This article has been accepted for publication in a future issue of this journal, but has not been fully edited. Content may change prior to final publication.

Specifying Dynamic Analyses by Extending Language Semantics

Adrian Lienhard, Tudor Gîrba, and Oscar Nierstrasz

Abstract—Dynamic analysis is increasingly attracting attention for debugging, profiling, and program comprehension. Ten to twenty years ago, many dynamic analyses investigated only simple method execution traces. Today, in contrast, many sophisticated dynamic analyses exist, for instance for detecting memory leaks, analyzing ownership properties, measuring garbage collector performance, or supporting debugging tasks. These analyses depend on complex program instrumentations and analysis models, making it challenging to understand, compare, and reproduce the proposed approaches. While formal specifications and proofs are common in the field of static analysis, most dynamic analyses are specified using informal, textual descriptions. In this article we propose a formal framework using operational semantics that allows researchers to precisely specify their dynamic analysis. Our goal is to provide an accessible and reusable basis on which researchers that may not be familiar with rigorous specifications of dynamic analyses can build. By extending the provided semantics, one can concisely specify how runtime events are captured and how this data is transformed to populate the analysis model. Furthermore, our approach provides the foundations to reason about properties of a dynamic analysis.

Index Terms—Dynamic Analysis, Formal Definitions and Theory, Tracing, Debugging

1 Introduction

Dynamic analysis has gained increasing attention during the last decade. Historically, dynamic analysis was used for debugging, testing, and profiling. As programs have become larger and more complex, dynamic analysis has come to play an important role in the research field of program comprehension [1]. Nevertheless, most models and instrumentation techniques are described only informally. This poses no problem as long as the analysis is based on simple call traces. Informal descriptions of more complex approaches, however, may be difficult to understand, let alone to reproduce.

We experienced this problem while describing our previous work on Object Flow Analysis [2], [3]. We illustrated our meta-model by means of a UML class diagram and the properties derived from it in OCL, and we described the overall approach informally. However, our description lacked a precise explanation of how the dynamic data is captured at runtime and how the object flow model is built in the first place. This is a nontrivial part as it involves tracking of how object references are transferred at runtime. Without these details the results are hardly reproducible by an independent researcher.

We can find similar limitations in many other publications. For instance, Quante *et al.* describe only superficially how in their Object Process Graph approach data are gathered from C programs, leaving room for different interpretations [4]. The following two sentences offer an example. "By object, we mean a local or global variable or a variable allocated on the heap at runtime. Hence, we consider a trace a sequence of operations applied to an object." [5]. Talking about variables allocated on the heap is rather ambiguous (*e.g.*, does this include primitive type values?). On the other hand it is not clear which operations on those objects are meant, as operations could mean to change the state of an array or record, passing pointers to a value, or changing the value of a variable.

To overcome the limited precision of many of today's dynamic analysis publications, a more rigorous approach is required that allows authors to specify how they trace a running system and how their analysis models are populated from the gathered data. There exist approaches to specify dynamic analyses formally, such as the work by Flanagan *et al.* [6], [7], [8] in the area of runtime verification. Nevertheless, using formal specifications has not yet become a common practice in dynamic analysis publications – in particular not in the fields of program comprehension and reverse engineering. Hence, the goal of this article is to provide a framework that is:

- *Reusable.* It should not be necessary for each publication to come up with a new kind of formalization, which would require an extensive explanation and hence take precious space. The formalizations should be concise and built from a common ground.
- *Accessible.* The formalization should be simple to devise and understand, taking into account that authors and readers of dynamic analysis publications may not be familiar with formal methods.

To meet these goals we propose a formal framework for specifying dynamic analyses of class-based object-oriented programs based on a small object-oriented imperative language with operational semantics, influenced

• *A. Lienhard, T. Gîrba and O. Nierstrasz are with the Software Composition Group, University of Bern, Switzerland, http://scg.unibe.ch/*

Digital Object Indentifier 10.1109/TSE.2011.38

by established approaches [9], [10], [11]. While existing approaches typically use a separate layer of instrumentation rules [6], we propose that for specific analyses the core reduction rules provided by our framework are directly extended and modified. This provides the freedom for novel analyses to specify unanticipated run-time events and models, and it makes the resulting formalization more concise and hence simpler to understand.

