Sequence Traces for Object-Oriented Executions

Carl Eastlund
Northeastern University
Boston, Massachusetts
cce@ccs.neu.edu

Matthias Felleisen
Northeastern University
Boston, Massachusetts
matthias@ccs.neu.edu

ABSTRACT

Over the past 30 years, 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. None of them, however, separate the object-oriented aspects of a computation from the procedural or applicative ones; none express the object-oriented computations via message-based communication among pools of objects.

In this paper, we present a novel, two-level framework of object-oriented computation. Our purpose is to create a foundation for program inspection and debugging tools that enable programmers to focus on the object-oriented part of a computation and to zoom in on the internal actions of objects only as needed. 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 models all those elements of a programming language that are procedural. We include several instantiations of the framework as well as its application to a uniform (parameterized) type soundness proof.

1. MODELS OF EXECUTION

Some 30 years ago, Hewitt [20, 21] 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, 22, 30]; object calculi [1]; small step and big step operational semantics [9]; reduction semantics [13]; formal variants of ACTOR [2]; and so on [4, 18].

While all of these semantic models have their appropriate uses and contributed to the community's understanding of object-oriented languages, none of them isolate the intuition of pools of objects that exchange messages with each other. As the messages go back and forth, objects change state; get to know more about other objects; and the pool of objects grows. Everything else is object-internal.

Northeastern University College of Computer and Information Science Technical Report NU-CCIS-06-11.

In contrast to semantic models of classes and objects, specification frameworks such as UML [14] provide graphical languages for describing classes, objects, and their desired behaviors. In particular, UML provides (among other notations) *sequence diagrams*. Requirements analysts and specifiers use them to record the desired time-line of interactions among (classes of) objects, repetitions, constraints, and other expected details.

Ideally, language researchers should provide a semantic framework that matches such object-oriented specifications of executions. Tools based on such a framework would then present programmers a view of the computation that is related to their specifications. When needed, they can expose internal actions of methods and objects. In this paper, we introduce such a two-level framework. The *upper level* represents all object-oriented actions during a program's execution. It tracks six kinds of actions: object creation, field inspection, field mutation, object tag inspection, method calls and method return; we do not consider any other action an object-oriented computation. Computations at this upper level are (equivalent to) manipulations of UML-like sequence diagrams; to assign a precise meaning to these manipulations, each diagram is expressible as a configuration and the manipulations are formulated as rewriting actions on configurations [23].

The upper level is parameterized over the internal semantics of method bodies, dubbed the *lower level*. Technically the parameterization takes the form of several sets and functions. 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. Thus, the lower level can be represented as a state machine, a set of mathematical functions, or whatever else a semanticist finds appropriate. We demonstrate how to instantiate the framework with several examples.

In addition to developing a precise mathematical meaning for the framework, we have also implemented a prototype of the framework. The prototype is a component of DrScheme [11] for the kernel of PLT Scheme's class system [12]. It draws a UML-like trace of the program's object-oriented actions and allows programmers to inspect object states.

The rest of the paper is organized as follows. Section 2 presents a high-level overview of the prototype, motivating the development of the semantics. Section 3 introduces the framework and section 4 demonstrates how to instantiate it for three radically different languages. Section 5 shows how

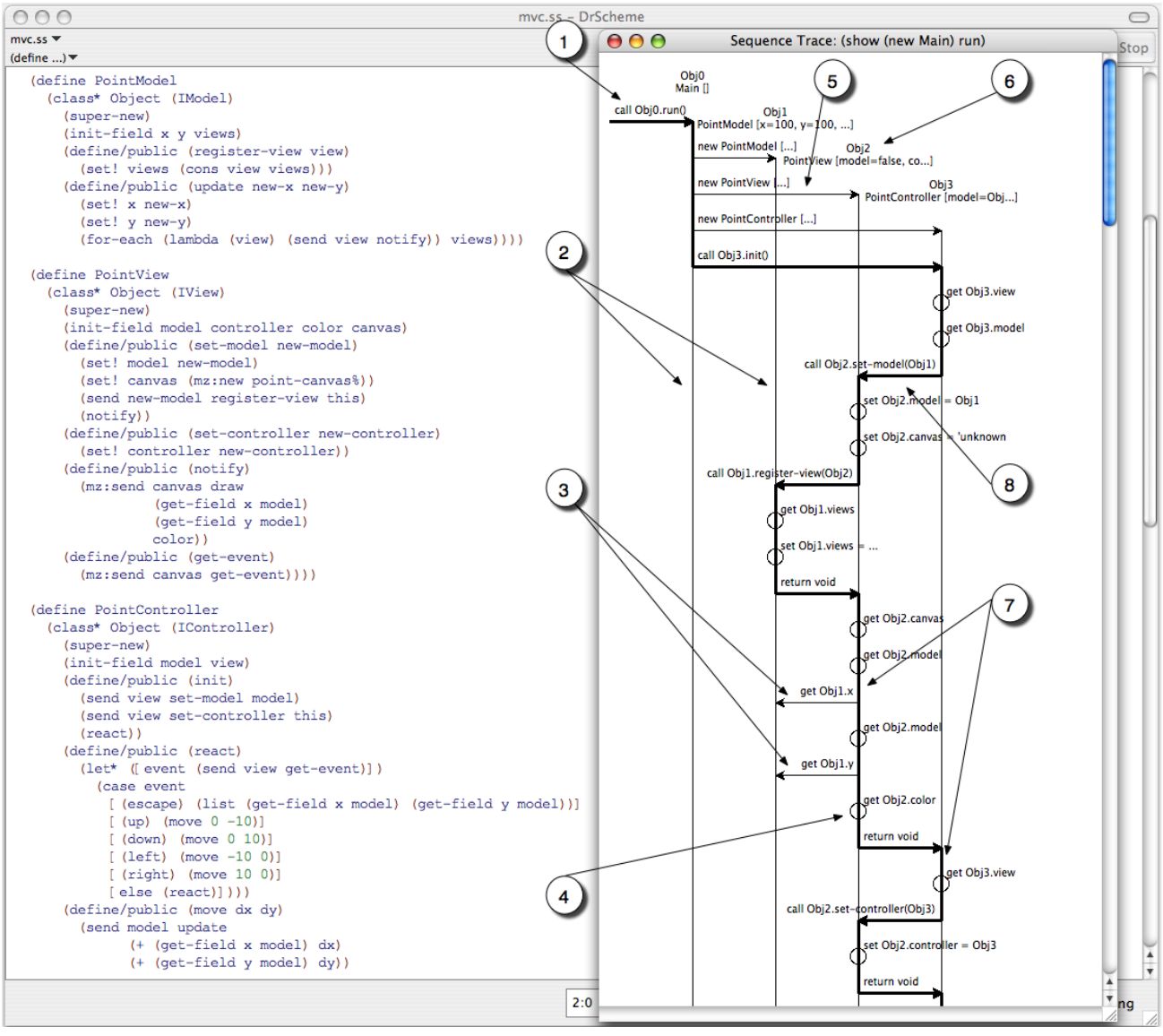


Figure 1: Graphical sequence trace of a Model-View-Controller program.

to prove most of the type soundness theorem at the upper level and applies this to one of the models from section 4. The last two sections are about related and future work.

What we do *not* do: The purpose of this paper is *not* to assign meaning to UML. We simply borrow some of UML's graphical elements. The following three sections clarify this difference implicitly; we explain it in some detail in the section on related work.

2. SEQUENCE TRACES

Good software development requires good software tools, including tools for inspecting program executions and debugging code. While many such tools exist, most are generic and don't capture the essence of object-oriented programming. In particular, debuggers often expose an execution state through stack traces and variables [25], which are procedural concepts rather than object-oriented ones. Some trace tools record events without an object-oriented model; others translate traces into abstract interpretations [10, 31, 34].

We have prototyped a tool that presents the execution of object-oriented programs in the style of UML sequence diagrams. The tool operates on the PLT Scheme class system [12]. Roughly speaking, the object-oriented portion of PLT Scheme is an untyped version of Java's class and interface system, extended with mixins in the spirit of Flatt et al [13]. Objects interact through fields, methods, constructors, and reflection. The tool captures a *sequence trace* of these interactions, reconstructing a snapshot of the program state at each object-oriented action.

Figure 1 depicts a sample sequence trace. The pictured program excerpt (on the left) is an instance of the Model-View-Controller pattern written with PLT Scheme classes. The visible portion of the trace (on the right) is a prefix of the program's execution, starting with the invocation of the `Main` object (number 1 on the figure). Following UML tradition, the vertical lines are object lifelines (number 2) representing the existence of an object. Horizontal arrows are messages (number 3) sent from one object to another; circles represent messages sent from an object to itself (number 4). New lifelines start after **new** messages (number 5). Every lifeline has a header (number 6) with a name for the object above and its class and fields below. The control flow of the execution is emphasized by the bold segments of messages and lifelines. Where an object lifeline is highlighted (number 7), that object has control. Where a message arrow is highlighted (number 8), one object transfers control to another.

Sequence traces, such as the one in Figure 1, suggest a model of computation as communication. 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 have any externally visible communication. Of course, this model is not particular to PLT Scheme; it is the core of any object-oriented programming language and deserves a formal exploration.

3. FRAMEWORK SEMANTICS

Our framework assigns semantics to object-oriented programming languages at two levels. The upper level is all about objects, their creation, their life lines, 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, and so on.

NAME^L	syntactic category NAME in language L, or a metavariable in it (by context)
$\overrightarrow{t}$	any number of elements of the form t
$c[e]$	expression e in evaluation context c
$e[v/x]$	substitution of v for free x in expression e
$d \xrightarrow{p} r$	partial functions from domain d to range r
$d \xrightarrow{f} r$	finite mappings from domain d to range r
$[\overrightarrow{a \mapsto b}]$	finite mapping of each a to b
$f[\overrightarrow{a \mapsto b}]$	finite mapping extension by juxtaposition overrides mappings in f

Figure 2: Notational conventions.

Figure 3 gives the full syntax of the upper level using the notation in Figure 2. The upper level represents the execution of an object-oriented program via a TRACE, which is a sequence of STATEs. Each state is a four-tuple, consisting of a POOL of objects, a STACK, an object reference, and an ACTION. A pool is a finite map from object references to OBJECTs. The latter have two parts: a DYNAMIC record, mapping FIELDs to VALUEs, and a STATIC record, containing all invariant parts of an object. The stack is a sequence of object references and so-called CONTinuations.

Finally, there are six kinds of messages: a **new** message, which contains an OBJECT to allocate; an **inspect** message for a designated object reference; a **get** message for a specific field in a designated object reference; a **set** message with a value for a specific field in a designated object reference; a **call** message for a specific method with a number of argument values; and a **return** message with a value. Most messages contain a continuation (using notation $\triangleright$). When an object is unable to send a message, it may signal an error instead.

In order to accommodate as many object-oriented languages as possible, this syntactic part of the upper level is parameterized over a number of language-specific sets: the text of PROGrams, FIELDs (field names); METHODs (method names); PRIMitive non-object values; STATIC; CONTs, which represent method-local continuations; andLANGERROR, the collection of language-specific errors.

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

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

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

PROG		lower level definition	Program
CONT		lower level definition	Continuation
STATIC		lower level definition	Static Record
FIELD		lower level definition	Field Name
METHOD		lower level definition	Method Name
PRIM		lower level definition	Primitive Value
LANGERROR		lower level definition	Language-specific Error
REF		countable set	Object Reference
TRACE	=	$\overrightarrow{\text{STATE}}$	Sequence Trace
STATE	=	$\langle$POOL, STACK, REF, ACTION$\rangle$	Execution State
POOL	:	REF $\xrightarrow{f}$ OBJECT	Object Pool
STACK	=	ϵ | $\langle$REF, CONT$\rangle$ STACK	Method Stack
OBJECT	=	STATIC DYNAMIC	Object Record
DYNAMIC	:	FIELD $\xrightarrow{f}$ VALUE	Dynamic Record
VALUE	=	PRIM | REF | STATIC	Value
ACTION	=	MESSAGE | ERROR	Object-Oriented Action
MESSAGE	=	**new** OBJECT $\triangleright$ CONT	Object Creation
	|	**inspect** REF $\triangleright$ CONT	Object Inspection
	|	**get** REF.FIELD $\triangleright$ CONT	Field Lookup
	|	**set** REF.FIELD := VALUE $\triangleright$ CONT	Field Update
	|	**call** REF.METHOD($\overrightarrow{\text{VALUE}}$) $\triangleright$ CONT	Method Call
	|	**return** VALUE	Method Return
RESULT	=	$\langle$POOL, ϵ, REF, **return** VALUE$\rangle$	Success Result
	|	$\langle$POOL, STACK, REF, ERROR$\rangle$	Failure Result
ERROR	=	LANGERROR | **error:reference** | **error:field**	Execution Error

Figure 3: Upper-level definition of sequence trace syntax.

method call A **call** action invokes a method in an object on a number of arguments.

