explicit simulation to show that this property holds. The work presented here focused on subtyping in the presence of aliasing.

3 Terminology

Here follows the dictionary of object-oriented terminology that was created at the workshop. The explanations are not definitive; there were substantial disagreements during their creation.

- **object.** A primitive term.
- **message.** A name, a member of a countable set. In the invocation $x.f$, f is a message.
- **method.** A procedure inside an object. It is associated with a message (it has a name). It can be called by invoking the enclosing object with the corresponding name.
- **dynamic lookup.** A term expressing the idea that the method that executes in response to the invocation $x.f$ depends on the current value of x.
- **class.** A function returning objects; an object constructor.
- **inheritance.** A mechanism for making one class or type from another, in which **self** is late-bound.
- **delegation.** A mechanism for one thing to call another, in which **self** is late-bound. Like inheritance, but between objects directly, rather than between their classes.

- **subtyping (or conformance).** A relationship between types such that if S is a subtype of T, then Ss are substitutable for Ts, i.e., a value of type S may be used in any context that expects a value of type T.

- **matching.** The relationship that holds between two types when one is derived from the other by type inheritance; it is Bruce's $\leq_{meth}$. Also, it is the relation required to define F-bounded parameters.

4 Discussions

The following ways of using a type system were listed during the discussions:

- Partial documentation during program design.
- Checking for errors.
- Maintenance; "understanding what you've got".
- Efficiency; provides information for an optimizing compiler.

It is unlikely that one single type system will be equally useful for all four purposes.

The following benchmark problems emerged. The idea is that a type system should be able to type the programs suggested in this list.

- Point and ColorPoint, including equality methods.
- Church numerals.
- Calculators.
- Encoding of λ-calculus.
- Overriding of methods that return modified versions of self.
- Expression evaluators.
- Binary tree with search tree as subclass.
- Sorting of arbitrary (but homogeneous) objects, under the constraint that all the objects have a comparison method.
- Generic sort, in which $<$ is part of the argument list.
- A class for nodes in a singly linked list, and a subclass for nodes in a doubly linked list.

The many models of object-oriented languages focus on a wide variety of aspects of object-oriented programming. Often, two given models focus on rather different aspects and even when they focus on the same aspect, they may model different design choices. To guide comparison of various models, the following extremes in some design dimensions were identified:

- **Specification types versus implementation types.** Types may contain more or less implementation information. At one extreme we find types that only specify such information as type checking interfaces (specification types), and at the other extreme we find types that give the full implementation of the objects of a type (implementation types). Commonly used object-oriented languages such as C++ tend to use implementation types.

- **Structural subtyping versus subtyping by name.** In some languages, all subtyping relationships must be explicitly declared (subtyping by name). The alternative is to enable the compiler to deduce if two types are in the subtyping relationship (structural subtyping). The latter choice requires the subtyping relationship to be decidable. If so, it is more flexible, but perhaps less transparent for programmers.

- **Shallow versus deep subtyping.** In many type systems, types can be arbitrarily nested, and, in the presence of recursive types, infinite. When defining the subtyping relation $A \leq B$, we may for example want to allow A to have more components than B. Given this, the following question emerges: should common components be of the *same* type (shallow subtyping), or should we for example allow the one in A to be a subtype of the one in B, or perhaps the other way round (deep subtyping)? In most models, care is needed to ensure the soundness of the type system.

- **Single versus many implementations for each interface.** Some models separate the class hierarchy and the type hierarchy. If we consider a type as specifying an interface for objects, then classes provides the implementation of the objects. A basic design question is whether there can be only a single or several implementations for each interface.

- **Type inference versus type annotation.** With any type discipline comes the question of type inference. It is only partially understood how powerful type systems we can get and at the same time have computable type inference.

- **Closed world assumption versus open world assumption.** Type-checking of programs can happen under various assumptions. If the compiler can assume that all parts of the program is known (closed world assumption), then it might do a better a job than if different units are to be compiled separately (open world assumption). The open world assumption is especially useful when compiling libraries, and the closed world assumption appears to be most useful towards the end of program development where global analysis techniques may be helpful. The system-level type-checking that has been announced for Eiffel uses the closed world assumption.

