Game semantics for object-oriented languages: a progress report

John Longley and Nicholas Wolverson

10 Aug 2006

School of informatics

Motivation

Why investigate game semantics for object oriented languages at all?

- Game semantics can model stateful computation.
- Game semantics is good for *data abstraction*. We can interpret an object as a strategy for its externally observable behaviour, and gain a full abstraction result.

Our approach is to start from a simple category of games $\mathscr{C}$, then give an object-oriented language which can naturally be modelled in that setting, and which captures the behaviour present there.

Language

+ Class-based (but classes are treated as a derived notion)

+ Inheritance

+ Shared mutable state

- No names. No object equality, or cyclic heap topology.

- Sequential computation.

  However, coroutines can be added for limited concurrency.

Talk outline

- The language
- Operational semantics
- Game semantics
- Proof of soundness
- Future work (FA/universality)

Language

We give an affine CBV λ-calculus, with contraction for certain *reusable* types, namely basic types and objects.

Types:

$$\begin{array}{rcl} \tau, \sigma & ::= & N \mid \tau_1 \otimes \tau_2 \mid \tau_1 \to \tau_2 \mid \\ & & \mathbf{Obj}\ \{m_1 \colon \tau_1, \ldots, m_n \colon \tau_n\} \end{array}$$

Terms:

$$\begin{array}{rcl} e & ::= & c_\varphi \mid \mathbf{ifz}\ e\ \mathbf{then}\ e_1\ \mathbf{else}\ e_2 \mid \\ & & \langle e_1, e_2 \rangle \mid \mathbf{let}\ \langle x, y \rangle\ \mathbf{be}\ e_1\ \mathbf{in}\ e_2 \mid \lambda x.e \mid e_1\ e_2 \mid \\ & & \mathbf{obj}\ \{m_1 = e_1, \ldots, m_n = e_n\} \mid e.m \mid Y(e) \mid \mathbf{constr}\ e_1\ e_2 \end{array}$$

Objects are approximately considered reusable (stateful) records of functions.

Classes

A class is a collection of named methods with open recursion through $self$, allowing for inheritance.

A method implementation for $m : \tau \to \tau'$ in a class with state type σ is given as a state-transforming function

$$m : \sigma \otimes \tau \to \sigma \otimes \tau'$$

So state is effectively read from before a method invocation, and written to after. For now we consider σ to be a *basic* type, that is a product of ground types.

Representation of classes

We do without explicit classes in our approach, treating them as a derived construct.

Consider a class as its *step function*, effectively a recursive definition of the resulting object, left "open" for class extension, and "closed" up at object creation time.

$$\mathbf{Class}\ \langle \sigma, \gamma; m\colon \tau_m \to \tau'_m \rangle_{m \in X}$$

$$\rightsquigarrow$$

$$\mathbf{Obj}\ \{m\colon \sigma \otimes \gamma \otimes \tau_m \to \gamma \otimes \tau'_m\rangle_{m \in X} \quad \to$$
$$\mathbf{Obj}\ \{m\colon \sigma \otimes \gamma \otimes \tau_m \to \gamma \otimes \tau'_m\rangle_{m \in X}$$

constr

Construction of stateful objects:

$$\frac{\Gamma \vdash c\colon \mathbf{Obj}\ \{m\colon \sigma \otimes \gamma \otimes \tau_m \to \gamma \otimes \tau'_m\}_{m\in A} \quad \Delta \vdash e\colon \sigma \otimes \gamma}{\Gamma, \Delta \vdash \mathbf{constr}\ c\ e\colon \mathbf{Obj}\ \{m\colon \tau_m \to \tau'_m\}_{m\in A}}\ basic(\gamma), re(\sigma)$$

Given a state-transforming object giving the *implementation* of the desired object, **constr** constructs a new object with the desired behaviour, internalising the specified stateful behaviour.