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

The **new** and **set** messages update the object pool; **new** adds new objects and **set** replaces their field values. The **call** and **return** messages manipulate the stack and the reference to the controlling object, transferring control between objects. The former pushes a frame onto the stack containing the current object reference and the action's continuation; the latter removes and reinstates them later. Aside from these cases, messages leave the pool, stack, and controlling object reference unchanged.

Every message changes the action field. At each step, the rewriting system uses either the function *invoke* or *resume* to compute the next action. Both functions are parameters of the rewriting system. The former (*invoke*) maps a method call for a designated object, method name, and a sequence of arguments to an action; the latter maps a value and a continuation to an action. Both functions are partial, admitting the possibility of non-termination. Also, both functions may map their inputs to a language-specific error.

3.1 To Diagrams and Back

The formal definition of sequence traces represents their visual form, but does not resemble it. Still, it is possible to convert one form into the other (relative to internal computation). The example in Figure 5 shows a visual sequence trace next to its formal representation.

Conversion from a formal trace to a visual one proceeds state by state "down" the execution. To draw a visual sequence trace from a formal one, we start with the initial object pool: a lifeline header for each object in the pool. A lifeline header is an object's reference above its static and dynamic records. For each state in the trace, we then perform the following steps:

1. Extend the lifeline of all objects in the pool.
2. If an object is created by a **new** action, add a new object header to the right of the object pool
3. Draw an arrow for the current action. The arrow starts at the controlling object, denoted by the reference in the current state. Most messages refer to a target object, where the arrow ends. A **new** message's arrow ends at the new object header. A **return** message's arrow ends at the object on top of the stack.
4. Label the arrow with the current action. If the action has a continuation, leave it off. Continuations represent internal actions of an object and are not part of a visual sequence trace.

To construct a formal sequence trace from a picture, we need to accumulate the state of the program along the trace. The initial store is simply the collection of lifeline headers at the top of the trace. The initial stack is empty; the controlling object is the source of the first arrow and the action is the arrow's label. Any continuations must be inferred since they are not shown in visual sequence traces. For the rest of the diagram, we use these steps.

1. After a **new** message, we add the new object header to the object pool. After a **set** message, we replace the appropriate field value. The object pool remains otherwise unchanged.

Lower level functions

init	$\textsc{Prog} \longrightarrow \textsc{State}$	Constructs the initial program state.
$invoke_{\textsc{Prog}}$	$\langle \textsc{Ref}, \textsc{Object}, \textsc{Method}, \overrightarrow{\textsc{Value}} \rangle \xrightarrow{p} \textsc{Action}$	Invokes a method.
$resume_{\textsc{Prog}}$	$\langle \textsc{Cont}, \textsc{Value} \rangle \xrightarrow{p} \textsc{Action}$	Resumes a suspended computation.

Evaluation

$\textsc{Prog} \downarrow \textsc{Result}$ means $init(\textsc{Prog}) \mapsto^*_{\textsc{Prog}} \textsc{Result}$

where

Object Creation

$\langle \textsc{Pool}, \textsc{Stack}, \textsc{Ref}, \textbf{new}\ \textsc{Object} \triangleright \textsc{Cont} \rangle$
$\mapsto_{\textsc{Prog}} \langle \textsc{Pool}[\textsc{Ref}' \mapsto \textsc{Object}], \textsc{Stack}, \textsc{Ref}, resume_{\textsc{Prog}}(\textsc{Cont}, \textsc{Ref}') \rangle$
where $\textsc{Ref}' \notin dom(\textsc{Pool})$

Object Inspection

$\langle \textsc{Pool}, \textsc{Stack}, \textsc{Ref}, \textbf{inspect}\ \textsc{Ref}' \triangleright \textsc{Cont} \rangle$
$\mapsto_{\textsc{Prog}} \langle \textsc{Pool}, \textsc{Stack}, \textsc{Ref}, resume_{\textsc{Prog}}(\textsc{Cont}, \textsc{Static}) \rangle$
if $\textsc{Pool}(\textsc{Ref}') = \textsc{Static Dynamic}$

Field Lookup

$\langle \textsc{Pool}, \textsc{Stack}, \textsc{Ref}, \textbf{get}\ \textsc{Ref}'.\textsc{Field} \triangleright \textsc{Cont} \rangle$
$\mapsto_{\textsc{Prog}} \langle \textsc{Pool}, \textsc{Stack}, \textsc{Ref}, resume_{\textsc{Prog}}(\textsc{Cont}, \textsc{Value}) \rangle$
if $\textsc{Pool}(\textsc{Ref}') = \textsc{Static Dynamic}$ and $\textsc{Dynamic}(\textsc{Field}) = \textsc{Value}$

Field Update

$\langle \textsc{Pool}, \textsc{Stack}, \textsc{Ref}, \textbf{set}\ \textsc{Ref}'.\textsc{Field} := \textsc{Value} \triangleright \textsc{Cont} \rangle$
$\mapsto_{\textsc{Prog}} \langle \textsc{Pool}[\textsc{Ref}' \mapsto \textsc{Static Dynamic}[\textsc{Field} \mapsto \textsc{Value}]], \textsc{Stack}, \textsc{Ref}, resume_{\textsc{Prog}}(\textsc{Cont}, \textsc{Value}) \rangle$
if $\textsc{Pool}(\textsc{Ref}') = \textsc{Static Dynamic}$ and $\textsc{Field} \in dom(\textsc{Dynamic})$

Method Call

$\langle \textsc{Pool}, \textsc{Stack}, \textsc{Ref}, \textbf{call}\ \textsc{Ref}'.\textsc{Method}(\overrightarrow{\textsc{Value}}) \triangleright \textsc{Cont} \rangle$
$\mapsto_{\textsc{Prog}} \langle \textsc{Pool}, \langle \textsc{Ref}, \textsc{Cont} \rangle\ \textsc{Stack}, \textsc{Ref}', invoke_{\textsc{Prog}}(\textsc{Ref}', \textsc{Pool}(\textsc{Ref}'), \textsc{Method}, \overrightarrow{\textsc{Value}}) \rangle$
if $\textsc{Ref}' \in dom(\textsc{Pool})$

Method Return

$\langle \textsc{Pool}, \langle \textsc{Ref}', \textsc{Cont} \rangle\ \textsc{Stack}, \textsc{Ref}, \textbf{return}\ \textsc{Value} \rangle$
$\mapsto_{\textsc{Prog}} \langle \textsc{Pool}, \textsc{Stack}, \textsc{Ref}', resume_{\textsc{Prog}}(\textsc{Cont}, \textsc{Value}) \rangle$

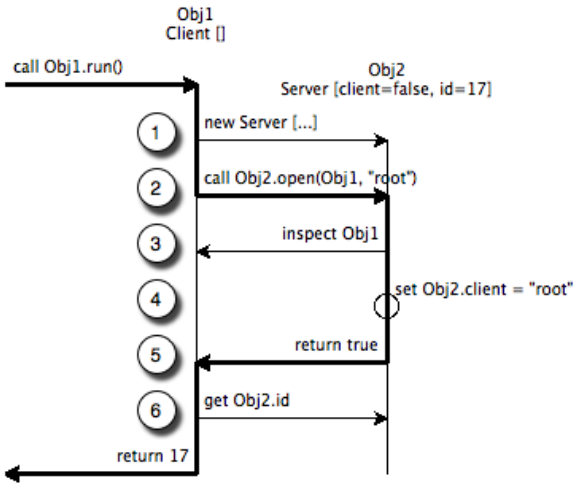
Reference Error

$\langle \textsc{Pool}, \textsc{Stack}, \textsc{Ref}, \textsc{Message} \rangle$
$\mapsto_{\textsc{Prog}} \langle \textsc{Pool}, \textsc{Stack}, \textsc{Ref}, \textbf{error:reference} \rangle$
if $\textsc{Message} = \textbf{inspect}\ \textsc{Ref}' \triangleright \textsc{Cont}$
$\quad \mid \textbf{get}\ \textsc{Ref}'.\textsc{Field} \triangleright \textsc{Cont}$
$\quad \mid \textbf{set}\ \textsc{Ref}'.\textsc{Field} := \textsc{Value} \triangleright \textsc{Cont}$
$\quad \mid \textbf{call}\ \textsc{Ref}'.\textsc{Method}(\overrightarrow{\textsc{Value}}) \triangleright \textsc{Cont}$
and $\textsc{Ref}' \notin dom(\textsc{Pool})$

Field Error

$\langle \textsc{Pool}, \textsc{Stack}, \textsc{Ref}, \textsc{Message} \rangle$
$\mapsto_{\textsc{Prog}} \langle \textsc{Pool}, \textsc{Stack}, \textsc{Ref}, \textbf{error:field} \rangle$
if $\textsc{Message} = \textbf{get}\ \textsc{Ref}'.\textsc{Field} \triangleright \textsc{Cont} \mid \textbf{set}\ \textsc{Ref}'.\textsc{Field} := \textsc{Value} \triangleright \textsc{Cont}$
and $\textsc{Pool}(\textsc{Ref}') = \textsc{Static Dynamic}$ and $\textsc{Field} \notin dom(\textsc{Dynamic})$

Figure 4: Upper-level definition of sequence trace semantics.



1. $\langle \textsc{Pool}_1, \epsilon, \text{Obj1}, \textbf{new Server } \textsc{Dynamic}_1 \triangleright \textsc{Cont}_1 \rangle$
   where $\textsc{Pool}_1 = [\text{Obj1} \mapsto \text{Client } []]$
   and $\textsc{Dynamic}_1 = [\text{client} \mapsto \textbf{false}, \text{id} \mapsto 17]$

2. $\langle \textsc{Pool}_2, \epsilon, \text{Obj1}, \textbf{call Obj2.open}(\text{Obj1}, \text{“root”}) \triangleright \textsc{Cont}_2 \rangle$
   where $\textsc{Pool}_2 = \textsc{Pool}_1[\text{Obj2} \mapsto \text{Server } \textsc{Dynamic}_1]$

3. $\langle \textsc{Pool}_2, \langle \text{Obj1}, \textsc{Cont}_2 \rangle, \text{Obj2}, \textbf{inspect Obj1} \triangleright \textsc{Cont}_3 \rangle$

4. $\langle \textsc{Pool}_2, \langle \text{Obj1}, \textsc{Cont}_2 \rangle, \text{Obj2},$
   $\textbf{set Obj2.client} := \text{“root”} \triangleright \textsc{Cont}_4 \rangle$

5. $\langle \textsc{Pool}_3, \langle \text{Obj1}, \textsc{Cont}_2 \rangle, \text{Obj2}, \textbf{return true} \rangle$
   where $\textsc{Pool}_3 = \textsc{Pool}_2[\text{Obj2} \mapsto \text{Server } \textsc{Dynamic}_2]$
   and $\textsc{Dynamic}_2 = \textsc{Dynamic}_1[\text{client} \mapsto \text{“root”}]$

6. $\langle \textsc{Pool}_3, \epsilon, \text{Obj1}, \textbf{get Obj2.id} \triangleright \textsc{Cont}_5 \rangle$

Figure 5: Correlation between visual and formal sequence traces.

2. After a **call** message, we add the current object reference to the stack. We install the message's receiver as the new object reference. After a **return** message, we remove a reference from the top of the stack and install it as the new object reference. The stack and object reference are otherwise unchanged.

3. The new state's action is always the label on the arrow in the next row. The action's continuation must be inferred.

4. FRAMEWORK INSTANTIATIONS

The framework accommodates a variety of programming language models. To demonstrate its flexibility in this regard, we demonstrate how to formulate three radically different, well-known models as lower levels of our framework: Abadi and Cardelli's object calculus [1] in section 4.1, Flatt et alii's ClassicJava [13] in section 4.2, and Castagna et alii's $\lambda\&$-calculus [5] in section 4.3. The first is a pure calculus of objects; the second is a semantics of a class-based object-oriented language in the spirit of Java, and the last is a model used to formalize multiple dispatch.

Formulating a language semantics in our framework has two purposes. Conceptually, it helps a semanticist determine what is object-oriented about a language and what is procedural. This point comes out especially well with our reformulation of ClassicJava and the modeling of super-method calls. It is strictly impossible in conventional models to discuss this issue. Pragmatically, the formulation of a language in our framework is the first step for instantiating our tracing tool for this language. That is, once we have a framework-based semantics, it is straightforward to instrument the compiler and to combine it with our tracing component to get a new debugging tool.

4.1 Object Calculus Sequence Traces

Abadi and Cardelli's object calculus [1] is the quintessential model of object-oriented computation; its role is similar to that of the λ-calculus in functional programming. The purpose of the calculus is to model objects and methods; in particular, objects don't have fields, only constant-value methods. The calculus has three actions: object creation, method invocation, and method override. Object creation specifies the names and bodies of methods. Method invocation specifies an object (the receiver) and a method name; methods have no external arguments. Method override provides an object, a method name, and the new body for that method. This creates a new object with the updated method body, inheriting all other methods from the original object.

