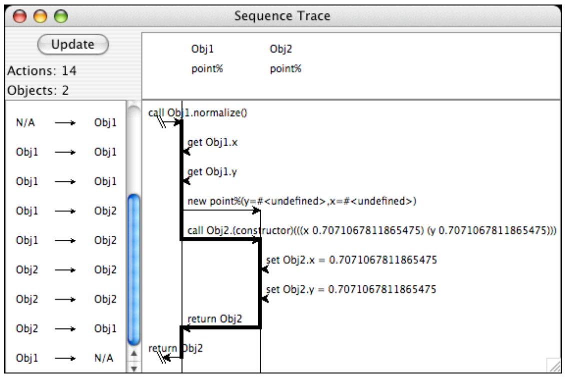
Sequence Traces for Object-Oriented Executions

Carl Eastlund Matthias Felleisen
Northeastern University
{cce,matthias}@ccs.neu.edu

Abstract

Researchers have developed a large variety of semantic models of object-oriented computations. These include object calculi as well as denotational, small-step operational, big-step operational, and reduction semantics. Some focus on pure object-oriented computation in small calculi; many others mingle the object-oriented and the procedural aspects of programming languages.

In this paper, we present a novel, two-level framework of object-oriented computation. The upper level of the framework borrows elements from UML's sequence diagrams to express the message exchanges among objects. The lower level is a parameter of the upper level; it represents all those elements of a programming language that are not object-oriented. We show that the framework is a good foundation for both generic theoretical results and practical tools, such as object-oriented tracing debuggers.



```
Actions: 14
Objects: 2

Obj1      Obj2
point%    point%

N/A  → Obj1    call Obj1.normalize()
Obj1 → Obj1    get Obj1.x
Obj1 → Obj1    get Obj1.y
Obj1 → Obj2    new point%(y=#<undefined>,x=#<undefined>)
Obj1 → Obj2    call Obj2.(constructor)(((x 0.7071067811865475) (y 0.7071067811865475)))
Obj2 → Obj2    set Obj2.x = 0.7071067811865475
Obj2 → Obj2    set Obj2.y = 0.7071067811865475
Obj2 → Obj1    return Obj2
Obj1 → N/A     return Obj2
```

Figure 1. Graphical sequence trace.

1. Models of Execution

Some 30 years ago, Hewitt [22, 23] introduced the ACTOR model of computation, which is arguably the first model of object-oriented computation. Since then, people have explored a range of mathematical models of object-oriented program execution: denotational semantics of objects and classes [7, 8, 25, 33], object calculi [1], small step and big step operational semantics [10], reduction semantics [16], formal variants of ACTOR [2], and others [4, 20].

While all of these semantic models have made significant contributions to the community's understanding of object-oriented languages, they share two flaws. First, consider theoretical results such as type soundness. For ClassicJava, the type soundness proof uses Wright and Felleisen's standard technique of ensuring that type information is preserved while the computation makes progress. If someone extends ClassicJava with constructs such as while loops or switch statements, it is necessary to re-prove everything even though the extension did not affect the object-oriented aspects of the model. Second, none of these models are good starting points for creating practical tools. Some models focus on pure core object-oriented languages; others are models of real-world languages but mingle the semantics of object-oriented constructs (e.g., method invocations) with those of procedural or applicative nature (internal blocks or `while` loops). If a programmer wishes to debug the object-oriented actions in a Java program, a tracer based on any of these semantics would display too much procedural information. In short, a typical realistic model is to object-oriented debugging as a bit-level representation is to symbolic data structure exploration.

In this paper, we introduce a two-level [32] semantic framework for modeling object-oriented programming languages that overcomes these shortcomings. The *upper level* represents all object-oriented actions of a program execution. It tracks six kinds of actions via a rewriting system on object-configurations [26]: object creation, class inspection, field inspection, field mutation, method calls, and method return; we do not consider any other action an object-oriented computation. The computations at this upper level have a graphical equivalent that roughly corresponds to UML sequence diagrams [17]. Indeed, each configuration in the semantics corresponds to a diagram, and each transition between two configurations is an extension of the diagram for the first configuration.

The upper level of the framework is parameterized over the internal semantics of method bodies, dubbed the *lower level*. To instantiate the framework for a specific language, a semanticist must map the object-oriented part of a language to the object-oriented level of the framework and must express the remaining actions as the lower level. The sets and functions defining the lower level may be represented many ways, including state machines, mathematical functions, or whatever else a semanticist finds appropriate. We demonstrate how to instantiate the framework with a Java subset.

In addition to developing a precise mathematical meaning for the framework, we have also implemented a prototype of the framework. The prototype traces a program's object-oriented actions and allows programmers to inspect the state of objects. It is a component of the DrScheme programming environment [13] and covers the kernel of PLT Scheme's class system [15].

The next section presents a high-level overview. Section 3 introduces the framework and establishes a generalized soundness theorem. Section 4 demonstrates how to instantiate the framework for a subset of Java and extends the soundness theorem to that instantiation. Section 5 presents our tool prototype. The last two sections are about related and future work.

$\overrightarrow{t}$	Any number of elements of the form t.
$c[e]$	Expression e in evaluation context c.
$e[x := v]$	Substitution of v for free variable x in expression e.
$d \xrightarrow{p} r$	The set of partial functions of domain d and range r.
$d \xrightarrow{f} r$	The set of finite mappings of domain d and range r.
$[\overrightarrow{a \mapsto b}]$	The finite mapping of each a to the corresponding b.
$f[\overrightarrow{a \mapsto b}]$	Extension of finite mapping f by each mapping of a to b (overriding any existing mappings).

Figure 2. Notational conventions.

2. Sequence Traces

Sequence traces borrow visual elements from UML sequence diagrams, but they represent concrete execution traces rather than specifications. A sequence trace depicts vertical object lifelines and horizontal message arrows with class and method labels, just as in sequence diagrams. The pool of objects extends horizontally; execution of message passing over time extends vertically downward. There are six kinds of messages in sequence traces: **new** messages construct objects, **get** and **set** messages access fields, **call** and **return** messages mark flow control into and out of methods, and **inspect** messages extract an object's tag.

