Semantics of Imperative Objects

Catalin Hritcu

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

November 2006
Overview

- Imperative Object Calculus
  - Operational Semantics
  - “Syntactic” Types
  - Axiomatic Semantics
  - Denotational Semantics
  - Step-indexed Semantics
  - Possible Extensions

Semantics is a way to assign a meaning to programs written in a programming language, in our case the imperative object calculus.

The goal of my thesis is develop a step-indexed semantics for the imperative object calculus, which we plan to extend it in several directions.

However, before that I’m going to present the imperative object calculus with it’s operational, axiomatic and denotational 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 (idealised morels) which capture the essence of object-oriented programming languages.

They have rigourously defined formal semantics.

And 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.

But they can also easily encode functions – the lambda calculus can be easily encoded.
Object Calculi

- Object-based (not class-based)
- In practice: JavaScript, Self, etc.
- Strongly-typed
- A Theory of Objects [Abadi & Cardelli ‘96]
- We study the imperative object calculus [Abadi & Leino, ‘04]

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#.

And even though **object-based** programming languages are **not very popular in practice**, there is one important **exception**: **JavaScript** is object-based. And in the upcoming 2.0 version it will have classes, namespaces and optional types.

The object calculi are of course all **typed**.

and they are described in the **book** by Abadi and Cardelli: **A Theory of Objects**

In this talk I will only discuss a variant of the **imperative object calculus**, as presented by Abadi and Leino in their paper from 2004.
Imperative Object Calculus

- Syntax

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

- More syntactic sugar used in examples

Here is the syntax of our calculus.

Since the language is imperative, computations are described in terms of a store and statements can read and from the store and write to the store.

Variables are identifiers and they are not mutable. 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#

Objects are references and the semantics allows for aliasing. 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

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

• Euclid’s gcd algorithm

\[
gcd = \zeta(y) \lambda x. \lambda z. \text{if } x < z \text{ then } y.gcd x (z - x) \\
\text{else if } z < x \text{ then } y.gcd (x - z) z \text{ else } x
\]
In order to define an operational semantics one constructs a simple **abstract machine**. The meaning of a program is the result obtained by **evaluating** it using this machine.

Since for our imperative object calculus we use a **small-step operational semantics**, **evaluation** is defined as the reflexive transitive closure of a **reduction relation**, which (the reduction relation) defines **transitions** or **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, one for each construct in our language.
## Operational Semantics

### (REL-OBJ)

\[
\frac{l \not\in \text{dom } \sigma \quad o = [f_i = v_i, m_j = \varsigma(y_j)b_j]_{i \in I, j \in J}}{\langle \sigma, [f_i = v_i, m_j = \varsigma(y_j)b_j]_{i \in I, j \in J} \rangle \rightarrow \langle \sigma[l \mapsto o], l \rangle}
\]

### (RED-CLOB)

\[
\frac{l' \not\in \text{dom } \sigma \quad \sigma \cdot l = o}{\langle \sigma, \text{clone } l \rangle \rightarrow \langle \sigma[l' \mapsto o], l' \rangle}
\]

### (REL-SEL)

\[
\frac{\sigma \cdot l.f = v}{\langle \sigma, l.f \rangle \rightarrow \langle \sigma, v \rangle}
\]

### (RED-INV)

\[
\frac{\sigma \cdot l.m = \varsigma(y)b}{\langle \sigma, l.m \rangle \rightarrow \langle \sigma, b[y \mapsto l] \rangle}
\]

### (RED-UPD)

\[
\frac{\sigma \cdot l.f \downarrow \quad \sigma' = \sigma[l \mapsto (\sigma \cdot l)[f \mapsto v]]}{\langle \sigma, l.f := v \rangle \rightarrow \langle \sigma', l \rangle}
\]
Example Reduction

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

Nonterminating evaluation

This program defines an object with a method that directly calls itself and then calls that method. This or course diverges.
Types of Objects

- **Simple types:** $T ::= \text{Bool} \mid \text{Nat} \mid [f_i : T_i, m_j : S_j]_{i \in I, j \in J}$
- **Typing relation:** $(\Gamma \vdash a : T) : (\text{Var} \rightarrow_{\text{fin}} \text{Ty}) \times \text{Prog} \times \text{Ty}$
- **Types are “purely syntactical”**
- **Subtyping (fields invariant, methods covariant)**

$$
\text{Bool} \preceq \text{Bool}
$$

$$
\text{Nat} \preceq \text{Nat}
$$

$$
\forall j \in J' : S_j \preceq R_j \quad I' \subseteq I \quad J' \subseteq J

\Rightarrow [f_i : T_i, m_j : S_j]_{i \in I, j \in J} \preceq [f_i : T_i, m_j : R_j]_{i \in I', j \in J'}
$$

**Simple types:** booleans, naturals and object types.

The **typing relation** is just a tri-place relation, inductively defined by rules.