This model is a natural fit for our framework. The bodies of all methods become the static record of an object. Object creation and method invocation are already framework messages. Method override takes two messages: see below.

Figures 6 shows the lower level parameters defining object calculus syntax. Programs in the object calculus are expressions. Continuations are evaluation contexts over the expressions. An expression can be a variable, reference, method call, method override, or object constructor. An object's static record maps its method names to method bodies. Finally, the lower level also defines method errors.

Figure 7 defines the semantics of the object calculus. Program initialization stores the initial expression in a method body and invokes that method as the first action. The *resume* and *invoke* functions are defined in terms of an *eval* function, operating on expressions via a reduction relation.

Object creation and method invocation evaluate directly to the appropriate messages; all method bodies are stored in an object's static record, and extracted when invoked. Method override is modelled as two messages. The first extracts the static record of the receiver to copy the inherited methods. The second constructs the updated object.

4.2 ClassicJava Sequence Traces

ClassicJava [13] is another widely used model of object-oriented computation. In contrast to the object calculus, it is class-based, including most of Java's core features such as classes, interfaces, inheritance, method overriding, and typecasts. Naturally, this does not match our framework perfectly, but the transition is remarkably smooth.

Figure 8 shows the lower-level parameters defining Clas-

$\textsc{Prog}^\varsigma$	=	$\textsc{Expr}^\varsigma$	Object Calculus Program
$\textsc{Cont}^\varsigma$	=	$[]$	Empty Context
	|	$\textsc{Cont}^\varsigma.\textsc{Method}^\varsigma$	Method Call Receiver
	|	$\textsc{Cont}^\varsigma.\textsc{Method}^\varsigma \leftarrow \textsc{Body}^\varsigma$	Method Override Receiver
$\textsc{Static}^\varsigma$	:	$\textsc{Method}^\varsigma \xrightarrow{f} \textsc{Body}^\varsigma$	Static Record
$\textsc{Field}^\varsigma$	=	$\emptyset$	Field Names
$\textsc{Method}^\varsigma$	=	method label	Method Names
$\textsc{Prim}^\varsigma$	=	$\emptyset$	Primitive Values
$\textsc{LangError}^\varsigma$	=	**error:method**	Method Error
where			
$\textsc{Body}^\varsigma$	=	$\varsigma(\textsc{Var}).\textsc{Expr}^\varsigma$	Method Body
$\textsc{Expr}^\varsigma$	=	$\textsc{Value} \mid \textsc{Var}$	Simple Expressions
	|	$\textsc{Expr}^\varsigma.\textsc{Method}^\varsigma$	Method Call
	|	$\textsc{Expr}^\varsigma.\textsc{Method}^\varsigma \leftarrow \textsc{Body}^\varsigma$	Method Override
	|	$\{\overrightarrow{\textsc{Method}^\varsigma = \textsc{Body}^\varsigma}\}$ where $(\textsc{Method}_i^\varsigma$ distinct)	Object Contruction

Figure 6: Lower-level parameters for object calculus syntax.

$init^\varsigma(\textsc{Expr})$	=	$\langle[\textsc{Ref}_0 \mapsto [\textsc{Method}_0 \mapsto \varsigma(\textsc{Var}_0).\textsc{Expr}]],$ $\epsilon, \textsc{Ref}_0, \textbf{call}\ \textsc{Ref}_0.\textsc{Method}_0() \triangleright []\rangle$
$resume^\varsigma_{\textsc{Prog}}(\textsc{Cont}, \textsc{Value})$	=	$eval^\varsigma(\textsc{Cont}[\textsc{Value}])$
$invoke^\varsigma_{\textsc{Prog}}(\textsc{Ref}, \textsc{Static}, \textsc{Method}, \langle\rangle)$	=	$eval^\varsigma(\textsc{Expr}[\textsc{Ref}/\textsc{Var}])$ where $\textsc{Static}(\textsc{Method}) = \varsigma(\textsc{Var}).\textsc{Expr}$
$invoke^\varsigma_{\textsc{Prog}}(\textsc{Ref}, \textsc{Static}, \textsc{Method}, \langle\rangle)$	=	**error:method** where $\textsc{Method} \notin dom(\textsc{Static})$
$invoke^\varsigma_{\textsc{Prog}}(\textsc{Ref}, \textsc{Static}, \textsc{Method}, \overrightarrow{\textsc{Value}})$	=	**error:arity** where $\lvert\overrightarrow{\textsc{Value}}\rvert > 0$
• $eval^\varsigma : \textsc{Expr} \longrightarrow \textsc{Action}$ **is:**		
$eval^\varsigma(\textsc{Value})$	=	**return** $\textsc{Value}$
$eval^\varsigma(\textsc{Cont}[\{\overrightarrow{\textsc{Method} = \textsc{Body}}\}])$	=	**new** $[\overrightarrow{\textsc{Method} \mapsto \textsc{Body}}] \triangleright \textsc{Cont}$
$eval^\varsigma(\textsc{Cont}[\textsc{Ref.Method}])$	=	**call** $\textsc{Ref.Method}() \triangleright \textsc{Cont}$
$eval^\varsigma(\textsc{Cont}[\textsc{Ref.Method} \leftarrow \textsc{Body}])$	=	**inspect** $\textsc{Ref} \triangleright \textsc{Cont}[[].\textsc{Method} \leftarrow \textsc{Body}]$
$eval^\varsigma(\textsc{Cont}[\textsc{Static.Method} \leftarrow \textsc{Body}])$	=	**new** $\textsc{Static}[\textsc{Method} \mapsto \textsc{Body}] \triangleright \textsc{Cont}$ where $\textsc{Method} \in dom(\textsc{Static})$
$eval^\varsigma(\textsc{Cont}[\textsc{Static.Method} \leftarrow \textsc{Body}])$	=	**error:method** where $\textsc{Method} \notin dom(\textsc{Static})$

Figure 7: Lower-level parameters for object calculus semantics.

DEF^{CJ}		class or interface definition	Definition
CLASS^{CJ}		class name	Class
TNAME^{CJ}		class or interface name	Type
FIELD^{ℓ}		countable set	Field Label
METHOD^{ℓ}		countable set	Method Label
PROG^{CJ}	$=$	$\overrightarrow{\text{DEF}^{\text{CJ}}}$ **new** $\text{CLASS}^{\text{CJ}}.\text{METHOD}^{\ell}()$	Program
CONT^{CJ}	$=$	[]	Empty Context
	$\mid$	$\text{CONT}^{\text{CJ}} : \text{CLASS}^{\text{CJ}}.\text{FIELD}^{\ell}$	Field Lookup Receiver
	$\mid$	$\text{CONT}^{\text{CJ}} : \text{CLASS}^{\text{CJ}}.\text{FIELD}^{\ell} = \text{EXPR}^{\text{CJ}}$	Field Update Receiver
	$\mid$	$\text{VALUE} : \text{CLASS}^{\text{CJ}}.\text{FIELD}^{\ell} = \text{CONT}^{\text{CJ}}$	Field Update Value
	$\mid$	$\text{CONT}^{\text{CJ}}.\text{METHOD}^{\ell}(\overrightarrow{\text{EXPR}^{\text{CJ}}})$	Method Call Receiver
	$\mid$	$\text{VALUE}.\text{METHOD}(\overrightarrow{\text{VALUE}}\ \text{CONT}^{\text{CJ}}\ \overrightarrow{\text{EXPR}^{\text{CJ}}})$	Method Call Argument
	$\mid$	**super** $= \text{REF} : \text{CLASS}^{\text{CJ}}.\text{METHOD}^{\ell}(\overrightarrow{\text{VALUE}}\ \text{CONT}^{\text{CJ}}\ \overrightarrow{\text{EXPR}^{\text{CJ}}})$	Super Call Argument
	$\mid$	**view** $\text{TNAME}\ \text{CONT}^{\text{CJ}}$	Typecast Argument
	$\mid$	**view** $\text{CONT}^{\text{CJ}} \leq \text{TNAME}\ \text{VALUE}$	Elaborated Typecast Class
	$\mid$	**let** $\text{VAR} = \text{CONT}^{\text{CJ}}$ **in** EXPR^{CJ}	Let Binding R.H.S.
$\text{STATIC}^{\text{CJ}}$	$=$	CLASS^{CJ}	Static Record
FIELD^{CJ}	$=$	$\langle \text{CLASS}^{\text{CJ}}, \text{FIELD}^{\ell} \rangle$	Field Name
$\text{METHOD}^{\text{CJ}}$	$=$	$\text{METHOD}^{\ell} \mid \langle \text{CLASS}^{\text{CJ}}, \text{METHOD}^{\ell} \rangle$	Method Name
PRIM^{CJ}	$=$	**null**	Primitive Value
$\text{LANGERROR}^{\text{CJ}}$	$=$	**error:method**	Method Error
	$\mid$	**error:null**	Null Error
	$\mid$	**error:typecast**	Typecast Error
where			
EXPR^{CJ}	$=$	REF	Object Reference
	$\mid$	**new** CLASS^{CJ}	Object Creation
	$\mid$	VAR	Variable
	$\mid$	**this**	Self Reference
	$\mid$	**null**	Null Value
	$\mid$	$\text{EXPR}^{\text{CJ}} : \text{CLASS.FIELD}$	Field Lookup
	$\mid$	$\text{EXPR}^{\text{CJ}} : \text{CLASS.FIELD} = \text{EXPR}^{\text{CJ}}$	Field Update
	$\mid$	$\text{EXPR}^{\text{CJ}}.\text{METHOD}(\overrightarrow{\text{EXPR}^{\text{CJ}}})$	Method Call
	$\mid$	**super** $=$ **this** $: \text{CLASS.METHOD}(\overrightarrow{\text{EXPR}^{\text{CJ}}})$	Super Call
	$\mid$	**view** $\text{TNAME}\ \text{EXPR}^{\text{CJ}}$	Typecast
	$\mid$	**view** $\text{CLASS} \leq \text{TNAME}\ \text{EXPR}^{\text{CJ}}$	Elaborated Typecast
	$\mid$	**let** $\text{VAR} = \text{EXPR}^{\text{CJ}}$ **in** EXPR^{CJ}	Let Binding

Figure 8: Lower level parameters for ClassicJava syntax.

$\text{TNAME} \leq^{\text{CJ}}_{\text{PROG}} \text{TNAME}'$	$\iff$	TNAME is a subtype of TNAME'
$\langle \text{CLASS}', \text{FIELD}, \text{TNAME} \rangle \in^{\text{CJ}}_{\text{PROG}} \text{CLASS}$	$\iff$	class CLASS includes or inherits field FIELD of type TNAME defined in class CLASS'
$\langle \text{METHOD}, \overrightarrow{\text{TNAME}} \to \text{TNAME}, \overrightarrow{\text{VAR}}, \text{EXPR} \rangle \in^{\text{CJ}}_{\text{PROG}} \text{CLASS}$	$\iff$	class CLASS includes or inherits a definition for method METHOD with type $\overrightarrow{\text{TNAME}} \to \text{TNAME}$, input variables $\overrightarrow{\text{VAR}}$, and method body EXPR.