We intelligibly introduce the formal semantics and provide examples for real, non-trivial dynamic analyses to illustrate how the framework can be used in practice. With our formal approach one can direct attention to issues that might otherwise be overlooked. Our framework: (i) allows analyses to be precisely specified, (ii) makes design dimensions and corner cases explicit, and (iii) serves as a common reference for comparing approaches.

Even though standard operational semantics have been known for years and approaches exist that use operational semantics to specify dynamic analyses, to the best of our knowledge there is no common practice nor a reusable framework, which is accessible to authors and readers of dynamic analysis publications. Hence, we envision this article to be able to serve as a common reference for future dynamic analysis specifications.

The remainder of this article is structured as follows. We present a categorization of data that dynamic analyses gather in Section 2. This categorization serves as a basis to decide what to support in the proposed base language. We define the syntax and operational semantics of this language in Section 3, and exemplify it in Section 4. In the second part we evaluate our approach by formalizing two significantly distinct, non-trivial dynamic analyses (Section 5, Section 6). We discuss related work in Section 7 and conclude in Section 8.

2 Categorizing Dynamic Analysis Data

To decide which language features are required for a generally applicable, yet minimal formal language, we systematically categorize the data used by recent dynamic analyses of class-based object-oriented programs.

We reviewed the literature to select illustrative examples that cover a large spectrum of the different kinds of data points generally analyzed. The most common technique used to gather runtime data is to instrument the bytecode by adding probes that emit events. Also, more recently, the Java VM Tool Interface and DTrace [12] with support for object-oriented languages provide means to efficiently capture runtime data. Both kinds of techniques yield a trace of *events*, which is analyzed either at runtime or after the execution has terminated.

2.1 Runtime events in object-oriented programs

We categorize the types of events into four categories as follows (see also Figure 1):

		De Pauw	Pothier	Rayside	Pheng
Control flow	Method entry	✔	✔	✔	
	Method return	✔	✔		
	Message send		✔		✔
	Exception raise		✔		✔
	Exception catch				✔
Data flow	Field read			✔	✔
	Field write		✔	✔	✔
	Array read			✔	✔
	Array write		✔	✔	✔
	Local variable read				✔
	Local variable write		✔		✔
	Method argument read				✔
	Method return value		✔		✔
	This reference				✔
	Literal reference				✔
Heap	Class instantiation	✔		✔	✔
	Array creation	✔		✔	✔
	Cloning				✔
	Object finalization	✔		✔	
VM	GC cycle begin/end			✔	
	Thread start/suspend/...				
	Lock (de-)allocation				
	Calls to reflection API				
	System calls: I/O, ...				

Figure 1. Types of dynamic analysis events (supported events highlighted in gray).

(A) **Control flow.** Most approaches trace method executions either by capturing method entry/exit or by capturing message sends. Exceptions belong also to this category as they influence the flow of control.

(B) **Data flow.** This category is composed of events of operations that transfer data (object references or primitive type values). This includes reading and writing to fields, arrays, and local variables. We also include in this category accessing of method arguments, returning of values from method calls, access to the pseudo variable this (the target of the current method call), and accessing of literal values (that is, each time the execution pushes a literal, such as a boolean or string literal, on the stack).

(C) **Heap.** This category includes the events of creating new objects and arrays on the heap and the event triggered when an object or array is garbage collected (finalization). As a variant of instantiation, cloning refers to making a copy of an existing object.

(D) **VM.** This category lists events that are triggered through primitive calls or that are not directly related to any code execution: garbage collection, thread start/stop, lock (de-)allocation, reflection API calls (both structural and behavioral reflection, including dynamic code loading), and system calls.