Figure 1 shows a sample sequence trace. This trace shows the execution of the method `normalize` on an object representing the cartesian point $(1, 1)$. The method constructs and returns a new object representing $(\frac{\sqrt{2}}{2}, \frac{\sqrt{2}}{2})$. The first object is labeled `Obj1` and belongs to class `point%`. Its lifeline spans the entire trace and gains control when an external agent calls `Obj1.normalize()`. The first two actions access its `x` and `y` fields (self-directed messages, represented by lone arrowheads). `Obj1` constructs the second `point%` object, `Obj2`, and passes control to its constructor method. `Obj2` initializes its `x` and `y` fields and returns control to `Obj1`. Finally, `Obj1` returns a reference to `Obj2` and yields control.

Sequence traces suggest a model of computation as communication similar to π-calculus models [35]. In this model, an execution for an object-oriented program is represented as a collection of object lifelines and the messages passed between them. The model "hides" computations that take place inside of methods and that don't require any externally visible communication. This is the core of any object-oriented programming language and deserves a formal exploration.

3. The Framework

Our framework assigns semantics to object-oriented languages at two levels. The upper level describes objects, their creation, their lifelines, and their exchanges of messages. The lower level concerns all those aspects of a language's semantics that are unrelated to its object-oriented nature, e.g., static methods, blocks, decision constructs, looping constructs, etc. In this section we provide syntax, semantics, a type system, and a soundness theorem for the upper level.

3.1 The Upper Level

For the remainder of the paper we use the notational conventions shown in Figure 2. Figure 3 gives the full syntax of the upper level using this notation and specifies the language-specific sets over which it is parameterized. A sequence trace is a series of states each containing a pool of objects, a stack of active methods, a reference to a controlling object, and a current action. Objects consist of a static record (their unchanging properties, such as their class) and a dynamic record (their mutable fields). Actions may be one of six message types (**new**, **inspect**, **get**, **set**, **call**, or **return**) or an execution error.

Syntax:

$T = \overrightarrow{S}$	Sequence trace
$S = \langle P, K, r, A \rangle$	Execution state
$P : r \xrightarrow{f} O$	Object pool
$K = \epsilon \mid \langle r, k \rangle\ K$	Method stack
$O = \langle s, D \rangle$	Object record
$D : f \xrightarrow{f} V$	Dynamic record
$V = v \mid r \mid s$	Value
$A = M \mid ERR$	Action
$M = \textbf{new}\ O;\ k \mid \textbf{inspect}\ r;\ k$ $\mid \textbf{get}\ r.f;\ k \mid \textbf{set}\ r.f := V;\ k$ $\mid \textbf{call}\ r.m(\overrightarrow{V});\ k \mid \textbf{return}\ V$	Message
$R = \langle P, \epsilon, r, \textbf{return}\ V \rangle$ $\mid \langle P, K, r, ERR \rangle$	Result
$ERR = err \mid \textbf{error:ref} \mid \textbf{error:field}$	Execution error

Where:

p	lower-level parameter	Program
k	lower-level parameter	Method-local continuation
s	lower-level parameter	Static record
f	lower-level parameter	Field name
m	lower-level parameter	Method name
v	lower-level parameter	Primitive value
err	lower-level parameter	Language-specific error
r	countable set	Object reference

Figure 3. Sequence trace syntax.

Figure 4 gives the upper-level operational semantics of sequence traces along with descriptions and signatures for its lower-level parameters. The parameter *init* is a function mapping a program to its initial state. A trace is the result of rewriting the initial state, step by step, into a final state. Each subsequent state depends on the previous state and action, as follows:

object creation A **new** action adds a reference and an object to the pool. The initiating object retains control.

object inspection An **inspect** action retrieves the static record of an object.

field lookup A **get** action retrieves the value of a field from an object.

field update A **set** action changes the value of a field in an object.

method call A **call** action invokes a method in an object, supplies a number of arguments, and transfers control.

method return A **return** action completes the current method call.

All of these transitions have a natural graphical equivalent (see Section 2).

At each step, the rewriting system uses either the (partial) function *invoke* or *resume* to compute the next action. These functions, like the step relation $\rightarrow$ and several others described below, are indexed by the source program p. Both functions are parameters of the rewriting system. The former begins executing a method; the latter continues one in progress using a method-local continuation. Both functions are partial, admitting the possibility of non-termination at the method-internal level. Also, both functions may map their inputs to a language-specific error.

3.2 Soundness

Our two-level semantic framework comes with a two-level type system. The purpose of this type system is to eliminate *all* upper-level type errors (reference error, field error) and to allow only those language-specific errors on which the lower-level insists. For

Evaluation:

$$\begin{array}{lll}
\langle P, K, r, \textbf{new}\ O;\ k\rangle & \rightarrow_p \langle P[r' \mapsto O], K, r, \mathit{resume}_p(k, r')\rangle & \text{where } r' \notin \mathit{dom}(P) \\
\langle P, K, r, \textbf{inspect}\ r';\ k\rangle & \rightarrow_p \langle P, K, r, \mathit{resume}_p(k, s)\rangle & \text{where } P(r') = \langle s, D\rangle \\
\langle P, K, r, \textbf{get}\ r'.f;\ k\rangle & \rightarrow_p \langle P, K, r, \mathit{resume}_p(k, V)\rangle & \text{where } P(r') = \langle s, D\rangle \text{ and } D(f) = V \\
\langle P, K, r, \textbf{set}\ r'.f := V;\ k\rangle & \rightarrow_p \langle P[r' \mapsto \langle s, D[f \mapsto V]\rangle], K, r, \mathit{resume}_p(k, V)\rangle & \text{where } P(r') = \langle s, D\rangle \text{ and } f \in \mathit{dom}(D) \\
\langle P, K, r, \textbf{call}\ r'.m(\overrightarrow{V});\ k\rangle & \rightarrow_p \langle P, \langle r, k\rangle\ K, r', \mathit{invoke}_p(r', P(r'), m, \overrightarrow{V})\rangle & \text{where } r' \in \mathit{dom}(P) \\
\langle P, \langle r', k\rangle\ K, r, \textbf{return}\ V\rangle & \rightarrow_p \langle P, K, r', \mathit{resume}_p(k, V)\rangle & \\
\left.\begin{array}{l}
\langle P, K, r, \textbf{inspect}\ r';\ k\rangle \\
\langle P, K, r, \textbf{get}\ r'.f;\ k\rangle \\
\langle P, K, r, \textbf{set}\ r'.f := V;\ k\rangle \\
\langle P, K, r, \textbf{call}\ r'.m(\overrightarrow{V});\ k\rangle
\end{array}\right\} & \rightarrow_p \langle P, K, r, \textbf{error:ref}\rangle & \text{where } r' \notin \mathit{dom}(P) \\
\left.\begin{array}{l}
\langle P, K, r, \textbf{get}\ r'.f;\ k\rangle \\
\langle P, K, r, \textbf{set}\ r'.f := V;\ k\rangle
\end{array}\right\} & \rightarrow_p \langle P, K, r, \textbf{error:field}\rangle & \text{where } P(r') = \langle s, D\rangle \text{ and } f \notin \mathit{dom}(D)
\end{array}$$