So **types are purely syntactical objects.**

A **subtyping** relation is defined on object types. Other than the subtyping in width (a subtype can have additional fields and methods compared to the supertype), we also have covariance on method types (an object having a method returning a subtype has a subtype of the same object having a method returning a supertype ... complicated). But in order to have soundness the fields need to be invariant because they are mutable references.
Typing Rules

(T-SUB) \[
\begin{array}{c}
T \preceq T' \\
\Gamma \vdash a : T
\end{array}
\quad \Rightarrow
\quad \begin{array}{c}
\Gamma \vdash a : T'
\end{array}
\]

(T-VAR) \[
\Gamma x = T \\
\quad \Rightarrow
\quad \Gamma \vdash x : T
\]

(T-TRUE) \[
\Gamma \vdash \text{true} : Bool
\]

(T-FALSE) \[
\Gamma \vdash \text{false} : Bool
\]

(T-IF) \[
\Gamma \vdash x : Bool \\
\Gamma \vdash a : T \\
\Gamma \vdash b : T
\quad \Rightarrow
\quad \Gamma \vdash \text{if } x \text{ then } a \text{ else } b : T
\]

(T-LET) \[
\Gamma \vdash a : S \\
\Gamma, x : S \vdash b : T
\quad \Rightarrow
\quad \Gamma \vdash \text{let } x = a \text{ in } b : T
The types of objects can be extended to specifications of program correctness – which brings us to the next concept we want to present: program logics.
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, '97] [Abadi & Leino, '04]
- A refinement of the type system (undecidable)
- Attempt to mechanize [Hofmann & Tang, '02]
- Specifications generalize types

\[ \Gamma \vdash a : T \leadsto E \models a : A :: \varphi \]

- \( \varphi \) is a first-order logic formula (pre + post)
- Sub-specification generalizes subtyping
- Subsumption corresponds to weakening

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. There were attempts to partially mechanize this logic, for example [Hofmann & Tang, ‘02] build a verification condition generator for this logic.

Specifications generalize types.

what was added to the usual typing judgement is a first–order logic formula talking about the store, both before and after the execution of the program. The authors call this formula phi, containing both precondition and postcondition, a transition relation.

Sub–specification generalizes subtyping and subsumption corresponds to weakening in Hoare logic.
The assertions about the store don’t have tight semantics (as in separation logic), this leads to many useless proof steps, proving that things don’t change.

In the paper from 2004, Abadi and Leino prove agreement between the operational semantics and their program logic: if you can prove a program returns a certain value, then the program either diverges or it reduces to that value.

The more usual soundness (everything that is derivable is valid) was much harder to prove [Schwinghammer & Reus, ‘06].
Higher-order Store

- Executable code can be stored
  - General references in ML
  - Pointers to functions in C
  - Callbacks in Java
- Recursion by “tying a knot in the store”
- Example (factorial through a field)

```ml
let x = [f = \( \varsigma(y) \lambda n. n \)] in
let z = [f = x, fac = \( \varsigma(y) \lambda n. \text{if } n = 0 \text{ then } 1 \text{ else } n \times y.f.fac(n - 1) \)] in
z.f := z
```

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

Higher-order store = executible code can be stored

Higher-order store is present in different forms in almost all practical programming languages:

- general references in ML
- pointers to functions in C/C++
- or callbacks in Java

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.
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 surely 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.
In a denotational semantics the meaning of a program is a mathematical object. For example in the simply typed lambda calculus the meaning of a lambda abstraction can be a function. However, once one starts to study more realistic programming languages things are not so simple any more. Heavy machinery from domain theory, category theory and order theory is often necessary.

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 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:

\[ \rightarrow \text{A specification is a predicate over [the denotation of] programs} \]

However, even so the known models are still not abstract enough in that many natural equivalences involving state do not hold.
So what could be an alternative to using a denotational semantics here? Step-indexed semantics, might be one. Step-indexed semantics was developed Appel and his collaborators in the context of foundational proof-carrying code. PCC is a way of certifying that untrusted mobile code does not harm its host. Foundational PCC is a variant of PCC that aims to be more flexible and more secure than PCC, by avoiding commitment to a particular type system. Operational semantics, safety properties and type systems are directly defined in an expressive logic (e.g. HOL) and then the properties are not proved only wrt some type system, but down to the foundations of mathematics.

The proponents of the step-indexed semantics argue that usual syntactic proofs of type soundness (subject reduction = progress + preservation) are hard to mechanize, since they require inductive definitions and reasoning about proofs.

In general proof carrying code deals with very low-level languages, like assembler or even machine code.

However, step-indexed semantics was also used for a lambda calculus with recursive types in 2001. Later this has been successfully extended to an imperative language with general references and impredicative polymorphism.
Step-indexed Semantics

- 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”
- Semantic approach to typing
  - Types are predicates
  - Type constructors functions
- Simpler than a cpo-based denotational semantics
- Used as a model for a program logic [Benton ‘05]

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.

Like in a denotational semantics types have a meaning: they can be viewed as predicates over programs, while type constructors are just functions over these predicates. However, our model is purely set theoretical, so much simpler than the models one usually has with a denotational semantics. Especially the proofs are usually by direct induction on the index.

Used as a model for a program logic (for a low-level languages) [Benton ‘05]
What We Are Working On

• Main goal
  • Develop a step-indexed semantics for the imperative object calculus with simple object types
  • Proving soundness of the typing rules

• Extensions
  • Enriching the type system with:
    • Subtyping
    • Recursive object types
      [Reus & Streicher, ‘02] [Schwinghammer, ‘06]
    • Impredicative polymorphism

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

And, once such a semantics is defined, prove the usual typing rules sound.

Once this has been achieved, there are several extensions of the construction that can be investigated. 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.
More Possible Extensions

- Defining a program logic extending types
  - Construct a sound deduction system
    - FOL assertions about the store [Benton, ‘05]
      [Abadi & Leino, ‘04] [Podelski & Schaefer ‘05]
    - Dependent types+shallow embedding to HOL
      - e.g. Hoare Type Theory [Nanevski et al., ‘06]
- Local and modular reasoning
  - Separation Logic [Reynolds, ‘02]
  - ADTs - Idealized ML [Krishnaswami et al., ‘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 for the imperative object calculus.
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.
Thank you!
References


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


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