Semantics of Imperative Objects

Catalin Hritcu

Supervisor: *Jan Schwinghammer*
Responsible: *Prof. Gert Smolka*
Programming Systems Lab
Saarland University, Saarbrücken

November 2006

Programming Languages

- Syntax (how programs look like)
  - Context-free grammars
  - Abstract syntax trees
- Semantics (the meaning of programs)
  - Operational, denotational, axiomatic semantics
  - Type systems
  - Step-indexed semantics

Object Calculi

- Object-oriented programming languages
- Idealized models
- Rigorously defined semantics
- Simple - only one primitive: objects
- Expressive
  - can encode: classes and inheritance
  - but also functions (the lambda calculus)

The object calculi are object-oriented programming languages. They are abstractions (idealized morels) which capture the essence of object-oriented programming languages.

They have rigorously defined **formal semantics**

They are simple since **only objects are considered as primitives**.

At the same time, they are **expressive** enough to **encode** all common features of practical object-oriented programming languages like **classes** and **inheritance**.

They can also **encode functions** – the **lambda calculus** can be easily encoded.

Object Calculi

- Object-based (not class-based)
- Strongly-typed
- A Theory of Objects [Abadi & Cardelli '96]
- Object-based in practice: JavaScript, Self, etc.

Since the object calculi only have objects as primitives they are called **object-based** in contrast to the much more widely used **class-based** programming languages like Java or C#.

They are **typed**. For example you cannot update a boolean field with a numeric value.

These calculi are described in the book by Abadi and Cardelli – **A Theory of Objects**

Not very popular in practice:
Quote by Backes: "Nice construction, very nice theoretical properties ... never used in practice".

With only one exception: JavaScript
-> the J in AJAX is for JavaScript – so there is still hype about object-based object-oriented programming languages in the context of rich web applications (also ActionScript – Flash)
-> in the upcoming 2.0 version JavaScript will also get classes, still just as an encoding

[Mention Scala? Not here]

Imperative Object Calculus

- Variant of the object calculus [Abadi & Leino, '04]
- Syntax

a, b	$::=$	x	variable
	$\mid$	let $x = a$ in b	variable binding
	$\mid$	$[f_i = x_i, m_j = \varsigma(y_j)b_j]_{i \in I, j \in J}$	object construction
	$\mid$	$x.f$	field selection
	$\mid$	$x.f := y$	field update
	$\mid$	$x.m$	method call
	$\mid$	true $\mid$ false	boolean constants
	$\mid$	if x then a else b	conditional
	$\mid$	$0 \mid$ succ $\mid \ldots$	numbers

- More syntactic sugar used in examples

In this I will talk about a ariant of Abadi and Cardelli's imperative object calculus, as presented by Abadi and Leino. So what is **imperative**?
- computations are described in terms of a **program state** (we will call **store**)
- statements can read and from the store and write to the store
[- side effects are the rule (C) not the exception (ML)]

Programs are **flat** -> this makes **evaluation** order explicit - given by the **let** constructs.

The **self reference** can be used **used for direct recursion** - same role as **this** in Java, C++, C#
Methods don't have arguments, but we can use the fields (in a similar way to the encoding of lambda calc.)
We have **update for fields** but **not for methods** [-> method updates can be easily be simulated]

Other than the objects we also have **booleans** and **numbers** which we will use in the examples.
In the examples we will also use a lot of **syntactic sugar**.

Example Programs

- Factorial

$$[fac = \varsigma(y)\lambda n.\text{if } n = 0 \text{ then } 1 \text{ else } n \times y.fac(n-1)]$$

- Euclid's gcd algorithm

$$\begin{aligned}[gcd = \varsigma(y)\lambda x.\lambda z.&\text{if } x < z \text{ then } y.gcd\ x\ (z - x) \\ &\text{else if } z < x \text{ then } y.gcd\ (x - z)\ z \text{ else } x]\end{aligned}$$

Operational Semantics

- Evaluate the program to get its meaning
- Abstract machine
- E.g. states of the machine = program + store
- Suffices for building a simple interpreter
- Simple, flexible and widely used
- Verification by testing, cannot be complete

What is the operational semantics of a programming language?
- As already said, a semantics is a way to **assign a meaning to programs**.
- In the case of **operational semantics** the meaning of a program is the result obtained by running (**evaluating**) it.