Where:

$\mathit{init} : p \longrightarrow S$	Constructs the initial program state.
$\mathit{invoke}_p : \langle r, O, m, \overrightarrow{V}\rangle \xrightarrow{\text{p}} A$	Invokes a method.
$\mathit{resume}_p : \langle k, V\rangle \xrightarrow{\text{p}} A$	Resumes a suspended computation.

Figure 4. Sequence trace semantics.

Upper level:

$p \vdash^u S : t$	State S has type t.
$p \vdash^u P$	Object pool P is well-formed.
$p, P \vdash^u K : t_1 \xrightarrow{\text{s}} t_2$	Stack K produces type t_2 if the current method produces type t_1.
$p, P \vdash^u r : o$	Reference r has type o.
$p, P \vdash^u s : t$	Static record s has type t as a value.
$p, P \vdash^u O$ OK in o	Object record O is an object of type o.
$p, P \vdash^u D$ OK in o	Dynamic record D stores fields for an object of type o.
$p, P \vdash^u A : t$	Action A's method returns type t.

Lower level:

$\vdash^\ell p : t$	Program p has type t.
$p, P \vdash^\ell k : t_1 \xrightarrow{\text{c}} t_2$	Continuation k produces an action of type t_2 when given input of type t_1.
$p, P \vdash^\ell s$ OK in o	Static record s is well-formed in an object of type o.
$p, P \vdash^\ell v : t$	Primitive value v has type t.

Figure 5. Type judgments.

t	any set	Value types
$o \subseteq t$		Object types
$\mathit{exn} \subseteq \mathit{err}$		Allowable exceptions
$\sqsubseteq_p$	partial order on t	Subtype relation

$$\left.\begin{array}{rcl}
\mathit{fields}_p & : & o \longrightarrow (f \xrightarrow{\text{f}} t) \\
\mathit{methods}_p & : & o \longrightarrow (m \xrightarrow{\text{f}} \langle \overrightarrow{t}, t\rangle) \\
\mathit{metatype}_p & : & o \longrightarrow t
\end{array}\right\} \text{Produce an object's field, method, or static record types.}$$

Figure 6. Sets, functions, and relations used by the type system.

example, in the case of Java, the lower level cannot rule out `null` pointer errors and must therefore raise the relevant exceptions.

Type judgments in this system are split between those defined at the upper level and those defined at the lower level, as shown in Figure 5. The upper level relies on the lower-level judgments and possibly vice versa. The lower-level type system must provide type judgments for programs, continuations, the static records of objects, and primitive values. The upper-level type system defines type judgments for everything else: program states, object pools, stacks, references, static records when used as values, object records, dynamic records, and actions of both the message and error variety.

$$\textsc{Init}\ \frac{\vdash^\ell p : t}{p \vdash^u \mathit{init}(p) : t}$$

$$\textsc{Resume}\ \frac{\begin{array}{c} p, P \vdash^{u\ell} V : t_1 \\ p, P \vdash^\ell k : t_2 \xrightarrow{\text{c}} t_3 \\ t_1 \sqsubseteq_p t_2 \qquad t_4 \sqsubseteq_p t_3 \end{array}}{p, P \vdash^u \mathit{resume}_p(k, V) : t_4}$$

$$\textsc{Invoke}\ \frac{\begin{array}{c} p, P \vdash^u r : o \qquad \overrightarrow{p, P \vdash^{u\ell} V : t_1} \\ \mathit{methods}_p(o)(m) = \langle \overrightarrow{t_2}, t_3\rangle \\ \overrightarrow{t_1 \sqsubseteq_p t_2} \qquad t_4 \sqsubseteq_p t_3 \end{array}}{p, P \vdash^u \mathit{invoke}(r, P(r), m, \overrightarrow{V}) : t_4}$$

Figure 7. Constraints on the lower-level type system.

The lower level must also define several sets, functions, and type judgments, shown in Figure 6. The set t defines types for the language's values; o defines the subset of t representing the types of objects. The subset exn of err distinguishes the runtime exceptions that well-typed programs may throw.

The subtype relation $\sqsubseteq$ induces a partial order on types. The total functions *fields* and *methods* define the field and method signatures of object types. The total function *metatype* determines the type of a static record from the type of its container object; it is needed to type **inspect** messages.

The Init, Resume, and Invoke typing rules, shown in Figure 7, constrain the lower-level framework functions of the same names. The Init rule states that a program must have the same type as its initial state. The Resume rule states that a continuation's argument object and result action must match its input type and output type, respectively. The Invoke rule states that when an object's method is invoked and given appropriately-typed arguments, it must produce an appropriately-typed action. In addition, a sound system requires all three to be total functions, whereas the untyped operational semantics allows *resume* and *invoke* to be partial. The

Syntax:

$$\begin{array}{rcl}
p & = & \overrightarrow{\Delta} \\
s & = & c \\
f & = & \langle c, f^{\mathrm{cj}} \rangle \\
m & = & m^{\mathrm{cj}} \mid \langle c, m^{\mathrm{cj}} \rangle \\
v & = & \texttt{null} \\
err & = & \textbf{error:method} \mid \textbf{error:null} \\
 & \mid & \textbf{error:typecast} \mid \textbf{error:var} \\
k & = & \{\, \tau\ x\texttt{=}k;\ \overrightarrow{\tau\ x\texttt{=}e;}\ e \,\} \\
 & \mid & \texttt{(}\tau\texttt{)}k \mid \texttt{(}k \sqsubseteq \tau\texttt{)}r \mid k\texttt{:}c\texttt{.}f^{\mathrm{cj}} \\
 & \mid & k\texttt{:}c\texttt{.}f^{\mathrm{cj}}\texttt{=}e \mid V\texttt{:}c\texttt{.}f^{\mathrm{cj}}\texttt{=}k \\
 & \mid & k\texttt{.}m^{\mathrm{cj}}\texttt{(}\overrightarrow{e}\texttt{)} \mid V\texttt{.}m^{\mathrm{cj}}\texttt{(}\overrightarrow{V}\ k\ \overrightarrow{e}\texttt{)} \\
 & \mid & \texttt{super}\equiv r\texttt{:}c\texttt{.}m^{\mathrm{cj}}\texttt{(}\overrightarrow{V}\ k\ \overrightarrow{e}\texttt{)} \mid []
\end{array}$$