The columns in Figure 1 show four selected dynamic analyses [13], [14], [15], [16]. For each event, a tick

indicates that the analysis uses the given event.

The first analysis we chose is "Modeling Object-Oriented Program Execution" proposed by De Pauw *et al.* [13], which represents a traditional dynamic analysis that traces only method execution and object creation and finalization. The second column shows the back-in-time debugger "Trace-Oriented Debugger" by Pothier *et al.* [14]. This debugger records the execution history to reconstruct past call stacks and object states. It captures control flow events and data flow events that produce side effects. The memory leak profiler proposed by Rayside *et al.* captures field and array reads and writes, object creation and finalization, and GC events [15]. The dynamic analysis proposed by Pheng *et al.* [16] captures a complete trace of instructions to reconstruct an internal model of the heap. Their goal is to help developers understand how programs use and manipulate heap-based data structures and the effect of garbage collection.

2.2 Selecting events for the base language

The categorization of the 23 runtime events shown in Figure 1 serves as the basis on which to decide what events should be supported. We have selected 10 events that capture the core of most OO languages and their dynamic analyses: (i) method entry, method return, and message send; (ii) field read and field write; (iii) method argument read, method return value, this reference, and literal reference; (iv) class instantiation. These events are essential because we have to represent the heap, and we need to capture message sending including the passing of arguments and return values.

The remaining events can be divided into the following three categories: (i) events that are closely related to the above selected events and hence do not necessarily need special treatment, (ii) events that are not often used but can be modeled by extending the base language without much effort, and (iii) events that are complex to model.

In the first category fall the three events array read, array write, and array creation. Without a significant loss of generality we can omit these events because they are very similar to the events field read, field write, and class instantiation. Arrays could be represented very similarly to objects on the heap, but with indexed slots instead of named slots. However, if an analysis requires one to explicitly specify the tracing of array accesses, the rules of our base language can be extended with modest effort.

The same rationale holds for not modeling local variables. The event *local variable read* is very similar to method argument read. An extension of the base language would again be possible, although a bit more complicated (method contexts need to be extended to hold local variable bindings). We omit the clone event because only very few analyses use it and it would not be difficult to model this event. On the other hand, we omit exceptions, events related to garbage collection, concurrency , and system calls. These events would significantly increase the complexity and hence we decided to not support them directly for the sake of simplicity.

We believe that the set of events selected is adequate for covering the bulk of the state of the art, while serving as a good starting point for future extensions. One of the most important extensions is to model concurrency. A recent study shows that 12 out of 114 articles concerning program understanding through dynamic analysis monitor multithreaded applications [1]. Although this is still a small minority, the analysis of concurrent programs is becoming increasingly important.

3 The Base Object Language L

We now specify the operational semantics of L, a minimal object-oriented language that expresses the events selected in Section 2. Structural operational semantics, pioneered by Plotkin [17], describes in a mathematically rigorous way how a program is interpreted as sequences of computational steps. We do not define the static semantics that describes well-formed programs. From a program P we only assume sets of identifiers for class names *Class*, field names *Field*, and method names *Method*, and we use variables $c \in Class$, $f \in Field$, and $m \in Method$. Moreover, we assume that $MethodLookup(m, c)$ yields a tuple (e, c'). The element e is the expression constituting the body of the method that is returned from a lookup of m starting in class c. The element c' is the class in which this method is implemented. $MainBody()$ returns the body of the main method (from which the program is started). The function $FieldsOf(c)$ returns the set of field identifiers of class c.

3.1 Syntax

In Figure 2 we define an abstract syntax of L with source expressions *Expr*. To simplify our presentation, but without loss of generality, we define methods to have exactly one argument, referred to by the variable x. A runtime expression *RExpr* is a source expression, an address, or a call with its stack frame σ. Runtime expressions do not have primitive type values, such as null. All values in runtime expressions are addresses referencing objects on the heap. The literal null is represented as the single instance of the class *UndefinedObject* (other literal values would be represented in the same way).