Defining an operational semantics means defining a simple **abstract machine**, for which the language you want to investigate is the machine code.

The **states** of the abstract machine could be for example a **program** together with the current **store**.

An operational semantics alone **suffices for** building a **simple interpreter** for a language.

It is also probably the **most widely used** kind of semantics, because of its **simplicity** and **flexibility**.

When you only have an interpreter for the language the only **verification** you can do is **by testing**, and while testing can show the presence of bugs it cannot show their absence, so testing can never be complete.

Operational Semantics

- Small-step: evaluation = $\rightarrow^*$
- Reduction Relation: $\rightarrow \subseteq \mathbb{C}\text{onfig} \times \mathbb{C}\text{onfig}$
- Configurations: $\langle \sigma, a \rangle \in \mathbb{C}\text{onfig} = \mathbb{S}\text{tore} \times \mathbb{L}\text{Prog}$
- Reduction defined by rules

$$(\textsc{Red-Let1})\ \frac{\langle \sigma, a \rangle \rightarrow \langle \sigma', a' \rangle}{\langle \sigma, \text{let } x = a \text{ in } b \rangle \rightarrow \langle \sigma', \text{let } x = a' \text{ in } b \rangle}$$

$$(\textsc{Red-Let2})\ \langle \sigma, \text{let } x = v \text{ in } b \rangle \rightarrow \langle \sigma, b[x \mapsto v] \rangle$$

$$(\textsc{Rel-Obj})\ \frac{l \notin dom\ \sigma \quad o = [f_i = v_i, m_j = b_j[y_j \mapsto l]]_{i \in I, j \in J}}{\langle \sigma, [f_i = v_i, m_j = \varsigma(y_j) b_j]_{i \in I, j \in J} \rangle \rightarrow \langle \sigma[l \mapsto o], l \rangle}$$

Let us discuss the operational semantics of our imperative object calculus.

In particular what you see is a **small-step semantics**. In a small-step semantics evaluation is defined with respect to a **reduction relation** which define **transitions** (steps) between two configurations of the abstract machine.

The **configurations** of our abstract machine are just pairs formed by a store and a partially evaluated program.

Reduction relation is inductively defined by rules.

Example Reduction

$$\langle \emptyset, \text{let } y \ = \ [m = \varsigma(x)x.m] \text{ in } y.m\rangle \rightarrow$$
$$\langle l = [m = l.m], \text{let } y \ = \ l \text{ in } y.m\rangle \rightarrow$$
$$\langle l = [m = l.m], l.m\rangle \rightarrow$$
$$\langle l = [m = l.m], l.m\rangle \rightarrow \ldots$$

Nonterminating evaluation

This program defines an object with a method that calls itself and then calls that method.

Type Systems

- Static program analysis technique
- Conservative (sound but incomplete)
- In general:
  - Limited to safety properties
  - Limited form of reasoning about programs
  - Efficiently decidable (syntax driven)
- We will see an exception soon
- Types and Programming Languages [Pierce, '02]

What are type systems? A static program analysis technique

Conservative (sound but incomplete) – contrast to other techniques like model checking.
In general limited to safety properties.

They also provide a limited way of reasoning about programs – the type of a procedure gives you some information about what the procedure does.

We usually have these **limitations** because we want type checking to be **efficiently decidable** (actually most type checking algorithms are syntax driven so they are extremely fast).

However we will soon see an **exception**, a type system for program correctness (very strong property) that is **undecidable**.

The best **reference** for type systems is the book of Benjamin **Pierce**: Types and Programming Languages.

Types of Objects

Conditional

$$\frac{E \vdash x : Bool \qquad E \vdash a_0 : A \qquad E \vdash a_1 : A}{E \vdash \mathit{if}\ x\ \mathit{then}\ a_0\ \mathit{else}\ a_1 : A}$$

Let

$$\frac{E \vdash a : A \qquad E, x{:}\, A \vdash b : B}{E \vdash \mathit{let}\ x = a\ \mathit{in}\ b : B}$$