From this, define:

$$\mathbf{new}\ c\ e \rightsquigarrow \mathbf{constr}\ Y(c)\ e$$

Class extension

We may extend a step function for the superclass to one for the subclass:

$$\mathbf{extend}\ c\ \mathbf{with}\ (\varsigma)\ \{m = e_m\}_{m \in B}$$
$$\rightsquigarrow$$
$$\lambda\varsigma.\ \mathbf{obj}\left\{\begin{array}{ll} m = (c\ \varsigma).m, & m \in A \backslash B \\ m = e_m & m \in B \end{array}\right\}$$

This definition allows the addition of new methods when subclassing, but does not cover adding new state, which would require an extension to our language such as parametric polymorphism.

Operational Semantics

We give an operational semantics with *heaps*. Heaps grow rightwards, and shall be acyclic (only leftward pointers).

The two key rules:

$$\frac{h, e_c \Downarrow h', v_c \quad h', e_v \Downarrow h'', v_s}{h, \mathbf{constr}\ e_c\ e_s \Downarrow h''[l \mapsto \langle v_s, v_c \rangle], l}\ l\ \mathit{fresh}$$

$$\frac{h, e_1 \Downarrow h', l.m \quad h', v_c.m\ \langle v_o, v_s, e_2 \rangle \Downarrow h'', \langle v_s', v \rangle}{h, e_1\ e_2 \Downarrow h''[l \mapsto \langle \langle v_o, v_s' \rangle, v_c \rangle], v}\ h(l) = \langle \langle v_o, v_s \rangle, v_c \rangle$$

Games

We take a category $\mathscr{C}$ of games, where a game is $\langle M_A, \lambda_A, P_A \rangle$ for move-set M_A, O/P labelling λ_A and a set P_A of permitted plays. Strategies $\sigma : A$ are suitable sets of plays from P_A.

Further define games

- $A \otimes B$: interleaving of play in A and B, allowing interference
- $A \& B$: play in either A or B (opponent chooses)
- $A \multimap B$: as $A \otimes B$ but with O/P switched in A
- $!A \cong \mu X.A \oslash X$: linear exponential allowing reuse, as an infinite ordered product of the game A

Games II

A restriction of the $\mathrm{Fam}(-)$ construction gives us the category $\mathscr{D}$ to model values.

The structure required to model our language comes in the form of three broad classes of strategies:

- Static copycat strategies ($\gamma \colon A \otimes B \to B \otimes A$)
- Content-independent dynamic copycat strategies ($d \colon !A \to !A \otimes !A$)
- Content-dependent dynamic strategies (to model $\mathbf{constr}$)

Denotational Semantics—Thread

Central in our denotational semantics is the definition of $[\![\mathbf{constr}\ c\ v]\!]$. We define

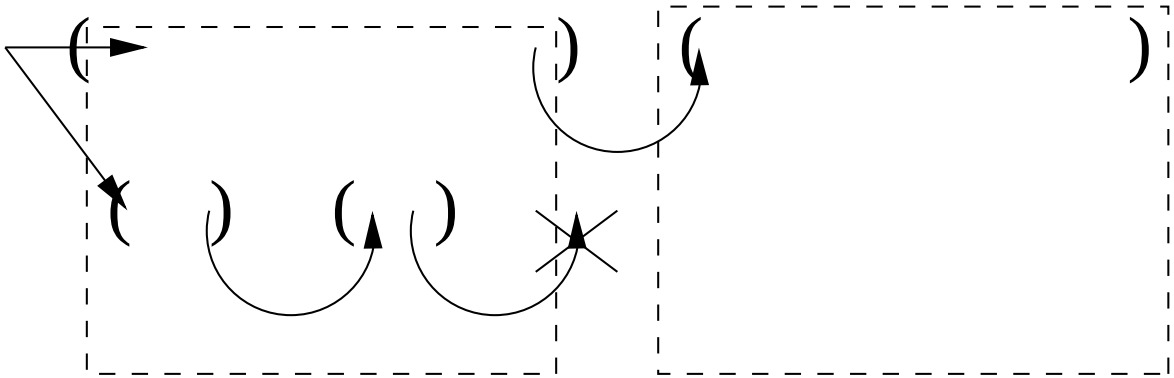
$$thread : S \otimes !(S \otimes X \multimap (S \otimes Y)_\perp) \to !(X \multimap Y_\perp)$$

by a recursively defined, highly dynamic strategy. Consider sequential method invocations—state is threaded through successive method calls, or in other words through successive components of the "!".