Figure 9: ClassicJava semantic relations.

$$
\begin{array}{lcl}
\mathit{init}^{\mathrm{CJ}}(\overrightarrow{\text{DEF}}\ \textbf{new}\ \text{CLASS.METHOD}()) & = & \langle[\text{REF}_0 \mapsto \mathit{newobject}^{\mathrm{CJ}}(\text{CLASS})], \\
 & & \ \epsilon, \text{REF}_0, \textbf{call}\ \text{REF}_0.\text{METHOD}() \triangleright []\rangle \\
\mathit{resume}^{\mathrm{CJ}}_{\mathrm{PROG}}(\text{CONT}, \text{VALUE}) & = & \mathit{eval}^{\mathrm{CJ}}_{\mathrm{PROG}}(\text{CONT}[\text{VALUE}]) \\
\mathit{invoke}^{\mathrm{CJ}}_{\mathrm{PROG}}(\text{REF}, \text{CLASS DYNAMIC}, \text{METHOD}, \overrightarrow{\text{VALUE}}) & = & \mathit{evalmethod}^{\mathrm{CJ}}_{\mathrm{PROG}}(\text{REF}, \text{CLASS}, \text{METHOD}, \overrightarrow{\text{VALUE}}) \\
\mathit{invoke}^{\mathrm{CJ}}_{\mathrm{PROG}}(\text{REF}, \text{OBJECT}, \langle\text{CLASS}, \text{METHOD}\rangle, \overrightarrow{\text{VALUE}}) & = & \mathit{evalmethod}^{\mathrm{CJ}}_{\mathrm{PROG}}(\text{REF}, \text{CLASS}, \text{METHOD}, \overrightarrow{\text{VALUE}})
\end{array}
$$

• $\mathit{newobject}^{\mathrm{CJ}}$: CLASS $\longrightarrow$ OBJECT is:

$$
\begin{array}{lcl}
\mathit{newobject}^{\mathrm{CJ}}(\text{CLASS}) & = & \text{CLASS}\ [\overrightarrow{\langle\text{CLASS}', \text{FIELD}\rangle \mapsto \textbf{null}}] \\
 & & \text{where} \overrightarrow{\langle\text{CLASS}', \text{FIELD}\rangle} = \{\langle\text{CLASS}', \text{FIELD}\rangle \mid \langle\text{CLASS}', \text{FIELD}, \text{TNAME}\rangle \in^{\mathrm{CJ}}_{\mathrm{PROG}} \text{CLASS}\}
\end{array}
$$

• $\mathit{evalmethod}^{\mathrm{CJ}}$: $\langle$REF, CLASS, METHOD, $\overrightarrow{\text{VALUE}}\rangle$ $\longrightarrow$ ACTION is:
$\mathit{evalmethod}^{\mathrm{CJ}}_{\mathrm{PROG}}(\text{REF}, \text{CLASS}, \text{METHOD}, \overrightarrow{\text{VALUE}})$ = ACTION by the following table:

ACTION	where
$\mathit{eval}^{\mathrm{CJ}}_{\mathrm{PROG}}(\text{EXPR}[\text{REF}/\textbf{this}][\overrightarrow{\text{VALUE}/\text{VAR}}])$	$\langle\text{METHOD}, \overrightarrow{\text{TNAME}} \rightarrow \text{TNAME}, \overrightarrow{\text{VAR}}, \text{EXPR}\rangle \in^{\mathrm{CJ}}_{\mathrm{PROG}} \text{CLASS}$ and $\lvert\overrightarrow{\text{VALUE}}\rvert = \lvert\overrightarrow{\text{VAR}}\rvert$
error:method	otherwise

• $\mathit{eval}^{\mathrm{CJ}}$: EXPR $\longrightarrow$ ACTION is:
$\mathit{eval}^{\mathrm{CJ}}_{\mathrm{PROG}}(\text{EXPR})$ = ACTION where EXPR $\mapsto^{*}_{\mathrm{PROG}}$ EXPR$'$ by the following table:

ACTION	EXPR$'$
return VALUE	VALUE
new $\mathit{newobject}^{\mathrm{CJ}}$(CLASS) $\triangleright$ CONT	CONT[**new** CLASS]
get REF.$\langle$CLASS, FIELD$\rangle$ $\triangleright$ CONT	CONT[REF : CLASS.FIELD]
set REF.$\langle$CLASS, FIELD$\rangle$:= VALUE $\triangleright$ CONT	CONT[REF : CLASS.FIELD = VALUE]
call REF.METHOD($\overrightarrow{\text{VALUE}}$) $\triangleright$ CONT	CONT[REF.METHOD($\overrightarrow{\text{VALUE}}$)]
call REF.$\langle$CLASS, METHOD$\rangle$($\overrightarrow{\text{VALUE}}$) $\triangleright$ CONT	**super** = REF : CLASS.METHOD($\overrightarrow{\text{VALUE}}$)
inspect REF $\triangleright$ CONT[**view** [] $\leq$ TNAME REF]	CONT[**view** TNAME REF]
error:typecast	CONT[**view** TNAME **null**]
error:typecast	CONT[**view** CLASS $\leq$ TNAME VALUE] if CLASS$\not\leq^{\mathrm{CJ}}_{\mathrm{PROG}}$TNAME
error:null	CONT[**null** : CLASS.FIELD]
error:null	CONT[**null** : CLASS.FIELD = VALUE]
error:null	CONT[**null**.METHOD($\overrightarrow{\text{VALUE}}$)]

• $\mapsto^{\mathrm{CJ}}_{\mathrm{PROG}}$: EXPR $\xrightarrow{p}$ EXPR is:

$$
\begin{array}{lcl}
\text{CONT}[\textbf{let}\ \text{VAR} = \text{VALUE}\ \textbf{in}\ \text{EXPR}] & \mapsto^{\mathrm{CJ}}_{\mathrm{PROG}} & \text{CONT}[\text{EXPR}[\text{VALUE}/\text{VAR}]] \\
\text{CONT}[\textbf{view}\ \text{CLASS} \leq \text{TNAME VALUE}] & \mapsto^{\mathrm{CJ}}_{\mathrm{PROG}} & \text{CONT}[\text{VALUE}] \text{ if } \text{CLASS} \leq^{\mathrm{CJ}}_{\mathrm{PROG}} \text{TNAME}
\end{array}
$$

Figure 10: Lower level parameters for ClassicJava semantics.

$$
\text{METHOD}^{\mathrm{CJ}} = \text{METHOD}^{\ell} \mid \sout{\langle\text{CLASS}^{\mathrm{CJ}}, \text{METHOD}^{\ell}\rangle}
$$

$$
\sout{\mathit{invoke}^{\mathrm{CJ}}_{\mathrm{PROG}}(\text{REF}, \text{OBJECT}, \langle\text{CLASS}, \text{METHOD}\rangle, \overrightarrow{\text{VALUE}}) = \mathit{evalmethod}^{\mathrm{CJ}}_{\mathrm{PROG}}(\text{REF}, \text{CLASS}, \text{METHOD}, \overrightarrow{\text{VALUE}})}
$$

$$
\mathit{eval}^{\mathrm{CJ}}_{\mathrm{PROG}}(\text{EXPR}) = \begin{cases}
\cdots \\
\sout{\textbf{call}\ \text{REF}.\langle\text{CLASS}, \text{METHOD}\rangle(\overrightarrow{\text{VALUE}}) \triangleright \text{CONT}} \\
\sout{\text{if EXPR} \mapsto^{*}_{\mathrm{PROG}} \text{CONT}[\textbf{super} = \text{REF} : \text{CLASS.METHOD}(\overrightarrow{\text{VALUE}})]} \\
\text{CONT}[\mathit{evalmethod}^{\mathrm{CJ}}_{\mathrm{PROG}}(\text{REF}, \text{CLASS}, \text{METHOD}, \overrightarrow{\text{VALUE}})] \\
\ \text{if EXPR} \mapsto^{*}_{\mathrm{PROG}} \text{CONT}[\textbf{super} = \text{REF} : \text{CLASS.METHOD}(\overrightarrow{\text{VALUE}})]
\end{cases}
$$

Figure 11: Changes for an alternate interpretation of ClassicJava.

sicJava syntax. Programs in ClassicJava are a sequence of definitions followed by a method invocation expression; this resembles the call to **main** in Java. Continuations are evaluation contexts over expressions. 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**. ClassicJava defines errors for method invocation, null dereference, and failed typecasts.

Our representation of ClassicJava is nearly identical to the original; there are only two changes. We restrict the form of the initial expression to invoke a method of no arguments on a newly constructed object. By making this standalone expression a special case, all meaningful computation takes place within an object.

The other difference lies in the **view** expression. A typecast operation requires an **inspect** message to access an object's runtime type; the message's continuation must accept the object's class and compare it to a static type. ClassicJava has no appropriate evaluation context. We add an elaborated form of typecasts to expressions and continuations. This form adds an explicit position for the class of the object. Our semantics uses this as an intermediate step in evaluating **view** expressions.

Figure 10 defines the semantics of ClassicJava relative to the type relations described in Figure 9. Program initialization installs the receiver of the initial method call as the controlling object and issues the method call. The $resume^{\mathrm{CJ}}$ and $invoke^{\mathrm{CJ}}$ functions are defined in terms of total functions $evalmethod^{\mathrm{CJ}}$, $eval^{\mathrm{CJ}}$, and partial function $\mapsto^{\mathrm{CJ}}$.

Method invocation uses $evalmethod^{\mathrm{CJ}}$ for dispatch. The $evalmethod^{\mathrm{CJ}}$ function looks up the appropriate method in the program's class definitions. It substitutes the method's receiver and parameters, then calls $eval^{\mathrm{CJ}}$ to evaluate the expression.

The ClassicJava step relation ($\mapsto^{\mathrm{CJ}}$) performs all purely object-internal computations. It reduces **let** expressions by substitution and completes successful typecasts by replacing an elaborated **view** expression with its argument.

The $eval^{\mathrm{CJ}}$ function first performs all internal reductions on its argument, then operates by cases on the result. Object creation, field lookup and mutation, method calls and method returns all generate framework actions of the same kind. Unelaborated **view** expressions produce inspection actions, using the elaborated typecast expression as shown above. The $eval^{\mathrm{CJ}}$ function signals an error for all null dereferences and typecast failures.

Calls to an object's superclass generate method call actions; the method name includes the superclass name for method dispatch, which distinguishes it from the current method.

4.2.1 Alternate Interpretation of ClassicJava

Our parameterization of the sequence trace framework for ClassicJava answers the question: "what parts of ClassicJava are object-oriented?" In the semantics above, the answer is clear: object creation, field lookup and mutation, method calls, method returns, and superclass method calls.

We can now consider the merits of this interpretation. The most debatable result is 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 ClassicJava. In the prior semantics, $eval^{\mathrm{CJ}}$ turns a **super** expression into a method call. A method call uses $invoke^{\mathrm{CJ}}$ which uses $evalmethod^{\mathrm{CJ}}$. We can rewrite $eval^{\mathrm{CJ}}$ to call $evalmethod^{\mathrm{CJ}}$ directly in the **super** rule. No object-oriented action is created. The extra clauses for $\textsc{Method}^{\mathrm{CJ}}$ and $evalmethod^{\mathrm{CJ}}$ that were used for superclass calls can be removed. These modifications to ClassicJava are shown in Figure 11.

Now that we have two different semantics for ClassicJava, we can compare them to see what the tradeoffs are; different applications of the ClassicJava model can use either interpretations as appropriate.

4.3 Multiple Dispatch for Sequence Traces

Method dispatch is an important feature of many object-oriented languages. They extend the usual receiver dispatch with some form of *multiple dispatch*, in which a method's execution is determined by its arguments as well as its receiver. The $\lambda\&$-calculus of Castagna et alii [5] provides a model for single-argument dispatch. We sketch a variant on the $\lambda\&$-calculus in the sequence trace framework. We can then simulate multiple dispatch with higher-order functions or extend the language with multiple function arguments to achieve multiple dispatch.

Figure 4.3 shows the syntax of our dispatch calculus. It extends the λ-calculus with type annotations on functions, an arbitrary set of typed primitive values, and overload expressions which join several typed functions together.

Programs in this setting are expressions. Continuations are evaluation contexts. Objects are functions of both the simple and overloaded kind; an overloaded function's static record maps each branch's input type to the branch's implementation and result type. The language has a single method: function application. The dispatch calculus assumes an arbitrary set of typed primitive values.

Expressions in this language include variables, values, simple and overloaded function abstractions, function application and dispatch. Simple function expressions are annotated with input and output types. Overloaded functions contain a number of function values along with their input and output types. Function application contains a function and its argument. Dispatch expressions are generated during evaluation; they represent the application of an overloaded function to an argument of an explicit type.

We provide only an informal discussion of the operations in this language. The evaluation of function and overload expressions constructs new objects. Function application invokes the operator's **apply** method on the argument. We split dispatch on overloaded functions into two steps. The **typeof** operator determines the type of an argument, issuing an **inspect** message when given a reference. The dispatch operation itself is given the set of branches, the argument, and the argument's type; it applies the single most appropriate branch to the argument or signals a dispatch error.