Object construction for $A \stackrel{syn}{=} [\mathrm{f}_i{:}\, A_i\ ^{i \in 1..n},\ \mathrm{m}_j{:}\, B_j\ ^{j \in 1..m}]$

$$\frac{E \vdash \diamond \qquad E \vdash x_i : A_i\ ^{i \in 1..n} \qquad E, y_j{:}\, A \vdash b_j : B_j\ ^{j \in 1..m}}{E \vdash [\mathrm{f}_i = x_i\ ^{i \in 1..n},\ \mathrm{m}_j = \varsigma(y_j) b_j\ ^{j \in 1..m}] : A}$$

Field selection

$$\frac{E \vdash x : [\mathrm{f}{:}\, A]}{E \vdash x.\mathrm{f} : A}$$

- Simple types: $A, B ::= Bool \mid Nat \mid [f : A, m : B]$
- Types of objects can be extended to specifications

Simple types: Booleans,
The types of objects can be **extended** to **specifications** of **program correctness** – which brings us to our next kind of semantics: **axiomatic semantics.**

Axiomatic Semantics

- Program meaning is what can be proved about it
- Focus on reasoning about program behavior
- Program logic
  - Deduction system for program correctness
  - E.g. Hoare logic [Floyd, '67] [Hoare, '69]

$$\frac{\vdash \{p \wedge b\} S_1 \{q\} \qquad \vdash \{p \wedge \neg b\} S_2 \{q\}}{\vdash \{p\} \text{if } b \text{ then } S_1 \text{ else } S_2 \{q\}}$$

- Widely used in program verification (E.g. Verisoft)

In an **axiomatic semantics** the meaning of a program is whatever can be **proved** about it.

The focus is on **reasoning about program behavior** [expressed as an input output behavior] using a **program logic**. A program logic is a **deduction system** for program **correctness**. The best example for a program logic is **Hoare logic**.

Explain rule: read from bottom up, triples: precondition, statement, postcondition

Program logics are still widely used in **program verification**, for example the Verisoft project uses **Hoare logic**.

The Logic of Objects

- [Abadi & Leino, '04]
- A (undecidable) refinement of the type system
- Specifications generalize types

$$E \vdash b : A \rightsquigarrow E \vdash b : A :: \varphi$$

- φ is a first-order logic formula
- Seems hard to partially mechanize [Tang, '01]
- Soundness hard to prove
  [Schwinghammer & Reus, '06]

The Logic of Objects defined by Abadi and Leino is a refinement of the type system we have seen. But since we are now talking about program correctness in the logic type checking no longer decidable.

Specifications generalize types.

where phi is a first-order logic formula talking about the store.

The logic seems hard to partially mechanize – building a verification condition generator was done by [Tang, '01].

And soundness for the proof rules is hard to prove [Schwinghammer, '06]

Soundness for it was hard to prove

Higher-order Store

- Executable code can be stored
  - Pointers to functions in C
  - Callbacks in Java
  - General references in ML
- Recursion by "tying a knot in the store"

What makes the imperative object calculus particularly interesting from a theoretical point ow view, is that it combines **objects** with **higher-order store**.

Higher-order store = **executable code can be stored**

Higher-order store is present in different forms **in almost all practical programming languages**:
-> pointers to functions in C/C++
-> callbacks in Java
-> or general references in ML

And it's easy to check that you have higher order store by implementing **recursion through the store**. Also called, **tying a knot in the store**.

Higher-order Store

- Imperative object calculus
  - Dynamically allocated, higher-order store
  - Challenge: finding good semantic models
    - Reasoning about the behavior of programs
    - Program correctness (using a program logic)
- Operational Semantics
  - Useful for proving safety(progress&preservation)
  - Not suitable as the basis for a program logic [Benton, '05] [Benton, '06] [Schwinghammer, '06]

The imperative object calculus features: **dynamically allocated, higher-order store.**

Challenging to find **good semantic models** in which one can **reason about the behavior of programs**.

Syntactic arguments, based solely on the **operational semantics**, are surelly enough to prove properties such as type safety, but are **not suitable as a basis for program logics** like the one we just discussed.

We believe that **specifications** should have a **meaning independent** of the particular proof system.

Denotational Semantics

- The meaning of a program is a mathematical object
- E.g. denotation of a lambda abstraction = function
- Higher-order store
  - Solving recursive domain equations
- Dynamic Allocation
  - Possible-world model =
    category of functors over cpos
- Domain theory, category theory, order theory
- More abstract - reasoning about program behavior

What is a denotational semantic? The **meaning** of a program is a **mathematical object**.

