The Semantic Challenge of Object-Oriented Programming

Dagstuhl Seminar 98261

28 June — 3 July 1998

Organisers: Luca Cardelli
Achim Jung
Peter O'Hearn
Jens Palsberg

Report editor: Ian Stark

Preface

Object-oriented programming is based on an informal concept of object as an entity or thing whose identity persists over time. The object concept is immediately meaningful to programmers, and has proven to be a useful and flexible organisational device in the analysis, design, and maintenance of complex systems. But though objects are attractively simple and intuitive in their initial conception, programming languages that support object-orientation are subtle and pose significant challenges for researchers.

Research on the Foundations of OOP began in the mid eighties. Many of the first inspirations came from denotation models, but it was quickly realised that existing semantic technology was not adequate to meet the demands of the object paradigm. For this reason, a number of researchers adopted a more ad-hoc (but more effective in the short term) operational and type-theoretical approach. This has resulted in significant achievements, the most visible of which are calculi and safe type systems for OOP, which simplify and deal soundly with intricate issues involving inheritance. This lead to important semantics work on polymorphism and subtyping.

In the meantime there have been substantial advances in more "mainstream" semantics. Some of this has been directly relevant to or inspired by OOP, especially work on modelling subtypes and polymorphism. But there have also been substantial new advances using game semantics, functor categories, and process calculi. This has lead, for example, to a better understanding of local state and interaction, both of which are integral to the essence of the object concept.

The purpose of this workshop was to bring researchers from the two camps together. On one hand, OOP provides a great challenge for current semantic methods, and attempting to apply them will likely bring up new problems and give new insight on the methods themselves. On the other hand, a deeper semantic analysis of object-oriented languages can potentially impact program specification, type systems, and static analysis.

In addition to the talks presented at the workshop (abstracted below), there were numerous lively "corridor discussions", and a wrap-up session, where the main themes and problems were discussed. Some of the central problems or issues that repeatedly arose included the following.

Objects versus Functions There are a number of translations from object-oriented to functional programming. Some researchers stressed, however, that these translations are somewhat indirect, and (especially) have difficulty explaining the typing concerns prevalent in OOP. For this reason, and also because translations go both ways, primitive ob-

ject calculi are gaining currency, where translations to (pure or impure) functional programming are viewed as complementing the calculi, not replacing them.

State, Extrusion, and Identity Extrusion occurs when an object passes a newly created object to the outside; this occurs frequently in OOP (and even forms the basis of Hewitt's Actor model). Purely functional object calculi do not give an accurate account of extrusion, however, because when the extruded entity is changed by the receiver, this change needs to be reflected in the object itself: the distinction between sharing and copying is crucial here. But while "adequate" or sound models of extrusion can easily be given using traditional methods, more accurate semantics, accounting for locality, presents a challenge.

Specification Specification methods for objects remain a challenge. Particular problems include abstraction and inheritance, and imperative behaviour.

Also, while progress in type systems represents perhaps the most significant advances in Foundations of OOP, and has formed the core of prior work, interest continues and further work is progressing on a number of fronts.

Peter O'Hearn

Participants

Stephen Brookes, Carnegie Mellon University
Kim Bruce, Williams College
Luca Cardelli, Microsoft Research
Pietro Cenciarelli, Ludwig-Maximilians-Universität München
Frank de Boer, University of Utrecht
Mariangiola Dezani-Ciancaglini, Università di Torino
Kathleen Fisher, AT&T Labs-Research
Giorgio Ghelli, Università degli studi di Pisa
Dan Ghica, IBM Toronto Laboratory
Bart Jacobs, University of Nijmegen
C. Barry Jay, University of Technology, Sydney
Achim Jung, University of Birmingham
Søren Bogh Lassen, Cambridge University
K. Rustan M. Leino, Compaq Systems Research Center
Peter O'Hearn, Queen Mary and Westfield College
Jens Palsberg, Purdue University
Benjamin Pierce, University of Pennsylvania
Erik Poll, University of Kent
Uday Reddy, University of Illinois
Didier Remy, INRIA-Rocquencourt
John Reynolds, Carnegie Mellon University
Jon Riecke, Bell Laboratories
Scott Smith, John Hopkins University
Ian Stark, University of Edinburgh
Robert D. Tennent, Queens University
Hayo Thielecke, Queen Mary and Westfield College

Further information on the seminar and participants is available electronically at http://www.dag.uni-sb.de/DATA/Seminars/98/#98261

Contents

Abstracts

Some Distinctions without a Difference?

Benjamin C. Pierce
University of Pennsylvania

In the spirit of the them of *duality* introduced by Erik Poll, this will be a "co-talk": I will ask questions and the audience will, I hope, help with some answers. In particular I would like to mention three puzzles arising in type systems for OO languages:

- the relation between the "by-name" type systems found in most mainstream OO languages and the "structural" type systems of typed lambda calculi;
- the striking similarities (striking in view of their evident fundamental differences) between the "recursive records of functions" model of objects and the "overloaded methods" model of CLOS and related languages; and
- the question of formalising the melange of ADT-like and "pure object" qualities in the classes of Java, C++ and Simula.