Operations such as multiple dispatch are not inherently object-oriented. We have implemented argument dispatch entirely internal to objects without adding anything to the upper level of our framework. In general, we can interpret new object-oriented features without resorting to changing our framework.

$$
\begin{array}{lcll}
\textsc{Prog}^{\&} & = & \textsc{Expr}^{\&} & \text{Programs} \\
\textsc{Cont}^{\&} & = & [] & \text{Empty Context} \\
 & | & (\textsc{Cont}^{\&}\ \textsc{Expr}^{\&}) & \text{Function Context} \\
 & | & (\textsc{Value}^{\&}\ \textsc{Cont}^{\&}) & \text{Argument Context} \\
 & | & (\textsc{Overload}^{\&}\ (\textbf{typeof}\ \textsc{Cont}^{\&})\ \textsc{Value}^{\&}) & \text{Type Context} \\
 & | & \begin{array}{l}[\overrightarrow{\textsc{Value}^{\&} : \textsc{Type}^{\&} \longrightarrow \textsc{Type}^{\&}} \\ \overrightarrow{\textsc{Cont}^{\&} : \textsc{Type}^{\&} \longrightarrow \textsc{Type}^{\&}} \\ \overrightarrow{\textsc{Expr}^{\&} : \textsc{Type}^{\&} \longrightarrow \textsc{Type}^{\&}}]\end{array} & \text{Overload Context} \\
\textsc{Static}^{\&} & = & \textsc{Function}^{\&} \mid \textsc{Overload}^{\&} & \text{Static Record} \\
\textsc{Field}^{\&} & = & \emptyset & \text{Field Names} \\
\textsc{Method}^{\&} & = & \textbf{apply} & \text{Method Name} \\
\textsc{Prim}^{\&} & & \text{typed constants} & \text{Primitive Values} \\
\textsc{LangError}^{\&} & = & \textbf{error:method} \mid \textbf{error:dispatch} & \lambda\&\text{-calculus Errors}
\end{array}
$$

where

$$
\begin{array}{lcll}
\textsc{Expr}^{\&} & = & \textsc{Var}^{\textsc{Type}^{\&}} & \text{Variable} \\
 & | & \textsc{Value} & \text{Value} \\
 & | & \lambda\textsc{Var}^{\textsc{Type}^{\&}} : \textsc{Type}^{\&}.\textsc{Expr}^{\&} & \text{Typed Function} \\
 & | & [\overrightarrow{\textsc{Expr}^{\&} : \textsc{Type}^{\&} \longrightarrow \textsc{Type}^{\&}}] & \text{Overloaded Function} \\
 & | & (\textsc{Expr}^{\&}\ \textsc{Expr}^{\&}) & \text{Function Application} \\
 & | & (\textsc{Expr}^{\&}\ \textsc{TExpr}^{\&}\ \textsc{Expr}^{\&}) & \text{Overload Dispatch} \\
\textsc{TExpr}^{\&} & = & \textsc{Type}^{\&} \mid (\textbf{typeof}\ \textsc{Expr}^{\&}) & \text{Type Expressions} \\
\textsc{Type}^{\&} & = & \textsc{Base}^{\&} & \text{Base Type} \\
 & | & [\overrightarrow{\textsc{Type}^{\&} \longrightarrow \textsc{Type}^{\&}}] & \text{Overloaded Function Type} \\
\textsc{Function}^{\&} & = & \langle \textsc{Var}^{\textsc{Type}^{\&}}, \textsc{Type}^{\&}, \textsc{Expr}^{\&} \rangle & \text{Function Record} \\
\textsc{Overload}^{\&} & : & \textsc{Type}^{\&} \xrightarrow{f} \langle \textsc{Type}^{\&}, \textsc{Value}^{\&} \rangle & \text{Overload Record}
\end{array}
$$

Figure 12: Lower-level paramters for modified $\lambda\&$-calculus syntax.

5. SEQUENCE TRACE 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 overall on which the lower-level insist. For example, in the case of ClassicJava, the lower level cannot rule out `null` pointer errors and must therefore raise the relevant exceptions.

Each level defines a portion of the type judgments. The upper level relies on the lower level judgments and vice versa. Naturally, the uniform type soundness theorem holds only if the lower-level elements of the type system satisfy certain constraints.

The lower level parameters include several sets, functions, and type judgments, shown together with the constraints they must satisfy in Figure 13. The set Type defines types for the languages values; ObjType defines the subset of Type representing the types of objects. The subset OK-Error of LangError distinguishes the runtime exceptions that well-typed programs may throw.

The relation $\leq$ 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 lower level type system must provide type judgments for programs, continuations, and primitive values. It must also provide a type judgment for static records when used in the context of an object record (as opposed to their type as a value).

The type rules Init, Resume, and Invoke constrain the lower-level framework functions of the same names. 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 lower level type system must guarantee these rules, while the upper level relies on them for a parametric soundness proof.

The upper-level type system defines type judgments for everything else: program states, object pools, stacks, references, static records (as values), object records, dynamic records, and actions of both the message and error variety. Figure 14 shows the full set of upper-level judgments and type rules.

5.1 Framework Soundness

The type system satisfies a conventional type soundness theorem. Its statement, however, assumes that lower-level exceptions are typed.

Theorem 1. *If* $\vdash^{\ell}$ Prog : Type*, then either* Prog $\uparrow$ *(diverges) or* Prog $\downarrow$ Result *and* Prog $\vdash^{u}$ Result : Type.

Proof. The proof uses progress and preservation lemmas, adapting the method of Wright and Felleisen [36]. By the Init rule, the initial state of any program preserves the program's type. Lemmas 1 and 2 establish the soundness of evaluation at all subsequent states. □

Lemma 1. *If* Prog $\vdash^{u}$ State : Type*, then either* State *is a result state or* State $\mapsto_{\text{Prog}}$ State$'$ *for some* State$'$.

Proof. The proof follows by cases on State. The step relation $\mapsto$ has a clause for every non-result state. □

Sets and functions:

$\textsc{Type}$		any set	Value types
$\textsc{ObjType}$	$\subseteq$	$\textsc{Type}$	Object types
$\textsc{OKError}$	$\subseteq$	$\textsc{LangError}$	Allowable runtime errors
$\leq_{\textsc{Prog}}$		partial order on $\textsc{Type}$	Subtype relation
$\mathit{fields}_{\textsc{Prog}}$	:	$\textsc{ObjType} \longrightarrow (\textsc{Field} \xrightarrow{f} \textsc{Type})$	Produces an object type's field signatures.
$\mathit{methods}_{\textsc{Prog}}$	:	$\textsc{ObjType} \longrightarrow (\textsc{Method} \xrightarrow{f} \langle\overrightarrow{\textsc{Type}}, \textsc{Type}\rangle)$	Produces an object type's method signatures.
$\mathit{metatype}_{\textsc{Prog}}$	:	$\textsc{ObjType} \longrightarrow \textsc{Type}$	Given an object's type, produces the type of its static record.

Type judgments:

$\vdash^{\ell} \textsc{Prog} : \textsc{Type}$	Program $\textsc{Prog}$ has type $\textsc{Type}$.
$\textsc{Prog}, \textsc{Pool} \vdash^{\ell} \textsc{Cont} : \textsc{Type}_1 \xrightarrow{c} \textsc{Type}_2$	Continuation $\textsc{Cont}$ produces an action of type $\textsc{Type}_2$ when given input of type $\textsc{Type}_1$.
$\textsc{Prog}, \textsc{Pool} \vdash^{\ell}_{\mathsf{o}} \textsc{Static} : \textsc{ObjType}$	Static record $\textsc{Static}$ is well-formed in an object of type $\textsc{ObjType}$.
$\textsc{Prog}, \textsc{Pool} \vdash^{\ell}_{\mathsf{v}} \textsc{Prim} : \textsc{Type}$	Primitive value $\textsc{Prim}$ has type $\textsc{Type}$.

Satisfying:

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

$$\textsc{Resume}\ \frac{\begin{array}{c}\textsc{Prog}, \textsc{Pool} \vdash^{u\ell}_{\mathsf{v}} \textsc{Value} : \textsc{Type}_1 \\ \textsc{Prog}, \textsc{Pool} \vdash^{\ell} \textsc{Cont} : \textsc{Type}_2 \xrightarrow{c} \textsc{Type}_3 \\ \textsc{Type}_1 \leq_{\textsc{Prog}} \textsc{Type}_2 \qquad \textsc{Type}_4 \leq_{\textsc{Prog}} \textsc{Type}_3\end{array}}{\textsc{Prog}, \textsc{Pool} \vdash^{u}_{\mathsf{a}} \mathit{resume}_{\textsc{Prog}}(\textsc{Cont}, \textsc{Value}) : \textsc{Type}_4}$$

$$\textsc{Invoke}\ \frac{\begin{array}{c}\textsc{Prog}, \textsc{Pool} \vdash^{u}_{\mathsf{v}} \textsc{Ref} : \textsc{ObjType} \\ \textsc{Prog}, \textsc{Pool} \vdash^{u\ell}_{\mathsf{v}} \overrightarrow{\textsc{Value}} : \overrightarrow{\textsc{Type}'} \\ \mathit{methods}_{\textsc{Prog}}(\textsc{ObjType})(\textsc{Method}) = \langle\overrightarrow{\textsc{Type}''}, \textsc{Type}_1\rangle \\ \overrightarrow{\textsc{Type}'} \leq_{\textsc{Prog}} \overrightarrow{\textsc{Type}''} \qquad \textsc{Type}_2 \leq_{\textsc{Prog}} \textsc{Type}_1\end{array}}{\textsc{Prog}, \textsc{Pool} \vdash^{u}_{\mathsf{a}} \mathit{invoke}(\textsc{Ref}, \textsc{Pool}(\textsc{Ref}), \textsc{Method}, \overrightarrow{\textsc{Value}}) : \textsc{Type}_2}$$

Figure 13: Lower level parameters to the type system.

Judgments:

$\textsc{Prog} \vdash^u \textsc{State} : \textsc{Type}$	State $\textsc{State}$ has type $\textsc{Type}$.
$\textsc{Prog} \vdash^u \textsc{Pool}$	Object pool $\textsc{Pool}$ is well-formed.
$\textsc{Prog}, \textsc{Pool} \vdash^u \textsc{Stack} : \textsc{Type}_1 \xrightarrow{s} \textsc{Type}_2$	Stack $\textsc{Stack}$ represents computation producing type $\textsc{Type}_2$ if the current method produces type $\textsc{Type}_1$.
$\textsc{Prog}, \textsc{Pool} \vdash^u_{\mathsf{v}} \textsc{Ref} : \textsc{ObjType}$	Reference $\textsc{Ref}$ has type $\textsc{ObjType}$.
$\textsc{Prog}, \textsc{Pool} \vdash^u_{\mathsf{v}} \textsc{Static} : \textsc{Type}$	Static record $\textsc{Static}$ has type $\textsc{Type}$ as a value.
$\textsc{Prog}, \textsc{Pool} \vdash^u_{\mathsf{o}} \textsc{Object} : \textsc{ObjType}$	Object record $\textsc{Object}$ is well-formed in an object of type $\textsc{ObjType}$.
$\textsc{Prog}, \textsc{Pool} \vdash^u_{\mathsf{o}} \textsc{Dynamic} : \textsc{ObjType}$	Dynamic record $\textsc{Dynamic}$ is well-formed in an object of type $\textsc{ObjType}$.
$\textsc{Prog}, \textsc{Pool} \vdash^u_{\mathsf{a}} \textsc{Action} : \textsc{Type}$	Action $\textsc{Action}$ represents computation in a method producing type $\textsc{Type}$.

Type rules:

$$\textsc{State}\ \frac{\begin{array}{c}\textsc{Prog} \vdash^u \textsc{Pool} \qquad \textsc{Prog}, \textsc{Pool} \vdash^u_{\mathsf{v}} \textsc{Ref} : \textsc{ObjType} \\ \textsc{Prog}, \textsc{Pool} \vdash^u_{\mathsf{a}} \textsc{Action} : \textsc{Type}_1 \qquad \textsc{Prog}, \textsc{Pool} \vdash^u \textsc{Stack} : \textsc{Type}_2 \xrightarrow{s} \textsc{Type}_3 \qquad \textsc{Type}_1 \leq_{\textsc{Prog}} \textsc{Type}_2\end{array}}{\textsc{Prog} \vdash^u \langle \textsc{Pool}, \textsc{Stack}, \textsc{Ref}, \textsc{Action} \rangle : \textsc{Type}_3}$$

$$\textsc{Pool}\ \frac{\forall \textsc{Ref} \in \mathit{dom}(\textsc{Pool}).\ \textsc{Prog}, \textsc{Pool} \vdash^u_{\mathsf{v}} \textsc{Ref} : \textsc{ObjType}}{\textsc{Prog} \vdash^u \textsc{Pool}} \qquad \textsc{EmptyStack}\ \frac{}{\textsc{Prog}, \textsc{Pool} \vdash^u \epsilon : \textsc{Type} \xrightarrow{s} \textsc{Type}}$$

$$\textsc{StackFrame}\ \frac{\begin{array}{c}\textsc{Prog}, \textsc{Pool} \vdash^u_{\mathsf{v}} \textsc{Ref} : \textsc{ObjType} \qquad \textsc{Prog}, \textsc{Pool} \vdash^\ell \textsc{Cont} : \textsc{Type}_1 \xrightarrow{c} \textsc{Type}_2 \\ \textsc{Prog}, \textsc{Pool} \vdash^u \textsc{Stack} : \textsc{Type}_3 \xrightarrow{s} \textsc{Type}_4 \qquad \textsc{Type}_2 \leq_{\textsc{Prog}} \textsc{Type}_3\end{array}}{\textsc{Prog}, \textsc{Pool} \vdash^u \langle \textsc{Ref}, \textsc{Cont} \rangle\ \textsc{Stack} : \textsc{Type}_1 \xrightarrow{s} \textsc{Type}_4}$$