For **example** in the simply typed lambda calculus the meaning of a **lambda abstraction** is a **function**. However, once one starts to study more realistic programing languages things are not so simple any more.

A denotational semantics for dynamically-allocated higher-order store tends to become rather **complex**.

For **higher-order store** the semantic domain is defined as mixed-variant **recursive equation** one has to solve (find fixed point?).

Also only for modeling **dynamic allocation** means one has to move to a **possible-world model**, formalized as a category of functors over cpos.

The major **advantage** of a denotational semantics is that it **abstracts** away from **evaluation**, so it is can be used to derive much more **powerful** laws for **reasoning about the behavior of programs**.

Denotational Semantics

- For the imperative object calculus
  - Complex [Reus & Schwinghammer, '06]
  - Separates logical validity from derivability
  - Specification = predicate on programs
  - Current models are still not abstract enough
    - Many natural equivalences do not hold

Since the imperative object calculus has dynamically-allocated higher order store, its denotational semantics is of course **complex** - Jan just finished his Ph.D. thesis on this topic.

This approach is nice since **separates** the notion of logical **validity from derivability** and so it **clarifies the meaning of specifications of the logic**:
-> A specification is a **predicate** over [the denotation of] programs

However, even so the known models are still not abstract enough in that many natural equivalences involving state do not hold.

Step-indexed Semantics

- Foundational proof-carrying code by Appel et. all.
- Lambda calculus
  - Recursive types [Appel & McAllester, '01]
  - General references and polymorphism [Ahmed et. all., '03]
- Also for (very) low-level languages [Benton, '06]
- Based on a small-step operational semantics
- Type = set of indexed values
- "$e :_k T$ if e behaves like an element of T for k steps"

So what could be an **alternative** to using a denotational semantics here?
Step-indexed semantics,might be one. Step-indexed semantics was was developed Appel and his collaborators in the context of foundational proof-carrying code.

It was first introduced for a lambda calculus with **recursive and polymorphic types** in 2001. Later this has been successfully **extended to** an imperative language with **general references and impredicative polymorphism**, [substructural state],
and has also been used for **low-level languages**.

Based on a small-step operational **semantics**,
and here **types** are just **sets of indexed values** - set-theoretical model. **Informally**, an expression has a certain type if it **behaves** like an element of that type for a **fixed number of steps**.

[The usual type inference rules then become derived lemmas, and type safety of the operational semantics is an immediate consequence of this interpretation of types.]

What We Are Working On

- Main goal: Develop a step-indexed semantics for the imperative object calculus with simple object types
  - Small-step semantics (done)
  - Safety for k steps
  - Types and store typings
    - Use indexing to solve cardinality paradoxes
  - Proving soundness of the typing rules

The **main goal** of my thesis is to develop a **step-indexed semantics** for the imperative object calculus with **simple object types**.

What we need to do in order to achieve this is?:
-> We have already defined a **small-step operational semantics** for this calculus.
-> We need to define the notion of **safety** - a term **does not get "stuck"** in k steps.
-> Then we define **types** and **store typings** - make the definition well-founded by using indexing to solve the **cardinality paradoxes**.
-> Last and most important, **prove the usual typing rules sound**.

Possible Extensions

- Enriching the type system with:
  - Subtyping
  - Recursive object types [Schwinghammer, ‘06]
  - Impredicative polymorphism

Once this has been achieved, there are several extensions of the construction that can be investigated. I will present **two.** The first one would be:

->Enriching the type system with features such as
subtyping – should be simple
recursive object types – like in Jan’s thesis
impredicative polymorphism – to our knowledge this was never done for the imperative object calculus before.

Possible Extensions

- Defining a program logic extending types
  - Construct a sound deduction system
    - FOL assertions about the store
      [Abadi & Leino, '04] [Benton '05]
    - Dependent types+shallow embedding to HOL
      - e.g. Hoare Type Theory [Nanevski et all., '06]
  - Local and modular reasoning
    - Separation Logic [Reynolds, '02]
    - Idealized ML [Krishnaswami et all., '06]

-> One further extension would be defining a **program logic** for the imperative object calculus, as an **extension of the type system** and **using the step-indexed semantics** we are currently developing.