- **Extension versus overriding**. Inheritance can be used both for extending classes with more instance variables and methods, and it can be used for method override. In some models, these two uses put rather different requirements on type systems.

- **Delegation versus inheritance.** Some models include both objects and classes, others just objects. As a consequence, the former models involve inheritance, while the latter

involve delegation. It is unclear if the same style of type systems can be used in both cases.

- **Object identity.** A key feature of many object-oriented languages is that objects retain their identity even when their local state changes, when they are put into secondary storage, and when they migrate to other computers. Only a few models so far faithfully model this.

- **Multi-methods: encapsulation, compositionality, extensibility, abstract types.** Most object-oriented languages are based on *single dispatch*. This means that the code to be executed in response to a message send is chosen on the basis of *one* object, the receiver. Other languages feature multi-methods and *multiple dispatch*. In such languages, the multi-method to be executed in response to a message send is determined on the basis of *several* objects. Languages with multi-methods tend to identify types with classes, and the employed lookup strategies seem to expose instance variables. It appears that the type systems required for single dispatch and multiple dispatch are rather different. Moreover, the models of multi-methods may help clarify the issues of encapsulation, compositionality, extensibility, and abstract types that are the subject of much experimentation in current object systems with multi-methods.

- **Meta-object protocols.** In recent years, some object systems have pioneered the idea of having a meta-object protocol. Only few models have attempted to model this construct, and it is unclear how well it can be integrated with a type system.

- **Concurrency.** Although there has been a plethora of work on concurrent object-oriented programming, no consensus has emerged on how to integrate objects, processes, and types in one framework.

Finally, the following list of open problems was compiled:

- ADT's versus objects. Can we translate back and forth? Issues: Extensibility versus hiding; abstraction versus efficiency; and subtyping and reuse.

- How can types be used in a programming-support methodology? How can they help one to clean up messy programs?

- Can we reduce objects to records? What basic record operations would be needed?

- Can and should we unify objects and processes?

- How can we extend ML with object-oriented features?

- Can we give an adjoint characterization of functions and objects?

In conclusion, the workshop helped clarify many issues and provided plenty of input for further research.

Acknowledgements

The authors thank Kim Bruce, Guiseppe Longo, and Mitchell Wand for helpful comments on drafts of the report.

Appendix: Participants

Martín Abadi	DEC, SRC	ma@src.dec.com
Ole Agesen	Stanford University/SUN	agesen@cs.stanford.edu
Roberto Bellucci	ENS, Paris	bellucci@dmi.ens.fr
Andrew Black	DEC, CRL	black@crl.dec.com
Francois Bouladoux	ENS, Paris	bouladou@dmi.ens.fr
Kim Bruce	Williams College	kim@cs.williams.edu
Luca Cardelli	DEC, SRC	luca@src.dec.com
Giuseppe Castagna	ENS, Paris	castagna@dmi.ens.fr
William Cook	Apple	william@applelink.apple.com
Roberto di Cosmo	ENS, Paris	dicosmo@dmi.ens.fr
K. Kishore Dhara	Iowa State University	dhara@cs.iastate.edu
Kathleen Fisher	Stanford University	kfisher@theory.stanford.edu
Robert van Gent	Stanford University	vangent@cs.stanford.edu
Giorgio Ghelli	University of Pisa	ghelli@di.unipi.it
Dinesh Katiyar	Stanford University	katiyar@theory.stanford.edu
Gary Leavens	Iowa State University	leavens@cs.iastate.edu
Giuseppe Longo	ENS, Paris	longo@dmi.ens.fr
John Mitchell	Stanford University	jcm@theory.stanford.edu
Eugenio Moggi	University of Genova	moggi%igecuniv@vm.cnuce.cnr.it
Jens Palsberg	Aarhus Univ./Northeastern Univ.	palsberg@ccs.neu.edu
Benjamin Pierce	Edinburgh	bcp@dcs.ed.ac.uk
Mitchell Wand	Northeastern University	wand@ccs.neu.edu
Phil Yelland	ParcPlace	yelland@parcplace.com

References

[1] Martín Abadi and Luca Cardelli. A theory of primitive objects. Manuscript, October 1993.