Where:

$$\begin{array}{rcll}
i & & \text{countable set} & \text{Interface name} \\
c & & \text{countable set} & \text{Class name} \\
m^{\mathrm{cj}} & & \text{countable set} & \text{Method label} \\
f^{\mathrm{cj}} & & \text{countable set} & \text{Field label} \\
\Delta & = & \texttt{interface}\ i\ \texttt{extends}\ \overrightarrow{i}\ \{\, \overrightarrow{\sigma} \,\} & \text{Definition} \\
 & \mid & \texttt{class}\ c\ \texttt{extends}\ c\ \texttt{implements}\ \overrightarrow{i}\ \{\, \overrightarrow{\phi}\ \overrightarrow{\delta} \,\} & \\
\sigma & = & \tau\ m^{\mathrm{cj}}\texttt{(}\overrightarrow{\tau}\texttt{);} & \text{Method signature} \\
\delta & = & \tau\ m^{\mathrm{cj}}\texttt{(}\overrightarrow{\tau\ x}\texttt{)}\ \{\, e \,\} & \text{Method definition} \\
\phi & = & \tau\ f^{\mathrm{cj}}\texttt{=}e\texttt{;} & \text{Field definition} \\
e & = & V \mid x \mid \texttt{this} \mid \{\, \overrightarrow{\tau\ x\texttt{=}e;}\ e \,\} \mid \texttt{new}\ c & \text{Expression} \\
 & \mid & \texttt{(}\tau\texttt{)}e \mid \texttt{(}c \sqsubseteq \tau\texttt{)}e \mid e\texttt{:}c\texttt{.}f^{\mathrm{cj}} \mid e\texttt{:}c\texttt{.}f^{\mathrm{cj}}\texttt{=}e & \\
 & \mid & e\texttt{.}m^{\mathrm{cj}}\texttt{(}\overrightarrow{e}\texttt{)} \mid \texttt{super}\equiv e\texttt{:}c\texttt{.}m^{\mathrm{cj}}\texttt{(}\overrightarrow{e}\texttt{)} &
\end{array}$$

Figure 8. Java core syntax.

$field_p : \langle c, f^{\mathrm{cj}} \rangle \longrightarrow \phi$	Looks up field definitions.
$method_p : \langle c, m^{\mathrm{cj}} \rangle \longrightarrow \delta$	Looks up method definitions.
$object_p : c \longrightarrow O$	Constructs new objects.
$call_p : \langle r, c, m^{\mathrm{cj}}, \overrightarrow{V} \rangle \longrightarrow A$	Picks a method's first action.
$eval_p : e \longrightarrow A$	Chooses the next action.
$\rightarrow_p^{\mathrm{cj}} : e \xrightarrow{\mathrm{P}} e$	Computes a single step.

Figure 9. Java core relations and functions.

lower level type system must guarantee these rules, while the upper level relies on them for a parametric soundness proof.

THEOREM 1 (Soundness). *If the functions init, resume, and invoke are total and satisfy constraints* INIT, RESUME, *and* INVOKE *respectively, then if* $\vdash^{\ell} p : t$, *then either* p *diverges or* $init(p) \twoheadrightarrow_p R$ *and* $p \vdash^{u} R : t$.

The type system satisfies a conventional type soundness theorem. Its statement assumes that lower-level exceptions are typed; however, they can only appear in the final state of a trace. Due to space limitations, the remaining details of the type system and soundness proof have been relegated to our technical report [12].

4. Framework Instantiations

The framework is only useful if we can instantiate its lower level for a useful object-oriented language. In this section we model a subset of Java in our framework, establishes its soundness, and consider an alternate interpretation of Java that strikes at the heart of the question of which language features are truly object-oriented. We also discuss a few other framework instantiations.

4.1 Java via Sequence Traces

Our framework can accomodate the sequential core of Java, based on ClassicJava [16], including classes, subclasses, interfaces, method overriding, and typecasts. Figure 8 shows the syntax of the Java core. Our set of expressions includes lexically scoped blocks, object creation, typecasts, field access, method calls, and superclass method calls. Field access and superclass method calls have class annotations on their receiver to aid the type soundness lemma in Section 4.3. Typecast expressions have an intermediate form used in our evaluation semantics. We leave out many other Java constructs such as conditionals, loops, etc.

Programs in this language are a sequence of class and interface definitions. An object's static record is the name of its class. Field names include a field label and a class name. Method names include a label and optionally a class name. The sole primitive value is `null`. We define errors for method invocation, null dereference, failed typecasts, and free variables. Last but not least, local continuations are evaluation contexts over expressions.

Figure 10 defines the semantics of our Java core using the relations and functions described in Figure 9. We omit the definitions of $\sqsubseteq$, *field*, and *method*, which simply inspect the sequence of class and interface definitions. The *init* function constructs an object of class `Program` and invokes its `main` method. The *resume* function constructs a new expression from the given value and the local continuation (a context), then passes it to *eval*; *invoke* simply uses *call*.

Method invocation uses *call* for dispatch. This function looks up the appropriate method in the program's class definitions. It substitutes the method's receiver and parameters, then calls *eval* to evaluate the expression.

The *eval* function is defined via a reduction relation $\rightarrow^{\mathrm{cj}}$. That is, its results are determined by the canonical forms of expression with respect to $\twoheadrightarrow^{\mathrm{cj}}$, the reflexive transitive closure. Object creation, field lookup, field mutation, method calls, and method returns all generate corresponding framework actions. Unelaborated typecast expressions produce inspection actions, adding an elaborated typecast context to their continuation. The *eval* function signals an error for all null dereferences and typecast failures.