Alternatives
-> Augment the type systems with FOL assertions about the store - like we have seen in the logic of Abadi and Leino
-> Extend the type systems with dependent types and use a shallow embedding to HOL (HTT)

And then, well ... there is the original question we had when I started reading about this thesis:
Can we have **local and modular reasoning** the imperative object calculus object.
Can we build a separation logic like the ones developed by Reynolds and O'Hearn? (sep logic eases reasoning about aliasing, pointer programs)
Can we reason about abstract data structures, similarly to the very recent results of Krishnaswami for Idealized ML?

Those area all still **open questions** and will be subject to further investigation.

Advertising

More in-depth discussion
Master Honors Program Seminar
Monday 13th Nov. starting at 18:00

If you are **interested** or **just curious** about what I presented today, especially of a more in depth discussion of the **research topics** I quickly presented at the end of this talk,

then you are invited to come to my talk in the Master Honors Program Seminar.

That is on Monday the 13th from 6PM.

Thank you!

References

[1] Martín Abadi and Luca Cardelli. *A Theory of Objects*. Springer, 1996.

[2] Martín Abadi and K. Rustan M. Leino. A logic of object-oriented programs. In Nachum Dershowitz, editor, *Verification: Theory and Practice. Essays Dedicated to Zohar Manna on the Occasion of his 64th Birthday*, Lecture Notes in Computer Science, pages 11–41. Springer, 2004.

[3] Amal J. Ahmed, Andrew W. Appel, and Roberto Virga. An indexed model of impredicative polymorphism and mutable references. Princeton University, January 2003.

[4] Amal J. Ahmed, Matthew Fluet, and Greg Morrisett. A step-indexed model of substructural state. In Olivier Danvy and Benjamin C. Pierce, editors, *International Conference on Functional Programming (ICFP'05)*, pages 78–91. ACM Press, 2005.

[5] Andrew W. Appel and David McAllester. An indexed model of recursive types for foundational proof-carrying code. *ACM Transactions on Programming Languages and Systems*, 23(5):657–683, September 2001.

[6] Nick Benton. A typed, compositional logic for a stack-based abstract machine. In Zoltan Esik, editor, *Asian Symposium on Programming Languages and Systems APLAS'05*, volume 3780 of *Lecture Notes in Computer Science*, pages 182–196. Springer, 2005.

[7] Nick Benton. Abstracting allocation: the new new thing. In *Computer Science Logic*, volume 4207 of *Lecture Notes in Computer Science*, pages 364–380. Springer, 2006.

This slide won't be shown, it's here just for completeness.

References

[8] Robert W. Floyd. Assigning meanings to programs. In Jacob T. Schwartz, editor, *Proceedings of Mathematical Aspects of Computer Science*, volume 19 of *Proceedings of Symposia in Applied Mathematics*, pages 19–32. American Mathematical Society, April 1967.

[9] C. A. R. Hoare. An Axiomatic Basis of Computer Programming. *Communications of the ACM*, 12:576–580, 1969.

[10] Neelakantan R. Krishnaswami, Lars Birkedal, Jonathan Aldrich, and John C. Reynolds. Idealized ML and its separation logic. Submitted, 2006.

[11] Aleksandar Nanevski, Amal Ahmed, Greg Morrisett, and Lars Birkedal. Abstract predicates and mutable adts in hoare type theory. Technical Report TR-14-06, Harvard University, 2006.

[12] Benjamin C. Pierce. *Types and Programming Languages*. The MIT Press, 2002.

[13] Bernhard Reus and Jan Schwinghammer. Denotational semantics for a program logic of objects. *Mathematical Structures in Computer Science*, 16(2):313–358, April 2006.

[14] John C. Reynolds. Separation logic: A logic for shared mutable data structures. In *Proceedings of the 17th IEEE Symposium on Logic in Computer Science*, pages 55–74. IEEE Computer Society, 2002.

[15] Jan Schwinghammer. *Reasoning about Denotations of Recursive Objects*. PhD thesis, Department of Informatics, University of Sussex, 2006.

[16] Francis Tang and Martin Hofmann. Generation of verification conditions for Abadi and Leino's logic of objects. Presented at 9th International Workshop on Foundations of Object-Oriented Languages, January 2002.

This slide won't be shown, it's here just for completeness.