Everything is an Object

Luca Cardelli
Microsoft Research

(Joint work with Martin Abadi.)

I discuss some foundational issues in object-oriented programming. Object-based (as opposed to class-based) languages have attempted to explore simpler object-oriented foundations. Recent results validate the long-standing intuition, embedded in object-based languages, that *everything* including functions and classes can be represented as objects. Moreover (and this was *not* a long-standing intuition) function types and class types can be represented as object types. The basic constructions are simple, flexible and powerful, reinforcing some people's belief that object-based languages form an alternative and simpler foundation for object-oriented programming.

Event-Based Operational Semantics of Java

Pietro Cenciarelli
Universität München

(Joint work with A. Knapp, B. Rens and M. Wirsing.)

A structural operational semantics of Java is presented; including the mechanisms for running and stopping threads, thread interaction via shared memory, synchronization via monitoring and notification, and sequential control mechanisms such as exception handling and return statements.

The operational semantics is parametric in the notion of *event space*, which formalises the rules that threads and memory must obey in their interaction. Different computational models are obtained by modifying the well-formedness conditions on event spaces while leaving the operational rules untouched. In particular, we implement the *prescient stores* described in the Java specification, which allow certain intermediate code optimizations, and prove that such stores do not affect the semantics of properly synchronized programs.

Reasoning about Java Classes

Bart Jacobs
University of Nijmegen, The Netherlands

(Joint work with Joachim van den Berg, Marieke Huisman, Martijn van Berkum, Ulrich Hensel and Hendrick Tews.)

First results are presented of a project on formal methods for the object-oriented language Java. It aims at verification of program, properties, with support of modern tools. We use our own front-end tool (which is still under construction) for translating Java classes into logic, and a back-end theorem proven (namely PVS) for reasoning. In several examples we will demonstrate how non-trivial properties of your programs and classes can be proved following this two step approach.

Reasoning about Dynamically Evolving Object-Structures

F. de Boer
University of Utrecht, The Netherlands

I presented a Hoare-logic for a simple (i.e. without sub-typing and inheritance) OO-language which allows one to reason about object-structures at an abstraction level which is at least as high as that of the programming language. This means that the only operations on 'pointers' are equality and

dereferencing. Moreover, in a given state of the system, it is only possible to measure the objects that exist in that state. Objects that do not (yet) exist never play a role. Axiomatizations are given for aliasing object-creation and method invocation. Extensions to concurrent system have been briefly indicated. Future work: sub-typing, inheritance and abstract types.

Extensible, Incomplete Objects

Mariangiola Dezani-Ciancaglini
Università di Torino

(Joint work with Viviana Bono, Michele Bugliesi and Luigi Liquori.)

The type system for the Lambda Calculus of Objects is extended to account for a notion of width subtyping. The main novelties over previous work are the use of subtype-bounded quantification to capture a new and more direct encoding of *MyType* polymorphism, and a treatment of other features that were accounted for via different systems in subsequent extensions of the original proposal. The new system provides for

- approximate type specialization of inherited methods;
- static detection of errors;
- width subtyping compatible with object extension;
- sound typing for partially specified objects.

Privacy via Subsumption

Jon Riecke
Bell Laboratories, Lucent Technologies

(Joint work with Christopher Stone, CMU.)

We describe an object calculus allowing object extension and structural subtyping. Each object has a "dictionary" to mediate the connection between names and components. This extra indirection yields the first object calculus combining both object extension and full width subtyping in a type-safe manner. If classes are modelled through object extension, private fields and methods can be achieved directly by scoping restrictions: private fields or methods are those hidden by subsumption. We prove that the type system is sound, discuss a variant allowing covariant self-types, and give some examples of the expressiveness of the calculus.

The Essence of Object-Oriented Program Semantics

K. Rustan M. Leino
Compaq Systems Research Center

I give a semantics for an imperative, object-oriented programming notation. The semantics is given in terms of what we already know about the weakest-precondition and refinement semantics of imperative, procedural languages, which I review in the talk.

From Classes to Objects via Subtyping

Didier Remy
INRIA — Rocquencourt

We extend the Abadi-Cardelli calculus of primitive objects with object extension. We enrich object types with a more precise, uniform and flexible type structure. This enables us to type object extension under both width and depth subtyping. Objects may also have extended-only or virtual contravariant methods and read-only co-variant methods. The resulting subtyping relation is richer, and types of objects can be weakened progressively from a class level to a more traditional object level along the subtype relationship.