Calls to an object's superclass generate method call actions; that is, an externally visible message. The method name includes the superclass name for method dispatch, which distinguishes it from the current definition of the method.

The step relation ($\rightarrow^{\mathrm{cj}}$) performs all purely object-internal computations. It reduces block expressions by substitution and completes successful typecasts by replacing the elaborated expression with its argument.

LEMMA 1. *For any expression* e, *there is some* e' *such that* $e \twoheadrightarrow_p^{\mathrm{cj}} e'$ *and* e' *is of canonical form.*

Together, the sets of canonical expressions and of expressions on which $\rightarrow^{\mathrm{cj}}$ is defined are exhaustive. Furthermore, each step of $\rightarrow^{\mathrm{cj}}$ strictly reduces the size of the expression. The expression must reduce in a finite number of steps to a canonical form for which *eval* produces an action. Therefore *eval* is total.

COROLLARY 1. *The functions invoke and resume are total.*

Because these functions are total, evaluation in the sequential core of Java cannot get stuck; each state must either have a successor or be a final result.

4.2 Alternate Interpretation of the Java Core

Our parameterization of the sequence trace framework for Java answers the question: "what parts of the Java core are object-

$$\begin{aligned}
&init(p) = \langle [r_0 \mapsto object_p(\texttt{Program})], \epsilon, r_0, \textbf{call}\ r_0.\texttt{main}();\ []\rangle\\
&resume_p(k, V) = eval_p(k[V])\\
&invoke_p(r, \langle c, D\rangle, m^{\text{cj}}, \overrightarrow{V}) = call_p(r, c, m^{\text{cj}}, \overrightarrow{V})\\
&invoke_p(r, \langle c, D\rangle, \langle c', m^{\text{cj}}\rangle, \overrightarrow{V}) = call_p(r, c', m^{\text{cj}}, \overrightarrow{V})
\end{aligned}$$

$$\begin{aligned}
&object(c) = \langle c, [\overrightarrow{\langle c', f^{\text{cj}}\rangle \mapsto \texttt{null}}]\rangle\\
&\text{where } \overrightarrow{field_p(c, f^{\text{cj}}) = \tau\ f^{\text{cj}}\texttt{=}c'\texttt{;}}
\end{aligned}$$

$$call_p(r, c, m^{\text{cj}}, \overrightarrow{V}) = \begin{cases} eval_p(e[\texttt{this} := r][\overrightarrow{x := V}]) \\ \quad \text{if } method_p(c, m^{\text{cj}}) = \tau\ m^{\text{cj}}\texttt{(}\overrightarrow{\tau\ x}\texttt{)}\ \{\ e\ \} \\ \textbf{error:method} \text{ otherwise} \end{cases}$$

$$eval_p(e) = \begin{cases}
\textbf{return}\ V & \text{if } e \twoheadrightarrow^{\text{cj}}_p V\\
\textbf{new}\ object(c);\ k & \text{if } e \twoheadrightarrow^{\text{cj}}_p k[\texttt{new}\ c]\\
\textbf{get}\ r.\langle c, f\rangle;\ k & \text{if } e \twoheadrightarrow^{\text{cj}}_p k[r\texttt{:}c\texttt{.}f]\\
\textbf{set}\ r.\langle c, f\rangle := V;\ k & \text{if } e \twoheadrightarrow^{\text{cj}}_p k[r\texttt{:}c\texttt{.}f\texttt{=}V]\\
\textbf{call}\ r.m(\overrightarrow{V});\ k & \text{if } e \twoheadrightarrow^{\text{cj}}_p k[r\texttt{.}m\texttt{(}\overrightarrow{V}\texttt{)}]\\
\textbf{call}\ r.\langle c, m\rangle(\overrightarrow{V});\ k & \text{if } e \twoheadrightarrow^{\text{cj}}_p k[\texttt{super}\equiv r\texttt{:}c\texttt{.}m^{\text{cj}}\texttt{(}\overrightarrow{V}\texttt{)}]\\
\textbf{inspect}\ r;\ k[\texttt{(}[] \sqsubseteq \tau\texttt{)}r] & \text{if } e \twoheadrightarrow^{\text{cj}}_p k[\texttt{(}\tau\texttt{)}r]\\
\textbf{error:typecast} & \text{if } e \twoheadrightarrow^{\text{cj}}_p k[\texttt{(}\tau\texttt{)null}] \text{ or } e \twoheadrightarrow^{\text{cj}}_p k[\texttt{(}c \sqsubseteq \tau\texttt{)}V] \text{ and } c \not\sqsubseteq_p \tau\\
\textbf{error:null} & \text{if } e \twoheadrightarrow^{\text{cj}}_p k[\texttt{null:}c\texttt{.}f] \text{ or } e \twoheadrightarrow^{\text{cj}}_p k[\texttt{null:}c\texttt{.}f\texttt{=}V] \text{ or } e \twoheadrightarrow^{\text{cj}}_p k[\texttt{null.}m\texttt{(}\overrightarrow{V}\texttt{)}]\\
\textbf{error:var} & \text{if } e \twoheadrightarrow^{\text{cj}}_p k[x] \text{ or } e \twoheadrightarrow^{\text{cj}}_p k[\texttt{this}]
\end{cases}$$

$$\begin{aligned}
&k[\{\ \tau'\ x'\texttt{=}V';\ \overrightarrow{\tau\ x\texttt{=}e;}\ e'\ \}] && \rightarrow^{\text{cj}}_p && k[\{\ \overrightarrow{\tau\ x\texttt{=}e[x' := V'];}\ e'[x' := V']\ \}]\\
&k[\{\ e\ \}] && \rightarrow^{\text{cj}}_p && k[e]\\
&k[\texttt{(}c \sqsubseteq \tau\texttt{)}V] && \rightarrow^{\text{cj}}_p && k[V] \text{ if } c \sqsubseteq_p \tau
\end{aligned}$$

Figure 10. Java core semantics and auxiliary definitions.

oriented?" In the semantics above, the answer is clear: object creation, field lookup and mutation, method calls, method returns, superclass method calls, and typecasts.

Let us reconsider this interpretation. The most debatable aspect of our model concerns superclass method calls. They take place entirely inside one object and cannot be invoked by outside objects, yet we have formalized them as messages. An alternate perspective might formulate superclass method calls as object-internal computation for comparison.