However, even in a sequential context, multiple method invocations can be active simultaneously, so this is not sufficient.

Thread II

Non-sequential method invocations may occur during interaction with a method argument, since a method argument may contain a reference to the object in question.



Our concrete definition of thread is given categorically by a few complicated diagrams involving structure associated with !, and a dynamic $branch$ operation.

Soundness

Given an interpretation of $[\![e]\!]([\![h]\!])$, we wish to show

Theorem 1. [Soundness] *If* $h, e \Downarrow h', v$ *then* $[\![e]\!]([\![h]\!]) = [\![v]\!]([\![h']\!])$.

The operational semantics is given for expressions in heaps, but the denotational semantics does not mention heaps. Interpret a heap cell $(l \mapsto (c, v))$ as $\mathbf{constr}\ c\ v$, and the location l in e as a free variable which will be bound to that object. Essentially,

$$[\![(l \mapsto (c, v)), e]\!] = [\![\mathbf{let}\ l\ \mathbf{be}\ (\mathbf{constr}\ c\ v)\ \mathbf{in}\ e]\!]$$

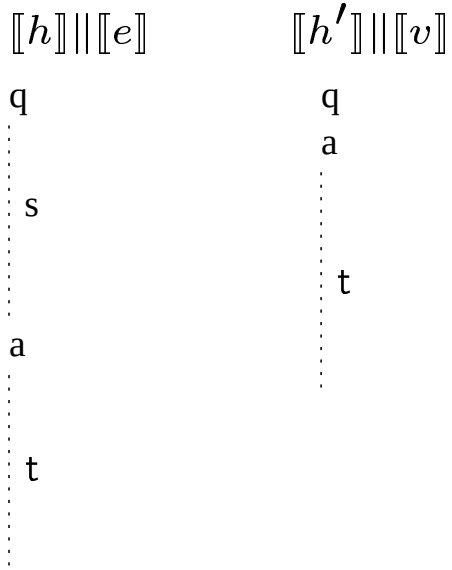
Proof of the above is surprisingly hard, but worthwhile.

A naïve approach

Assuming $h, e \Downarrow h', v$ consider interaction $[\![h]\!] \| [\![e]\!]$. Want

$$qsat \in [\![h]\!] \| [\![e]\!] \Leftrightarrow qat \in [\![h']\!] \| [\![v]\!]$$

where s *takes h to h'*.

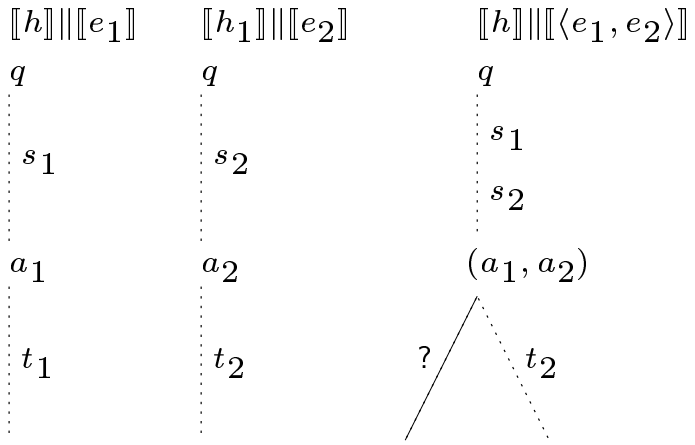


Oops

But consider the simple case of pairing:

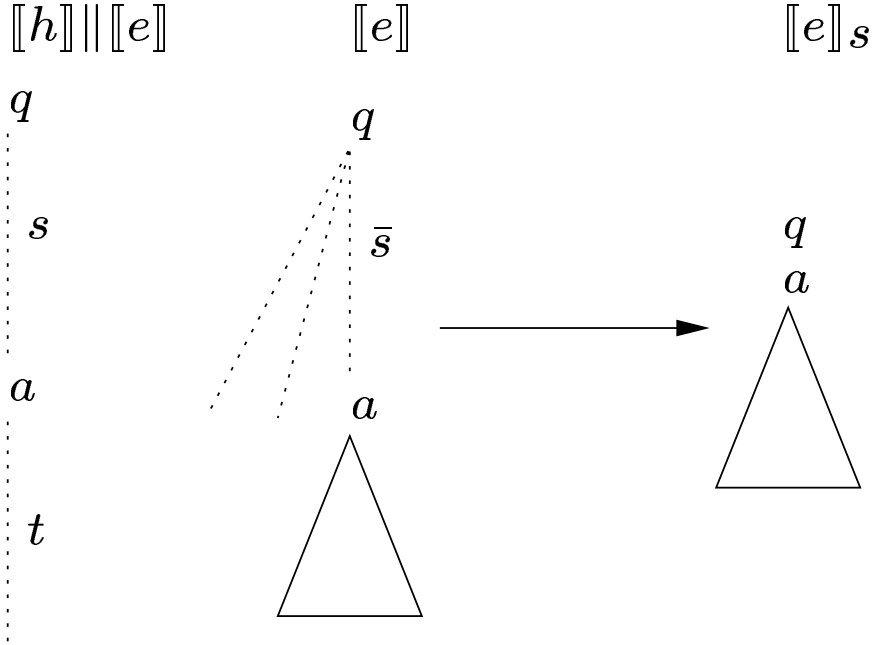
$$\frac{h, e_1 \Downarrow h_1, v_1 \quad h_1, e_2 \Downarrow h', v_2}{h, \langle e_1, e_2 \rangle \Downarrow h', \langle v_1, v_2 \rangle}$$

It's not enough! We need to consider potential future states.



A construction on strategies

We define a construction on strategies for expressions as depicted here, and a corresponding construction on strategies for heaps.



Definition (Simplified)

It is a general fact that

$$[\![e]\!]([\![h]\!]) = [\![e]\!]_s([\![h]\!]^s)$$

so the following definition satisfies our soundness requirements:

Lemma 2. [Soundness] *If* $h, e \Downarrow h', v$ *then* $\exists qsa \in [\![h]\!] \| [\![e]\!]$ *with* $[\![e]\!]_s = [\![v]\!]$ *and* $[\![h]\!]^s = [\![h']\!]$.

This definition separates the immediate interaction with the given heap from the potential future interaction with some descendant heap.

Proof is then by induction on the operational semantics derivation.

Soundness Complications

Complications for soundness proof:

- New state
- Suspended computation in heap
- Result dependent on heap
- Argument interaction and externally mediated recursion
- Substitution

Further work

We have not proved other (completeness) direction of adequacy—use logical relations.

Full abstraction and universality—we hope to prove our interpretation satisfies these properties. Our plan is to construct a program of the following type

$$interpret_\tau : \mathbf{Obj}\ \{\mathrm{step} : N \to N\} \to \tau$$

with the property that

$$\forall e : \mathbf{Obj}\ \{\mathrm{step} : N \to N\}, a : 1 \to [\![\tau]\!].\ (e\ \mathsf{codes}\ a)\ \Rightarrow\ [\![interpret_\tau\ e]\!] = a$$

and construct a proof of both universality and full abstraction with respect to the well-bracketed version of our model.

Language extensions

- Control operators (coroutines) and non-wb games
- Subclass state extension
- Pointer update
- Pointer capture
- Names etc.