$$\textsc{Metatype}\ \frac{\textsc{Prog}, \textsc{Pool} \vdash^\ell_{\mathsf{o}} \textsc{Static} : \textsc{ObjType}}{\textsc{Prog}, \textsc{Pool} \vdash^u_{\mathsf{v}} \textsc{Static} : \mathit{metatype}_{\textsc{Prog}}(\textsc{ObjType})} \qquad \textsc{Reference}\ \frac{\begin{array}{c}\textsc{Pool}(\textsc{Ref}) = \textsc{Object} \\ \textsc{Prog}, \textsc{Pool} \vdash^u_{\mathsf{o}} \textsc{Object} : \textsc{ObjType}\end{array}}{\textsc{Prog}, \textsc{Pool} \vdash^u_{\mathsf{v}} \textsc{Ref} : \textsc{ObjType}}$$

$$\textsc{Object}\ \frac{\textsc{Prog}, \textsc{Pool} \vdash^\ell_{\mathsf{o}} \textsc{Static} : \textsc{ObjType} \qquad \textsc{Prog}, \textsc{Pool} \vdash^u_{\mathsf{o}} \textsc{Dynamic} : \textsc{ObjType}}{\textsc{Prog}, \textsc{Pool} \vdash^u_{\mathsf{o}} \textsc{Static}\ \textsc{Dynamic} : \textsc{ObjType}}$$

$$\textsc{Dynamic}\ \frac{\begin{array}{c}\mathit{dom}(\textsc{Dynamic}) = \mathit{dom}(\mathit{fields}_{\textsc{Prog}}(\textsc{ObjType})) \\ \forall \textsc{Field} \in \mathit{dom}(\textsc{Dynamic}).\ \textsc{Prog}, \textsc{Pool} \vdash^{u\ell}_{\mathsf{v}} \textsc{Value} : \mathit{Type}\ \wedge\ \textsc{Type} \leq_{\textsc{Prog}} \mathit{fields}_{\textsc{Prog}}(\textsc{ObjType})(\textsc{Field})\end{array}}{\textsc{Prog}, \textsc{Pool} \vdash^u_{\mathsf{o}} \textsc{Dynamic} : \textsc{ObjType}}$$

$$\textsc{New}\ \frac{\begin{array}{c}\textsc{Prog}, \textsc{Pool} \vdash^u_{\mathsf{o}} \textsc{Object} : \textsc{ObjType} \\ \textsc{Prog}, \textsc{Pool} \vdash^\ell \textsc{Cont} : \textsc{Type}_1 \xrightarrow{c} \textsc{Type}_2 \\ \textsc{ObjType} \leq_{\textsc{Prog}} \textsc{Type}_1\end{array}}{\textsc{Prog}, \textsc{Pool} \vdash^u_{\mathsf{a}} \textbf{new}\ \textsc{Object} \triangleright \textsc{Cont} : \textsc{Type}_2} \qquad \textsc{Inspect}\ \frac{\begin{array}{c}\textsc{Prog}, \textsc{Pool} \vdash^u_{\mathsf{v}} \textsc{Ref} : \textsc{ObjType} \\ \textsc{Prog}, \textsc{Pool} \vdash^\ell \textsc{Cont} : \textsc{Type}_1 \xrightarrow{c} \textsc{Type}_2 \\ \mathit{metatype}_{\textsc{Prog}}(\textsc{ObjType}) \leq_{\textsc{Prog}} \textsc{Type}_1\end{array}}{\textsc{Prog}, \textsc{Pool} \vdash^u_{\mathsf{a}} \textbf{inspect}\ \textsc{Ref} \triangleright \textsc{Cont} : \textsc{Type}_2}$$

$$\textsc{Get}\ \frac{\begin{array}{c}\textsc{Prog}, \textsc{Pool} \vdash^u_{\mathsf{v}} \textsc{Ref} : \textsc{ObjType} \\ \mathit{fields}_{\textsc{Prog}}(\textsc{ObjType})(\textsc{Field}) = \textsc{Type}_1 \\ \textsc{Prog}, \textsc{Pool} \vdash^\ell \textsc{Cont} : \textsc{Type}_2 \xrightarrow{c} \textsc{Type}_3 \\ \textsc{Type}_1 \leq_{\textsc{Prog}} \textsc{Type}_2\end{array}}{\textsc{Prog}, \textsc{Pool} \vdash^u_{\mathsf{a}} \textbf{get}\ \textsc{Ref}.\textsc{Field} \triangleright \textsc{Cont} : \textsc{Type}_3} \qquad \textsc{Set}\ \frac{\begin{array}{c}\textsc{Prog}, \textsc{Pool} \vdash^u_{\mathsf{v}} \textsc{Ref} : \textsc{ObjType} \\ \textsc{Prog}, \textsc{Pool} \vdash^{u\ell}_{\mathsf{v}} \textsc{Value} : \textsc{Type}_1 \\ \mathit{fields}_{\textsc{Prog}}(\textsc{ObjType})(\textsc{Field}) = \textsc{Type}_2 \\ \textsc{Prog}, \textsc{Pool} \vdash^\ell \textsc{Cont} : \textsc{Type}_3 \xrightarrow{c} \textsc{Type}_4 \\ \textsc{Type}_1 \leq_{\textsc{Prog}} \textsc{Type}_2 \qquad \textsc{Type}_2 \leq_{\textsc{Prog}} \textsc{Type}_3\end{array}}{\textsc{Prog}, \textsc{Pool} \vdash^u_{\mathsf{a}} \textbf{set}\ \textsc{Ref}.\textsc{Field} := \textsc{Value} \triangleright \textsc{Cont} : \textsc{Type}_4}$$

$$\textsc{Call}\ \frac{\begin{array}{c}\textsc{Prog}, \textsc{Pool} \vdash^u_{\mathsf{v}} \textsc{Ref} : \textsc{ObjType} \\ \textsc{Prog}, \textsc{Pool} \vdash^{u\ell}_{\mathsf{v}} \overrightarrow{\textsc{Value}} : \overrightarrow{\textsc{Type}'} \qquad \mathit{methods}_{\textsc{Prog}}(\textsc{ObjType})(\textsc{Method}) = \langle \overrightarrow{\textsc{Type}''}, \textsc{Type}_1 \rangle \\ \textsc{Prog}, \textsc{Pool} \vdash^\ell \textsc{Cont} : \textsc{Type}_2 \xrightarrow{c} \textsc{Type}_3 \qquad \overrightarrow{\textsc{Type}' \leq_{\textsc{Prog}} \textsc{Type}''} \qquad \textsc{Type}_1 \leq_{\textsc{Prog}} \textsc{Type}_2\end{array}}{\textsc{Prog}, \textsc{Pool} \vdash^u_{\mathsf{a}} \textbf{call}\ \textsc{Ref}.\textsc{Method}(\overrightarrow{\textsc{Value}}) \triangleright \textsc{Cont} : \textsc{Type}_3}$$

$$\textsc{Return}\ \frac{\textsc{Prog}, \textsc{Pool} \vdash^{u\ell}_{\mathsf{v}} \textsc{Value} : \textsc{Type}}{\textsc{Prog}, \textsc{Pool} \vdash^u_{\mathsf{a}} \textbf{return}\ \textsc{Value} : \textsc{Type}} \qquad \textsc{Error}\ \frac{}{\textsc{Prog}, \textsc{Pool} \vdash^u_{\mathsf{a}} \textsc{OKError} : \textsc{Type}}$$

Figure 14: Upper level definition of the type system.

LEMMA 2. *If* PROG $\vdash^u$ STATE : TYPE *and* STATE $\mapsto_{\text{PROG}}$ STATE′, *then there exists* TYPE′ *such that* PROG $\vdash^u$ STATE′ : TYPE′ *and* TYPE′ $\leq_{\text{PROG}}$ TYPE.

PROOF. The proof proceeds by cases on the clauses in the definition of $\mapsto$. See Figure 4 for reference. These cases use lemmas 3 and 4, which establish type preservation under modified object pools.

Object Creation. The type of REF′ in the extended pool is the same as the type of OBJECT in POOL, by the REFERENCE rule and lemma 3. The type derivation of the **new** message proves that CONT accepts a supertype of this as input, so by the RESUME rule the action's type is preserved. Lemma 3 proves that the rest of the state's type is preserved.

Object Inspection. From the action's type derivation we know that REF has type OBJTYPE and CONT's input is a supertype of $\mathit{metatype}_{\text{PROG}}$(OBJTYPE). We know PROG, POOL $\vdash^{\ell}_{o}$ STATIC : OBJTYPE from the REFERENCE and OBJECT rules; therefore METATYPE gives us PROG, POOL $\vdash^{u}_{v}$ STATIC : $\mathit{metatype}_{\text{PROG}}$(OBJTYPE). We can apply the RESUME rule to see that the action's type is preserved; the rest of the state is unchanged.

Field Lookup. From the action's type derivation, we know REF has type OBJTYPE and the signature for FIELD in OBJTYPE is a subtype of CONT's input. By the type derivation of REF and the DYNAMIC rule, we know that VALUE is a suitable input for CONT. The RESUME rule proves the action's type is preserved; the rest of the state is unchanged.

Field Mutation. The **set** action's type derivation proves that REF has type OBJTYPE, that VALUE provides an appropriate type for FIELD in OBJTYPE, and that CONT accepts input of that type. By lemma 4, the modified object pool is well-formed and the types of the stack, object reference, and continuation are preserved. The type of the action is preserved due to the RESUME rule.

Method Call. The type derivation of STATE proves that REF has type OBJTYPE, STACK accepts CONT's output, that CONT accepts the output of METHOD in OBJTYPE, and that the method in turn accepts the argument types given to it. The STACKFRAME rule proves the new stack is well-typed and INVOKE proves the new action is appropriate for the stack. The overall state's type is preserved.

Method Return. From the first state's type derivation, we know that STACK accepts CONT's output and that CONT's input type accepts VALUE. By the RESUME rule, the new stack accepts the type of the new action. The STATE rule establishes that the state's type is preserved.

Reference Error. The type rules INSPECT, GET, SET, and CALL all require the receiver reference be typed; the REFERENCE rule dictates that typed references have object pool entries. Therefore the type system rejects reference errors statically. This case is impossible in well-typed programs.

Field Error. The GET and SET rules require that FIELD has a signature in the receiver's object type. By the rules REFERENCE, OBJECT, and DYNAMIC the field must therefore exist in the dynamic record. All field errors are rejected statically. This case is impossible in a well-typed program.

With all the cases proved, the type of STATE is preserved in the transition to STATE′. □

LEMMA 3. *If*

PROG $\vdash^u$ POOL,
REF $\notin$ POOL, *and*
PROG, POOL $\vdash^{u}_{o}$ OBJECT : OBJTYPE,

then PROG $\vdash^u$ POOL[REF $\mapsto$ OBJECT]. *Additionally, all type derivations valid with respect to the first pool remain valid with respect to the new pool.*

PROOF. The proof of well-formedness of the new pool follows from the POOL, REFERENCE, and OBJECT rules. The proof of preserved types relies on the fact that the object pool is only used directly in the REFERENCE and OBJECT rules. Types derived from those rules are preserved; therefore so are all others. □

LEMMA 4. *If*

PROG $\vdash^u$ POOL,
PROG, POOL $\vdash^{u}_{v}$ REF : OBJTYPE,
$\mathit{fields}_{\text{PROG}}$(OBJTYPE)(FIELD) = TYPE$_1$,
PROG, POOL $\vdash^{u\ell}_{v}$ VALUE : TYPE$_2$,
TYPE$_2$ $\leq_{\text{PROG}}$ TYPE$_1$, *and*
POOL(REF) = STATIC DYNAMIC,

then

PROG $\vdash^u$ POOL[REF $\mapsto$ STATIC DYNAMIC[REF $\mapsto$ VALUE]].

Additionally, all type derivations valid with respect to the first pool remain valid with respect to the new pool.

PROOF. The proof is a slight variation on lemma 3. It uses the DYNAMIC and OBJECT rules to prove that the new object record is well-typed and the REFERENCE rule to prove that references to the object are still well-typed. □

5.2 ClassicJava Soundness

Figure 15 presents the lower-level definitions of the ClassicJava type system. To obtain a type soundness result, we must prove that these definitions satisfy the criteria for soundness; together with the upper-level soundness theorem, this establishes a result analogous to Flatt et alii's ClassicJava soundness proof [13].

Types in this formulation of ClassicJava are class or interface names or a **meta**-type, representing the static record for an object of a given type. Object types must be class or interface names. ClassicJava allows null dereferences and failed typecasts in well-typed programs. The subtyping relation is unchanged from the original definition; **meta**-types have no relationship with other types.

The $\mathit{fields}^{\text{CJ}}$ and $\mathit{methods}^{\text{CJ}}$ functions are defined in terms of the $\in^{\text{CJ}}$ relation. The $\mathit{metatype}^{\text{CJ}}$ function defines the static record of CLASS to have type **meta** CLASS.