Our framework is flexible enough to allow this reinterpretation of Java. In our semantics above, as in other models of Java [3, 10, 16, 24], **super** expressions evaluate to method calls. Method calls use *invoke* which uses *call*. We can change *eval* to use *call* directly in the **super** rule, i.e. no object-oriented action is created. The extra clauses for method names and *call* that were used for superclass calls can be removed. These modifications are shown in Figure 11.[1]

Now that we have two different semantics for Java, it is possible to compare them and to study the tradeoffs; implementors and semanticists can use either interpretation as appropriate.

4.3 Soundness of the Java Core

We have interpreted the type system for the Java core in our framework and established its soundness. Again, the details of the type system and soundness proof can be found in our technical report.

LEMMA 2. *The functions init, resume, and invoke are total and satisfy constraints* INIT, RESUME, *and* INVOKE.

According to Corollary 1, these functions are total. Since INIT, RESUME, and INVOKE hold, type soundness is just a corollary of Theorem 1.

COROLLARY 2 (Java Core Soundness). *In the Java core, if* $\vdash^{\ell} p : t$*, then either* p *diverges or* $init(p) \twoheadrightarrow_p R$ *and* $p \vdash^{u} R : t$.

[1] Note that *invoke* and *resume* are no longer total for cyclic class graphs. A soundness proof for this formulation must account for this exception, or *call* must be further refined to reject looping **super** calls.

$m = m^{\text{cj}}$ ~~$\mid \langle c, m^{\text{cj}}\rangle$~~

$invoke_p(r, \langle c, D\rangle, m^{\text{cj}}, \overrightarrow{V}) = call_p(r, c, m^{\text{cj}}, \overrightarrow{V})$

~~$invoke_p(r, \langle c, D\rangle, \langle c', m^{\text{cj}}\rangle, \overrightarrow{V}) = call_p(r, c', m^{\text{cj}}, \overrightarrow{V})$~~

$eval_p(e) =$

$\begin{cases} \ldots \\ \text{~~} \textbf{call}\ r.\langle c, m\rangle(\overrightarrow{V});\ k \text{ if } e \twoheadrightarrow^{\text{cj}}_p k[\texttt{super}\equiv r\texttt{:}c\texttt{.}m^{\text{cj}}\texttt{(}\overrightarrow{V}\texttt{)}] \text{~~} \\ call_p(r, c, m^{\text{cj}}, \overrightarrow{V}) \text{ if } e \twoheadrightarrow^{\text{cj}}_p k[\texttt{super}\equiv r\texttt{:}c\texttt{.}m^{\text{cj}}\texttt{(}\overrightarrow{V}\texttt{)}] \end{cases}$

Figure 11. Changes for an alternate interpretation of Java.

4.4 Other Languages

The expressiveness of formal sequence traces is not limited to just one model. In addition to ClassicJava, we have modeled Abadi and Cardelli's object calculus [1], the λ-calculus, and the $\lambda\&$-calculus [5] in our framework. The λ-calculus is the canonical model of functional computation, and the $\lambda\&$-calculus is a model of dispatch on multiple arguments. These instantiations demonstrate that sequence traces can model diverse (even non-object-oriented) languages and complex runtime behavior. Our technical report contains the full embeddings.

5. Practical Experience

To demonstrate the practicality of our semantics, we have implemented a Sequence Trace tool for the PLT Scheme class system [15]. As a program runs, the tool displays messages passed between objects. Users can inspect data associated with objects and messages at each step of execution. Method-internal function calls or other applicative computations remain hidden.

PLT Scheme classes are implemented via macros [9, 14] in a library, but are indistinguishable from a built-in construct. Traced programs link to an instrumented version of the library. The instrumentation records object creation and inspection, method entry and exit, and field access, exactly like the framework. Both instru-

```
(define point%
  (class object%
    ...
    (define (translate dx dy) ...)))

(define polygon%
  (class object%
    ...
    (define (add-vertex v) ...)
    (define (translate dx dy) ...)))

(send* (new polygon%)
  (add-vertex ...)
  (add-vertex ...)
  (add-vertex ...)
  (translate 5 5))
```

Figure 12. Excerpt of an object-oriented PLT Scheme program.

mented and non-instrumented versions of the library use the same implementation of objects, so traced objects may interact with untraced objects; however, untraced objects do not pay for the instrumentation overhead.

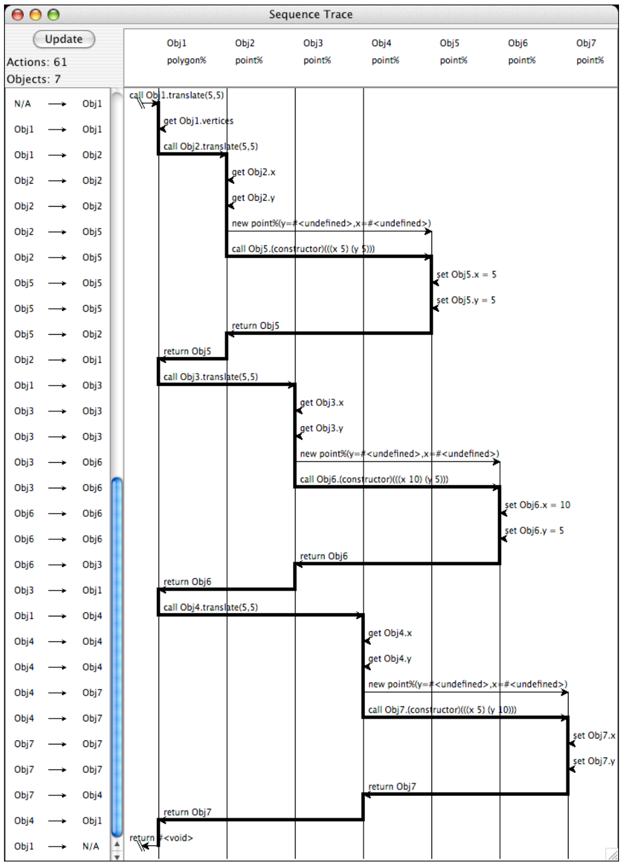
Figure 13 shows a sample sequence trace generated by our tool. This trace represents a program fragment, shown in Figure 12, using a class-based geometry library. The primary object is a `polygon%` containing three `point%` objects. The trace begins with a call to the polygon's `translate` method. The polygon must in turn translate each point, so it iterates over its vertices invoking their `translate` methods. Each original point constructs, initializes, and returns a new translated point.