Adding Classes to a Language with Modules

Kathleen Fisher
AT&T Labs — Research

We believe that modern programming languages require both module systems and class mechanisms. Unfortunately, current class constructs overlap significantly with module features, causing undue complexity. We describe the design of a simple class mechanism that works in concert with the module system, to support a rich class-based programming style. The minimal class construct is part of MOBY, an ML-like language with support for object-oriented features. We formalize MOBY in MiniMOBY, a core language that captures the essence of our design. We validate the expressiveness of our language by showing how various programming idioms from class-based languages can be written in MOBY.

Parameters and Linked Structures in Algol-like Languages

Dan Ghica
IBM, Toronto

A mechanism of generating dynamic variables inspired from the ν-calculus

(Stark & Pitts) is adapted to an Algol-like language with local variables and extended to support assignment. A functor category semantics is given and several typical equivalences are proved. The notions of isomorphism and observability are introduced, and it is proved that isomorphism implies observational equivalence, leading to an updated semantic of state. More equivalences are proved this way.

Concurrent Objects in Idealised CSP

Stephen Brookes
Carnegie Mellon University

Idealised CSP in an Algol-like language combining asynchronous communicating process with a simply typed λ-calculus. The language supports a form of concurrent object-oriented programming, in which an object is represented by a collection of local variables and local elements together with a list of procedures for accessing this local data. These "methods" can be executed concurrently and objects can be designed so as to ensure desirable properties such as mutually exclusive access to an object's data. The language has a straightforward semantics based on "transition traces", which builds in the assumption of fair execution. The semantics validates a number of useful laws of program equivalence, leading to a methodology for reasoning about the behaviour of concurrent objects.

(Co)Inductive Types with Sub-typing

Erik Poll
University of Kent

Inductive (or algebraic) datatypes are well known, for instance in functional programming languages such as ML or Haskell. I will consider a simply typed λ-calculus with not only inductive types, but also their duals — co-inductive types — and subtyping. These conductive types can be used as types for objects, which provides subtyping for object types and a form of code reuse that corresponds to a simple form of inheritance. (This is essentially just another syntax for existing object encodings).

Exploring the duality between inductive and conductive types now suggests a notion of subtyping for inductive types and an associated form of code reuse which I call *coinheritance*.

When coinductive types are used as object types, object types are just interfaces (or signatures) and subtyping is so-called structural subtyping. I will show how this approach can be refined to provide "classes" as types of objects, and a notion of behavioural subtyping for those classes.

Grouping Constructs of Semantics in Object-Oriented Languages

Kim Bruce
Williams College

We review the semantics of class-based object-oriented languages (exemplified by our language LOOM) in order to explain how typing rules and the notion of matching flow directly from the semantics. The semantics of OO languages with *MyType* generalize nicely to groups of mutually reclusive object types. Until now, the only way of handling inheritance of these groups is to use constructs like Beta's virtual classes. Unfortunately virtual classes seem to require dynamic type checking. We show how the generalized semantics leads to straightforward static typing rules via collections of *MyType*-like variables that are visible to all object types in the group.

When is a Local Variable not a Local Variable?

Ian Stark
University of Edinburgh

This talk presents a method for reasoning about the interaction between first-class recursive functions and local variables. This combination of features is found in Standard ML and allows a number of useful program idioms involving hidden or shared state: however it can also cause problems of aliasing and interference.

The basis for our reasoning is a novel logical relations that acts not just between expressions but also the contexts in which they may be used. The relation is defined inductively over types and gives a complete operational characterisation of contextual equivalence. From the logical relation we derive a practical proof method of *local invariants*, which turns informal intuition about why programs should be equivalent into a proof that they are. This part is joint work with Andrew Pitts and can be found in the paper *Operational Reasoning for Functions with Local State* (Higher Order Operational Techniques in Semantics, Gordon and Pitts CUP 1998).

Interesting, all the examples given fit the idiom of object-based programming; the tricky interaction is now that between private instance variables and first-class objects. We propose that the same technology of local invariants might usefully be adapted to treat object-oriented programming: in particular to the issue of reasoning about code that involves methods of unknown foreign objects. Hence the alternative title for the talk: *How to Deal with Objects who Steal your Instance Variables.*

Bisimulation up to Context for Sequential Higher-Order Languages

S. B. Lassen
University of Cambridge, Computer Laboratory

Bisimulation up to context is a syntactic method which is useful for reasoning about contextual equivalence in higher-order sequential languages. In the talk I demonstrated how it could be used to reason about local state and dynamic references as a hybrid approach between applicative bisimulation and logical relations.

A FISH Language Demonstration

C. Barry Jay
University of Technology, Sydney

Objects with Roles

Giorgio Ghelli
Pisa University, Computer Science Department

In object-oriented database languages, it is essential to be able to extend an object in different, unrelated ways. In doing this, we run the risk of extending the object in non-compatible ways. This problem can be faced by extending objects with a notion of "role": an object may play different roles, and the way it answers messages depends on the role it is accessed through. We present a formalization of this notion, and how it is related to object extension.