The type rules are defined in terms of the original ClassicJava judgments for typing programs and expressions. We

$$
\begin{array}{lcll}
\textsc{Type}^{\mathrm{CJ}} & = & \textsc{TName}^{\mathrm{CJ}} \mid \textbf{meta}\ \textsc{TName}^{\mathrm{CJ}} & \text{Types} \\
\textsc{ObjType}^{\mathrm{CJ}} & = & \textsc{TName}^{\mathrm{CJ}} & \text{Object Types} \\
\textsc{OKError}^{\mathrm{CJ}} & = & \textbf{error:null} \mid \textbf{error:typecast} & \text{Runtime Errors} \\
\leq^{\mathrm{CJ}}_{\textsc{Prog}} & & \text{as used in Section 4.2} & \text{Subtyping}
\end{array}
$$

$$
\begin{array}{lcl}
\mathit{fields}^{\mathrm{CJ}}_{\textsc{Prog}}(\textsc{TName}) & = & [\langle \textsc{Class}, \textsc{Field}\rangle \mapsto \textsc{Type} \mid \langle \textsc{Class}, \textsc{Field}, \textsc{Type}\rangle \in^{\mathrm{CJ}}_{\textsc{Prog}} \textsc{TName}] \\
\mathit{methods}^{\mathrm{CJ}}_{\textsc{Prog}}(\textsc{TName}) & = & [\langle \textsc{Class}, \textsc{Method}\rangle \mapsto \langle \overrightarrow{\textsc{Type}}, \textsc{Type}\rangle \mid \textsc{TName} \leq^{\mathrm{CJ}}_{\textsc{Prog}} \textsc{Class} \\
 & & \quad \text{and } \langle \textsc{Method}, \overrightarrow{\textsc{Type}} \longrightarrow \textsc{Type}, \overrightarrow{\textsc{Var}}, \textsc{Expr}\rangle \in^{\mathrm{CJ}}_{\textsc{Prog}} \textsc{Class}] \\
 & & [\textsc{Method} \mapsto \langle \overrightarrow{\textsc{Type}}, \textsc{Type}\rangle \mid \langle \textsc{Method}, \overrightarrow{\textsc{Type}} \longrightarrow \textsc{Type}, \overrightarrow{\textsc{Var}}, \textsc{Expr}\rangle \in^{\mathrm{CJ}}_{\textsc{Prog}} \textsc{TName}] \\
\mathit{metatype}^{\mathrm{CJ}}_{\textsc{Prog}}(\textsc{TName}) & = & \textbf{meta}\ \textsc{TName}
\end{array}
$$

$$
\textsc{Prog}\ \frac{\vdash^{\mathrm{CJ}}_{\mathsf{p}} \textsc{Prog} : \textsc{ObjType}}{\vdash^{\ell}\ \textsc{Prog} : \textsc{ObjType}}
\qquad
\textsc{Cont}\ \frac{\textsc{Var} \notin \textsc{Cont} \qquad \textsc{Prog}, \mathit{getenv}^{\mathrm{CJ}}(\textsc{Pool})[\textsc{Var} \mapsto \textsc{Type}_1] \vdash^{\mathrm{CJ}}_{\underline{\mathsf{e}}} \textsc{Cont}[\textsc{Var}] : \textsc{Type}_2}{\textsc{Prog}, \textsc{Pool} \vdash^{\ell}\ \textsc{Cont} : \textsc{Type}_1 \xrightarrow{c} \textsc{Type}_2}
$$

$$
\textsc{Class}\ \frac{}{\textsc{Prog}, \textsc{Pool} \vdash^{\ell}_{\mathsf{o}} \textsc{Class} : \textsc{Class}}
\qquad
\textsc{Null}\ \frac{}{\textsc{Prog}, \textsc{Pool} \vdash^{\ell}_{\mathsf{v}} \textbf{null} : \textsc{ObjType}}
$$

Original ClassicJava judgments:

$\textsc{Env}$	ClassicJava environment.
$\vdash^{\mathrm{CJ}}_{\mathsf{p}} \textsc{Prog} : \textsc{Type}$	ClassicJava program $\textsc{Prog}$ has type $\textsc{Type}$.
$\textsc{Prog}, \textsc{Env} \vdash^{\mathrm{CJ}}_{\underline{\mathsf{e}}} \textsc{Expr} : \textsc{Type}$	ClassicJava expression $\textsc{Expr}$ has type $\textsc{Type}$.

where

$$
\textsc{ViewSub}^{\mathrm{CJ}}\ \frac{\begin{array}{c}\textsc{Prog}, \textsc{Env} \vdash^{\mathrm{CJ}}_{\underline{\mathsf{e}}} \textsc{Expr} : \textbf{meta}\ \textsc{TName}' \\ \textsc{Prog}, \textsc{Env} \vdash^{\mathrm{CJ}}_{\underline{\mathsf{e}}} \textsc{Value} : \textsc{TName}'\end{array}}{\textsc{Prog}, \textsc{Env} \vdash^{\mathrm{CJ}}_{\underline{\mathsf{e}}} \textbf{view}\ \textsc{Expr} \leq \textsc{TName}\ \textsc{Value} : \textsc{TName}}
$$

and $\mathit{getenv}^{\mathrm{CJ}} : \textbf{Pool} \longrightarrow \textbf{Env}$ **is:**

$\mathit{getenv}^{\mathrm{CJ}}(\textsc{Pool}) = [\textsc{Ref} \mapsto \textsc{Class} \mid \textsc{Pool}(\textsc{Ref}) = \textsc{Class Dynamic}]$

Figure 15: Lower level parameters for the ClassicJava type system.

extend the ClassicJava rules to cover the elaborated form of **view** and define a function $getenv^{\text{CJ}}$ to construct a ClassicJava environment from the object pool.

THEOREM 2. *The sequence trace representation of ClassicJava is type-sound.*

PROOF. The goal of our proof is to validate that all the lower-level type relations and functions satisfy the three constraints from the preceding section. We dedicate one lemma to each constraint. Specifically, lemma 10 concerns $init^{\text{CJ}}$, lemma 12 is about $invoke^{\text{CJ}}$, and lemma 11 establishes $resume^{\text{CJ}}$. □

Before we can prove anything meaningful about ClassicJava evaluation as presented in Figure 10, we must prove that the primitive operations of substitution for a variable, plugging expressions into evaluation contexts, $\mapsto^{\text{CJ}}$, $eval^{\text{CJ}}$, and $evalmethod^{\text{CJ}}$ all preserve the types of their inputs.

LEMMA 5. *If*

$$\begin{array}{l}\text{PROG}, \text{ENV}[\text{VAR} \mapsto \text{TYPE}_0] \vdash^{\text{CJ}}_{\underline{e}} \text{EXPR} : \text{TYPE}_1, \\ \text{VAR} \notin dom(\text{ENV}), \text{ and} \\ \text{PROG}, \text{ENV} \vdash^{\text{CJ}}_{\underline{s}} \text{VALUE} : \text{TYPE}_0,\end{array}$$

then $\text{PROG}, \text{ENV} \vdash^{\text{CJ}}_{\underline{e}} \text{EXPR}[\text{VALUE}/\text{VAR}] : \text{TYPE}_2$ *such that* $\text{TYPE}_2 \leq_{\text{PROG}} \text{TYPE}_1$.

PROOF. This proof proceeds by induction on the structore of the type derivation for EXPR. □

LEMMA 6. *If*

$$\begin{array}{l}\text{PROG}, \text{POOL} \vdash^{\ell} \text{CONT} : \text{TYPE}_2 \xrightarrow{c} \text{TYPE}_3, \\ \text{PROG}, getenv^{\text{CJ}}(\text{POOL}) \vdash^{\text{CJ}}_{\underline{e}} \text{EXPR} : \text{TYPE}_1, \text{ and} \\ \text{TYPE}_1 \leq_{\text{PROG}} \text{TYPE}_2,\end{array}$$

then $\text{PROG}, getenv^{\text{CJ}}(\text{POOL}) \vdash^{\text{CJ}}_{\underline{e}} \text{CONT}[\text{EXPR}] : \text{TYPE}_4$ *such that* $\text{TYPE}_4 \leq_{\text{PROG}} \text{TYPE}_3$.

PROOF. This is a corollary of lemma 5; because the CONT rule types continuations by plugging in a variable, plugging in an arbitrary expression to the same type derivation is a simple matter of substitution. It is worth noting that none of the evaluation contexts capture free variables, so that is not an issue. □

LEMMA 7. *If*

$$\begin{array}{l}\text{PROG}, \text{ENV} \vdash^{\text{CJ}}_{\underline{e}} \text{EXPR} : \text{TYPE} \text{ and} \\ \text{EXPR} \mapsto^{\text{CJ}}_{\text{PROG}} \text{EXPR}',\end{array}$$

then $\text{PROG}, \text{ENV} \vdash^{\text{CJ}}_{\underline{e}} \text{EXPR}' : \text{TYPE}'$ *such that* $\text{TYPE}' \leq_{\text{PROG}} \text{TYPE}$.

PROOF. There are two cases to examine:

Let Binding. This case follows directly from lemma 5.

Successful Typecast. By the type derivation of an elaborated typecast, VALUE has type TNAME', and the expression is replaced with VALUE only when

$$\text{TNAME}' \leq^{\text{CJ}}_{\text{PROG}} \text{TNAME}.$$

Therefore the type TNAME of the expression is preserved.

□

LEMMA 8. *If*

$$\text{PROG}, getenv^{\text{CJ}}(\text{POOL}) \vdash^{\text{CJ}}_{\underline{e}} \text{EXPR} : \text{TYPE}$$

then

$$\text{PROG}, \text{POOL} \vdash^{u}_{a} eval^{\text{CJ}}_{\text{PROG}}(\text{EXPR}) : \text{TYPE}'$$

such that $\text{TYPE}' \leq_{\text{PROG}} \text{TYPE}$.

PROOF. The $eval^{\text{CJ}}$ function first reduces its input before generating a message. After reduction, $eval^{\text{CJ}}$ creates an action from the resulting expression. We first argue that the creation of these actions is a total function, then that the function preserves its input's type.

The step function $\mapsto^{\text{CJ}}$ strictly reduces the total number of **let** and **view** subexpressions at each step, so there cannot be an infinite sequence of reductions. The clauses of the step relation and the final action generation are exhaustive, so every reduced expression will result in an action. Therefore $eval^{\text{CJ}}$ is a total function.

The proof that $eval^{\text{CJ}}$ preserves types proceeds by cases on the fully reduced expression. In each case, the type derivation for the resulting action can be assembled from pieces of the type derivation of the expression; the pieces of the expression are shuffled around but nothing new is generated (there is no substitution or replication). □

LEMMA 9. *If*

$$\begin{array}{l}\text{PROG}, \text{POOL} \vdash^{\ell}_{o} \text{STATIC} : \text{TNAME}, \\ \text{PROG}, \text{POOL} \vdash^{u}_{v} \text{REF} : \text{TNAME}, \\ \text{PROG}, \text{POOL} \vdash^{u\ell}_{v} \overrightarrow{\text{VALUE}} : \overrightarrow{\text{TYPE}'}, \\ methods^{\text{CJ}}(\text{TNAME})(\text{METHOD}) = \langle \overrightarrow{\text{TYPE}''}, \text{TYPE}_1 \rangle, \text{ and} \\ \overrightarrow{\text{TYPE}' \leq_{\text{PROG}} \text{TYPE}''},\end{array}$$

then $evalmethod^{\text{CJ}}_{\text{PROG}}(\text{REF}, \text{STATIC}, \text{METHOD}, \overrightarrow{\text{VALUE}})$ *has the type* TYPE_2 *such that* $\text{TYPE}_2 \leq_{\text{PROG}} \text{TYPE}_1$.

PROOF. This lemma relies on the type derivation of the program's class definitions. Specifically, it requires that if:

$$\begin{array}{l}\langle \text{METHOD}, \overrightarrow{\text{TYPE}'} \longrightarrow \text{TYPE}, \overrightarrow{\text{VAR}}, \text{EXPR} \rangle \in^{\text{CJ}}_{\text{PROG}} \text{CLASS}, \\ \text{PROG}, \text{ENV} \vdash^{\text{CJ}}_{\underline{s}} \overrightarrow{\text{VALUE}} : \overrightarrow{\text{TYPE}''}, \\ \text{PROG}, \text{ENV} \vdash^{\text{CJ}}_{\underline{s}} \text{REF} : \text{CLASS}', \\ \overrightarrow{\text{TYPE}'' \leq^{\text{CJ}}_{\text{PROG}} \text{TYPE}'} \text{ and } \text{CLASS}' \leq^{\text{CJ}}_{\text{PROG}} \text{CLASS},\end{array}$$

then $\text{PROG}, \text{ENV} \vdash^{\text{CJ}}_{\underline{e}} \text{EXPR}[\text{REF}/\textbf{this}][\overrightarrow{\text{VALUE}/\text{VAR}}] : \text{TYPE}$. Put simply, when a method body's parameters of appropriate types are filled in, the method must return the right type.