The graphical layout allows easy inspection and navigation of a program. The left edge of the display allows access to the sender and receiver objects of each message. Each object lifeline provides access to field values and their history. Each message exposes the data and objects passed as its parameters. Highlighted sections of lifelines and message arrows emphasize flow control. Structured algorithms form recognizable patterns, such as the three iterations of the method `translate` on class `point%` shown in Figure 13, aiding in navigating the diagram, tracking down logic errors, and comparing executions to specifications.

6. Related Work

Our work has two inspirational sources. Calculi for communicating processes often model just those actions that relate to process creation, communication, etc. This corresponds to our isolation of object-oriented actions in the upper level of the framework. Of course, our framework also specifies a precise interface between the two levels and, with the specification of a lower level, has the potential to model entire languages. Starting from this insight, Graunke et al. [18, 19, 27] have recently created a trace calculus for a sequential client-server setting. This calculus models a web client (browser) and web server with the goal of understanding systemic flaws in interactive web programs. Roughly speaking, our paper generalizes Graunke et al.'s research to an arbitrarily large and growing pool of objects with a general set of actions and a well-defined interface to the object-internal computational language.

Other tools for inspecting and debugging program traces exist, tackling the problem from many different perspectives. Lewis [28] presents a so-called omniscient debugger, which records every change in program state and reconstructs the execution after the fact. Intermediate steps in the program's execution can thus be debugged even after program completion. This approach is similar to our own, but with emphasis on the pragmatics of debugging rather



```
N/A  → Obj1   call Obj1.translate(5,5)
Obj1 → Obj1   get Obj1.vertices
Obj1 → Obj2   call Obj2.translate(5,5)
Obj2 → Obj2   get Obj2.x
Obj2 → Obj2   get Obj2.y
Obj2 → Obj5   new point%(y=#<undefined>,x=#<undefined>)
Obj2 → Obj5   call Obj5.(constructor)(((x 5) (y 5)))
Obj5 → Obj5   set Obj5.x = 5
Obj5 → Obj5   set Obj5.y = 5
Obj5 → Obj2   return Obj5
Obj2 → Obj1   return Obj5
Obj1 → Obj3   call Obj3.translate(5,5)
Obj3 → Obj3   get Obj3.x
Obj3 → Obj3   get Obj3.y
Obj3 → Obj6   new point%(y=#<undefined>,x=#<undefined>)
Obj3 → Obj6   call Obj6.(constructor)(((x 10) (y 5)))
Obj6 → Obj6   set Obj6.x = 10
Obj6 → Obj6   set Obj6.y = 5
Obj6 → Obj3   return Obj6
Obj3 → Obj1   return Obj6
Obj1 → Obj4   call Obj4.translate(5,5)
Obj4 → Obj4   get Obj4.x
Obj4 → Obj4   get Obj4.y
Obj4 → Obj7   new point%(y=#<undefined>,x=#<undefined>)
Obj4 → Obj7   call Obj7.(constructor)(((x 5) (y 10)))
Obj7 → Obj7   set Obj7.x
Obj7 → Obj7   set Obj7.y
Obj7 → Obj4   return Obj7
Obj4 → Obj1   return Obj7
Obj1 → N/A    return #<void>
```

Figure 13. Sample output of the PLT Scheme Sequence Trace tool.

than presenting an intuitive model of computation. Lewis does not present a theoretical framework and does not abstract his work from Java.

Execution traces are used in many tools for program analysis. Walker et al.'s tool [36] allows users to group program elements into abstract categories, then coalesces program traces accordingly and presents the resulting abstract trace. Richner and Ducasse [34] demonstrate automated recovery of class collaborations from traces. Ducasse et al. [11] provide a regression test framework in which successful logical queries over existing execution traces become specifications for future versions. Our tool is similar to these in that it uses execution traces; however, we do not generate abstract specifications. Instead we allow detailed inspection of the original trace itself.

Even though our work does not attempt to assign semantics to UML's sequence diagrams, many pieces of research in this direction exist and share some similarities with our own work. We therefore describe the most relevant work here. Many semantics for UML provide a definition for sequence diagrams as program specifications. Xia and Kane [37] and Li et al. [29] both develop paired static and dynamic semantics for sequence diagrams. The static semantics validate classes, objects, and operations referenced by methods; the dynamic semantics validate the execution of individual operations. Nantajeewarawat and Sombatsrisomboon [31] define a model-theoretic framework that can infer class diagrams from sequence diagrams. Cho et al. [6] provide a semantics in a new temporal logic called HDTL. These semantics are all concerned with specifications; unlike our work, they do not address object-oriented computation itself.

Lund and Stølen [30] and Hausmann et al. [21] both provide an operational semantics for UML itself, making specifications executable. Their work is dual to ours: we give a graphical, UML-inspired semantics to traditional object-oriented languages, while they give traditional operational semantics to UML diagrams.

7. Conclusions and Future Work

This paper presents a two-level semantics framework for object-oriented programming. The framework carefully distinguishes actions on objects from internal computations of objects. The two levels are separated via a collection of sets and partial functions. At this point the framework can easily handle models such as the core features of Java, as demonstrated in section 4, and languages such as PLT Scheme, as demonstrated in section 5.

Sequence traces still present several opportunities for elaboration at the object-oriented level. Most importantly, the object-oriented level currently assumes a functional creation mechanism for objects. While we can simulate the complex object construction of Java or PLT Scheme with method calls, we cannot model them directly. Conversely, the framework does not support a destroy action. This feature would require the extension of sequence traces with an explicit memory model, possibly parameterized over lower level details.

References

[1] Abadi, M. and L. Cardelli. *A Theory of Objects*. Springer, 1996.

[2] Agha, G., I. A. Mason, S. F. Smith and C. L. Talcott. A foundation for actor computation. *J. Functional Programming*, 7(1):1–72, 1997.

[3] Bierman, G. M., M. J. Parkinson and A. M. Pitts. MJ: an imperative core calculus for Java and Java with effects. Technical report, Cambridge University, 2003.

[4] Bruce, K. B. *Foundations of Object-Oriented Languages: Types and Semantics*. MIT Press, 2002.

[5] Castagna, G., G. Ghelli and G. Longo. A calculus for overloaded functions with subtyping. *Information and Computation*, 117(1):115–135, 1995.