[2] Martín Abadi and Luca Cardelli. A theory of primitive objects: Second-order systems. In *Proc. ESOP'94, European Symposium on Programming*. Springer-Verlag, 1994. To appear.

[3] Martín Abadi and Luca Cardelli. A theory of primitive objects: Untyped and first-order systems. In *Proc. TACS'94, Theoretical Aspects of Computing Sofware*. Springer-Verlag, 1994. To appear.

[4] Ole Agesen, Jens Palsberg, and Michael I. Schwartzbach. Type inference of Self: Analysis of objects with dynamic and multiple inheritance. In *Proc. ECOOP'93, Seventh European Conference on Object-Oriented Programming*, pages 247–267, Kaiserslautern, Germany, July 1993.

[5] Kim Bruce and John C. Mitchell. PER models of subtyping, recursive types and higher-order polymorphism. In *Nineteenth Annual ACM SIGACT-SIGPLAN Symposium on Principles of Programming Languages. Albuquerque, New Mexico*, pages 316–327, January 1992.

[6] Kim B. Bruce and Robert van Gent. TOIL: A new type-safe object-oriented imperative language. To appear, 1993.

[7] Giuseppe Castagna, Giorgio Ghelli, and Giuseppe Longo. A calculus for overloaded functions with subtyping. *Information and Computation*. To appear.

[8] Guiseppe Castagna. $F^{\&}_{\leq}$: Integrating parametric and "ad hoc" second order polymorphism. In *Proc. 4th International Workshop on Database Programming Languages*, pages 335–355. Springer-Verlag, 1993.

[9] William Cook. Object-oriented programming versus abstract data types. In *Proc. REX Workshop/School on the Foundations of Object-Oriented Languages*. Springer-Verlag (*LNCS* 489), 1990.

[10] Dinesh Katiyar, David Luckham, and John Mitchell. A type system for prototyping languages. In *Proc. POPL'94, 21st Annual Symposium on Principles of Programming Languages*, 1994. To appear. Available by anonymous ftp from theory.stanford.edu:pub/katiyar/papers/popl-94.dvi.Z.

[11] Gary T. Leavens and Krishna Kishore Dhara. A model theory for abstract data types with mutable objects (extended abstract). Technical Report 93-21, Department of Computer Science, Iowa State University, September 1993. Available by anonymous ftp from ftp.cs.iastate.edu, and by e-mail from almanac@cs.iastate.edu.

[12] Gary T. Leavens and William E. Weihl. Subtyping, modular specification, and modular verification for applicative object-oriented programs. Technical Report 92-28b, Department of Computer Science, Iowa State University, October 1993. Available by anonymous ftp from ftp.cs.iastate.edu, and by e-mail from almanac@cs.iastate.edu.

[13] J. C. Mitchell, S. Meldal, and N. Madhav. An extension of Standard ML modules with subtyping and inheritance. In *Eighteenth Symposium on Principles of Programming Languages*, pages 316–327. ACM Press, January 1991.

[14] Jens Palsberg and Michael I. Schwartzbach. *Object-Oriented Type Systems*. John Wiley & Sons, 1993.

[15] Benjamin C. Pierce, Didier Rémy, and David N. Turner. A typed higher-order programming language based on the pi-calculus. Draft report; an earlier version was presented as an invited lecture at the Workshop on Type Theory and its Application to Computer Systems, Kyoto University, July 1993, July 1993.

[16] Benjamin C. Pierce and David N. Turner. Simple type-theoretic foundations for object-oriented programming. *Journal of Functional Programming*. To appear. A preliminary version appeared in Principles of Programming Languages, 1993, and as University of Edinburgh technical report ECS-LFCS-92-225, under the title "Object-Oriented Programming Without Recursive Types".

[17] Robert van Gent. TOIL: An imperative type-safe object-oriented language. Williams College Senior Honors Thesis, 1993.