When these conditions hold, $evalmethod^{\text{CJ}}$ passes the substituted method body to $eval^{\text{CJ}}$. By lemma 8, the expression's type is preserved. In a well-typed program, these conditions will always hold; $evalmethod^{\text{CJ}}$ preserves types and method errors are rejected statically. □

LEMMA 10. *The* INIT *rule holds for ClassicJava.*

PROOF. The type of the program

$$\overrightarrow{\text{DEF}}\ \textbf{new}\ \text{CLASS.METHOD}()$$

is the result type of METHOD in CLASS. By inspection, we can determine that the initial state's type is the result type of $methods^{\text{CJ}}_{\text{PROG}}(\text{CLASS})(\text{METHOD})$. By the definition of $methods^{\text{CJ}}$, these are the same types; therefore $init^{\text{CJ}}$ preserves the program's type. □

LEMMA 11. *The* RESUME *rule holds for ClassicJava.*

PROOF. This is a straightforward application of lemmas 6 and 8. □

LEMMA 12. *The* INVOKE *rule holds for ClassicJava.*

PROOF. There are two clauses in $invoke^{CJ}$'s definition; one for simple method names and one for method names specifying a class for dispatch. Both invoke $evalmethod^{CJ}$ on one of REF's (super)classes (as constrained by $methods^{CJ}$); the former clause merely defaults to the most specific class. It suffices to consider the second case.

The premises of the INVOKE rule (plus an application of the REFERENCE and OBJECT rules to determine STATIC's type) allow us to apply lemma 9. □

6. RELATED WORK

Our work has two inspirational sources. Calculi for communicating processes often model just those actions that relate to process creation, communication, and so on. 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 for the lower level and, with the specification of a lower level, models entire languages. In the exploration of process calculi, traces play a natural role, though research there often concerns observational equivalence or problems of parallel, concurrent, and even distributed computations. Starting from this insight, Graunke, the second author, and others [15, 16, 24] 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 prior research to an arbitrarily large and growing pool of objects with a rich set of actions and a wide interface to the underlying computational language.

Others tools for inspecting and debugging program traces exist, tackling the problem from many different perspectives. Lewis [25] 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 than presenting an intuitive view of computation.

The work of Walker, Murphy, Steinbok, and Robillard [34] develops a method for constructing abstract interpretations of a trace interactively. They allow a user to write rules grouping primitive program elements into abstract elements. The program trace is reinterpreted with respect to the abstract elements; they provide visualization and summarization tools for the resulting abstract trace.

Richner and Ducasse [32] demonstrate a tool for recovering class collaborations from the execution of object-oriented programs. The process is iterative and interactive; a programmer writes and refines queries to find the meaningful collaborations in a program trace. Their tool presents a browser for collaborations and interactions discovered in the program.

Ducasse, Gîrba, and Wuyts [10] create regression tests by examining execution traces of legacy software. Queries are written as logic programming predicates; queries that the system satisfies can be kept as tests to apply to updated versions of the software. This allows the creation of test suites without the need for documentation, understanding the code, or bringing the program into a known state for each test.

Even though our work does not attempt to assign semantics to UML's sequence charts, many pieces of research in this direction exist and share some similarities with our own work. We therefore describe the most relevant pieces of work in this direction here. Lund and Stølen [27] implement an operational semantics for UML 2.0 sequence diagrams with an execution system and a projection system. The execution system handles the representation of the communication medium and the projection system handles choices among lifelines when executing events. They also formalize a type system that separates valid and invalid traces and employs strategies for guiding execution.

Xia and Kane [37] develop a semantics for UML class and sequence diagrams based on attribute grammars. They formalize static attributes that express the objects and operations referenced from within an operation and dynamic attributes that express the relative ordering of execution of operations. They demonstrate how to prove theorems about class and sequence diagrams within their semantics.

Whittle [35] implements a method for generating statecharts from collections of sequence diagrams. Ambiguities and inconsistencies among sequence diagrams are resolved with Object Constraint Language annotations. He lacks a formal semantics and says that his approach is pragmatic.

Okazaki, Aoki, and Katayama [29] also use state machines to combine sequence diagrams. They formulate a new language for state machines called Concurrent Regular Expressions. They present the translation of sequence diagrams into this language and the method for combining the resulting machines. They also present two examples of checking consistency properties of the diagrams using the new state machines.

Li, Liu, and He [26] develop a formal static and dynamic semantics for UML sequence diagrams. Their static semantics relates a sequence diagram to a hierarchical tree structure of method calls and checks consistency against class diagrams. Their dynamic semantics relates sequence diagrams to consistency predicates on the execution of single operations. Much like our sequence trace semantics, their dynamic semantics presents executions as the composition of message passing and small connecting programs within each object.

Nantajeewarawat and Sombatsrisomboon [28] define a new model theoretic framework for class and sequence diagrams. They provide conditions for valid interpretations and a set of inference rules for reasoning about diagrams. They demonstrate the use of these rules to infer a class diagram from a sequence diagram.

Hausmann, Heckel, and Sauer [19] introduce a technique dubbed dynamic meta modeling and demonstrate it with a dynamic semantics for executing sequence diagrams directly. Roughly speaking dynamic meta modeling is a collection of graphical rewriting rules on collaboration diagrams.

Cho, Kim, Cha, and Bae [6] present a semantics for sequence diagrams in a new temporal logic called HDTL. They provide a method for computing HDTL formulae from a given sequence diagram and a method for validating the formulae. The validation produces instances of the sequence diagram from traces of many independently operating ob-

jects.

Bernardi, Donatelli, and Merseguer [3] provide a translation from statecharts and sequence diagrams to Labeled Generalized Stochastic Petri Nets. The partial order on messages expressed by a sequence diagram is preserved, as are the synchronous/asynchronous and delayed/instantaneous properties of each action. The resulting Petri nets are used to compute logical properties of the diagrams as well as predict performance results.

7. CONCLUSIONS AND FUTURE WORK

The 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 mathematical entities: an unspecified set plus a number of functions. At this point the framework can easily handle models such as ClassicJava, as demonstrated in section 4.2, and language such as PLT Scheme, as demonstrated in section 2.

In the near future, we expect to elaborate the object-oriented level in several directions. Most importantly, the object-oriented level currently assumes a functional creation mechanism for objects. While this works well for the PLT Scheme tracer, it cannot cope with Java's complex object instantiation. Conversely, the framework does not support a destroy action. To understand its addition fully, we intend to explore how an unsafe object-oriented language such as C++ can be modeled with a revised and extended framework. Furthermore, object-oriented languages often provide a modicum of reflection, which is a way of turning information about class hierarchies into objects. Last but not least we are interested in accommodating language constructs such as mixins [13] and traits [33], which require some amount of class construction at run-time.

Concerning the implementation of the framework, we intend to scale it from the kernel of PLT Scheme's class language to the full language and ProfessorJ [17], which is an implementation of Java 1.1 in PLT Scheme. Doing so will confirm that the framework is the basis for a generally useful object-oriented tracing tool. Naturally it will require some amount of human interface work to make sure the details of an execution trace are presented on demand and properly. Once finished, the implementation will help us test the usefulness of trace-based debugging with a reasonably large user base.

8. REFERENCES

[1] M. Abadi and L. Cardelli. *A Theory of Objects*. Springer-Verlag New York, Inc., Secaucus, NJ, USA, 1996.

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

[3] S. Bernardi, S. Donatelli, and J. Merseguer. From UML sequence diagrams and statecharts to analysable petri net models. In *WOSP '02: Proceedings of the 3rd international workshop on Software and performance*, pages 35–45, New York, NY, USA, 2002. ACM Press.

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

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

[6] S. M. Cho, H. H. Kim, S. D. Cha, and D. H. Bae. A semantics of sequence diagrams. *Inf. Process. Lett.*, 84(3):125–130, 2022.

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

[8] W. R. Cook and J. Palsberg. A denotational semantics of inheritance and its correctness. In *OOPSLA*, pages 433–443, 1989.

[9] S. Drossopoulou and S. Eisenbach. Java is type safe - probably. In M. Aksit and S. Matsuoka, editors, *ECOOP*, volume 1241 of *Lecture Notes in Computer Science*, pages 389–418. Springer, 1997.

[10] S. Ducasse, T. Gîrba, and R. Wuyts. Object-oriented legacy system trace-based logic testing. In *Proceedings 10th European Conference on Software Maintenance and Reengineering (CSMR 2006)*. IEEE Computer Society Press, 2006.

[11] R. B. Findler, C. Flanagan, M. Flatt, S. Krishnamurthi, and M. Felleisen. DrScheme: A pedagogic programming environment for Scheme. In H. Glaser, P. Hartel, and H. Kuchen, editors, *Programming Languages: Implementations, Logics, and Programs*, volume 1292 of *LNCS*, pages 369–388, Southampton, UK, September 1997. Springer.

[12] M. Flatt. PLT MzScheme: Language manual. Technical Report PLT-1, PLT, Inc., 2005. http://www.plt-scheme.org/software/.

[13] M. Flatt, S. Krishnamurthi, and M. Felleisen. Classes and mixins. In *Proc. 25th ACM Symposium on Principles of Programming Languages*, pages 171–183, 1998.

[14] M. Fowler and K. Scott. *UML Distilled: Applying the Standard Object Modeling Language*. Addison-Wesley Longman Ltd., Essex, UK, 1997.

[15] P. Graunke, R. Findler, S. Krishnamurthi, and M. Felleisen. Modeling web interactions. In *European Symposium on Programming*, pages 238–252, April 2003.

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

[17] K. E. Gray and M. Flatt. ProfessorJ: a gradual introduction to Java through language levels. In R. Crocker and G. L. S. Jr., editors, *OOPSLA Companion*, pages 170–177. ACM, 2003.

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

[19] J. H. Hausmann, R. Heckel, and S. Sauer. Towards dynamic meta modeling of UML extensions: An extensible semantics for UML sequence diagrams. In *HCC*, pages 80–87. IEEE Computer Society, 2001.

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

[21] C. Hewitt, P. Bishop, and R. Steiger. A universal modular actor formalism for artificial intelligence. In *Int. J. Conference on Artifical Intelligence*, pages

235–245, 1973.

[22] S. N. Kamin. Inheritance in SMALLTALK-80: a denotational definition. In *Proc. 15th ACM Symposium on Principles of Programming Languages*, pages 80–87, 1988.

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

[24] S. Krishnamurthi, R. B. Findler, P. Graunke, and M. Felleisen. Modeling web interactions and errors. In *Interactive Computation: The New Paradigm*. Springer Verlag, 2006 – to appear.

[25] B. Lewis. Debugging backwards in time. `http://www.lambdacs.com/debugger/AADEBUG_Mar_03.pdf`, September 2003.

[26] X. Li, Z. Liu, and J. He. A formal semantics of UML sequence diagram. In *Australian Software Engineering Conference*, pages 168–177. IEEE Computer Society, 2004.

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

[28] E. Nantajeewarawat and R. Sombatsrisomboon. On the semantics of unified modeling language diagrams using Z notation. *Int. J. Intell. Syst.*, 19(1-2):79–88, 2004.

[29] M. Okazaki, T. Aoki, and T. Katayama. Formalizing sequence diagrams and state machines using concurrent regular expression. In *Proceedings of 2nd International Workshop on Scenarios and State Machines: Models, Algorithms, and Tools SCESM'03*, pages 74–79, 2003.

[30] U. S. Reddy. Object as closures: abstract semantics of object oriented languages. In *Proc. ACM Conference on Lisp and Functional Programming*, pages 289–297, 1988.

[31] T. Richner. *Recovering Behavioral Design Views: a Query-Based Approach*. PhD thesis, University of Berne, May 2002.

[32] T. Richner and S. Ducasse. Using dynamic information for the iterative recovery of collaborations and roles. In *Proceedings of ICSM '2002 (International Conference on Software Maintenance)*, Oct. 2002.

[33] N. Schärli, S. Ducasse, O. Nierstrasz, and A. P. Black. Traits: Composable units of behaviour. In *ECOOP 2003—Object-Oriented Programming: 17th European Conference*, volume 2743, pages 248–274. Springer-Verlag, 2003.

[34] R. J. Walker, G. C. Murphy, J. Steinbok, and M. P. Robillard. Efficient mapping of software system traces to architectural views. In S. A. MacKay and J. H. Johnson, editors, *CASCON*, page 12. IBM, 2000.

[35] J. Whittle. On the relationship between UML sequence diagrams and state diagrams. In *Proc. of OOPSLA 2000 Workshop: Scenario-based round-trip engineering*, pages 1–6. Tampere University of Technology, Software Systems Laboratory, October 2000. Report 20.

[36] A. K. Wright and M. Felleisen. A syntactic approach to type soundness. *Inf. Comput.*, 115(1):38–94, 1994.

[37] F. Xia and G. S. Kane. Defining the semantics of UML class and sequence diagrams for ensuring the consistency and executability of OO software specification. `http://cc.ee.ntu.edu.tw/~atva03/papers/16.pdf`, December 2003.