[6] Cho, S. M., H. H. Kim, S. D. Cha and D. H. Bae. A semantics of sequence diagrams. *Information Processing Letters*, 84(3):125–130, 2002.

[7] Cook, W. R. *A Denotational Semantics of Inheritance*. PhD thesis, Brown University, 1989.

[8] Cook, W. R. and J. Palsberg. A denotational semantics of inheritance and its correctness. In *Proc. 1989 Conference on Object-Oriented Programming: Systems, Languages, and Applications*, p. 433–443. ACM Press, 1989.

[9] Culpepper, R., S. Tobin-Hochstadt and M. Flatt. Advanced macrology and the implementation of Typed Scheme. In *Proc. 8th Workshop on Scheme and Functional Programming*, p. 1–14. ACM Press, 2007.

[10] Drossopoulou, S. and S. Eisenbach. Java is type safe—probably. In *Proc. 11th European Conference on Object-Oriented Programming*, p. 389–418. Springer, 1997.

[11] Ducasse, S., T. Gîrba and R. Wuyts. Object-oriented legacy system trace-based logic testing. In *Proc. 10th European Conference on Software Maintenance and Reengineering*, p. 37–46, 2006.

[12] Eastlund, C. and M. Felleisen. Sequence traces for object-oriented executions. Technical report, Northeastern University, 2006.

[13] Findler, R. B., J. Clements, C. Flanagan, M. Flatt, S. Krishnamurthi, P. Steckler and M. Felleisen. DrScheme: a programming environment for Scheme. *J. Functional Programming*, 12(2):159–182, 2002.

[14] Flatt, M. Composable and compilable macros: you want it *when?* In *Proc. 7th ACM SIGPLAN International Conference on Functional Programming*, p. 72–83. ACM Press, 2002.

[15] Flatt, M., R. B. Findler and M. Felleisen. Scheme with classes, mixins, and traits. In *Proc. 4th Asian Symposium on Programming Languages and Systems*, p. 270–289. Springer, 2006.

[16] Flatt, M., S. Krishnamurthi and M. Felleisen. Classes and mixins. In *Proc. 25th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, p. 171–183. ACM Press, 1998.

[17] Fowler, M. and K. Scott. *UML Distilled: Applying the Standard Object Modeling Language*. Addison-Wesley, 1997.

[18] Graunke, P., R. Findler, S. Krishnamurthi and M. Felleisen. Modeling web interactions. In *Proc. 15th European Symposium on Programming*, p. 238–252. Springer, 2003.

[19] Graunke, P. T. *Web Interactions*. PhD thesis, Northeastern University, 2003.

[20] Gunter, C. A. and J. C. Mitchell, editors. *Theoretical Aspects of Object-Oriented Programming: Types, Semantics, and Language Design*. MIT Press, 1994.

[21] Hausmann, J. H., R. Heckel and S. Sauer. Towards dynamic meta modeling of UML extensions: an extensible semantics for UML sequence diagrams. In *Proc. IEEE 2001 Symposia on Human Centric Computing Languages and Environments*, p. 80–87. IEEE Press, 2001.

[22] Hewitt, C. Viewing control structures as patterns of passing messages. *Artificial Intelligence*, 8(3):323–364, 1977.

[23] Hewitt, C., P. Bishop and R. Steiger. A universal modular ACTOR formalism for artificial intelligence. In *Proc. 3rd International Joint Conference on Artificial Intelligence*, p. 235–245. Morgan Kaufmann, 1973.

[24] Igarashi, A., B. Pierce and P. Wadler. Featherweight Java: a minimal core calculus for Java and GJ. In *Proc. 1999 Conference on Object-Oriented Programming: Systems, Languages, and Applications*, p. 132–146. ACM Press, 1999.

[25] Kamin, S. N. Inheritance in SMALLTALK-80: a denotational definition. In *Proc. 15th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, p. 80–87. ACM Press, 1988.

[26] Klop, J. W. Term rewriting systems: a tutorial. *Bulletin of the EATCS*, 32:143–182, 1987.

[27] Krishnamurthi, S., R. B. Findler, P. Graunke and M. Felleisen. Modeling web interactions and errors. In *Interactive Computation: the New Paradigm*, p. 255–275. Springer, 2006.

[28] Lewis, B. Debugging backwards in time. In *Proc. 5th International Workshop on Automated Debugging*, 2003. `http://www.lambdacs.com/debugger/AADEBUG_Mar_03.pdf`.

[29] Li, X., Z. Liu and J. He. A formal semantics of UML sequence diagrams. In *Proc. 15th Australian Software Engineering Conference*, p. 168–177. IEEE Press, 2004.

[30] Lund, M. S. and K. Stølen. Extendable and modifiable operational semantics for UML 2.0 sequence diagrams. In *Proc. 17th Nordic Workshop on Programming Theory*, p. 86–88. DIKU, 2005.

[31] Nantajeewarawat, E. and R. Sombatsrisomboon. On the semantics of Unified Modeling Language diagrams using Z notation. *Int. J. Intelligent Systems*, 19(1–2):79–88, 2004.

[32] Nielson, F. and H. R. Nielson. *Two-level functional languages*. Cambridge University Press, 1992.

[33] Reddy, U. S. Objects as closures: abstract semantics of object-oriented languages. In *Proc. 1988 ACM Conference on LISP and Functional Programming*, p. 289–297. ACM Press, 1988.

[34] Richner, T. and S. Ducasse. Using dynamic information for the iterative recovery of collaborations and roles. In *Proc. International Conference on Software Maintenance*, p. 34–43. IEEE Press, 2002.

[35] Sangiorgi, D. and D. Walker. *The Pi-Calculus: A Theory of Mobile Processes*. Cambridge University Press, 2003.

[36] Walker, R. J., G. C. Murphy, J. Steinbok and M. P. Robillard. Efficient mapping of software system traces to architectural views. In *Proc. 2000 Conference of the Centre for Advanced Studies on Collaborative Research*, p. 12. IBM Press, 2000.

[37] Xia, F. and G. S. Kane. Defining the semantics of UML class and sequence diagrams for ensuring the consistency and executability of OO software specification. In *Proc. 1st International Workshop on Automated Technology for Verification and Analysis*, 2003. `http://cc.ee.ntu.edu.tw/~atva03/papers/16.pdf`.