3.2 Dynamic aspects

The heap maps addresses ι to objects o. Objects are defined as tuples carrying their class name and a finite mapping from field names to addresses. We use the notation $H(\iota)$ to denote the lookup of the object stored at address ι on the heap H. The double lookup $H(\iota)(f)$ returns the address stored in the field f of object $H(\iota)$. The

$$\text{(NULL)}\ \frac{}{\sigma \cdot \mathsf{null}, H, T \to \sigma \cdot 0, H, T} \qquad \text{(THIS)}\ \frac{\sigma = (\iota, \_, \_, \_)}{\sigma \cdot \mathsf{this}, H, T \to \sigma \cdot \iota, H, T} \qquad \text{(FIELD-READ)}\ \frac{\iota' = H(\iota)(f)}{\sigma \cdot \iota.f, H, T \to \sigma \cdot \iota', H, T}$$

$$\text{(FIELD-WRITE)}\ \frac{H' = H[\iota \mapsto H(\iota)[f \mapsto \iota']]}{\sigma \cdot \iota.f = \iota', H, T \to \sigma \cdot \iota', H', T} \qquad \text{(ARG)}\ \frac{\sigma = (\_, \iota, \_, \_)}{\sigma \cdot \mathsf{x}, H, T \to \sigma \cdot \iota, H, T} \qquad \text{(CONTEXT)}\ \frac{\sigma \cdot e_r, H, T \to \sigma \cdot e_r', H', T'}{\sigma \cdot \mathcal{C}[e_r], H, T \to \sigma \cdot \mathcal{C}[e_r'], H', T'}$$

$$\text{(NEW)}\ \frac{\begin{array}{c}\mathit{FieldsOf}(c) = \{f_1, ..., f_n\} \\ \iota \text{ is fresh in } H \\ H' = H[\iota \mapsto (c, f_1 \mapsto 0, ..., f_n \mapsto 0)]\end{array}}{\sigma \cdot \mathsf{new}\ c, H, T \to \sigma \cdot \iota, H', T} \qquad \text{(MESSAGE-SEND)}\ \frac{\begin{array}{c}(c, \_) = H(\iota) \\ \mathit{MethodLookup}(m, c) = (e, c') \\ \sigma' = (\iota, \iota', m, \sigma) \\ \sigma' \cdot e, H, T \to \sigma' \cdot \iota'', H', T'\end{array}}{\sigma \cdot \iota.m(\iota'), H, T \to \sigma \cdot \iota'', H', T'}$$

$$\mathit{log}(T, v) ::= (T[\mathit{id} \mapsto v], \mathit{id}) \quad \text{where } \mathit{id} \text{ smallest } n \in \mathbb{N}, n \text{ fresh in } T$$

Figure 3. Reduction rules of L.

Source and runtime expressions

$e \in \mathit{Expr}$	::=	new c	*(new object)*
	|	this	*(this reference)*
	|	$e.m(e)$	*(message send)*
	|	x	*(argument)*
	|	$e.f$	*(field read)*
	|	$e.f{=}e$	*(field write)*
	|	null	*(null reference)*
$e_r \in \mathit{RExpr}$	::=	e	*(source expressions)*
	|	ι	*(address)*
	|	$\sigma \cdot e_r$	*(nested call)*

Dynamic aspects

$$\begin{array}{rcl} o \in \mathit{Object} & = & \mathit{Class} \times (\mathit{Field} \to \mathit{Address}) \\ \iota \in \mathit{Address} & = & \mathbb{N} \\ H \in \mathit{Heap} & = & \mathit{Address} \to \mathit{Object} \\ \sigma \in \mathit{StackFrame} & = & \mathit{Address} \times \mathit{Address} \times \mathit{Method} \times \mathit{Sender} \\ \mathit{Sender} & = & \mathit{StackFrame} \mid \epsilon \\ T \in \mathit{Trace} & = & \mathit{ID} \to \mathit{Event} \\ v \in \mathit{Event} & = & \text{(to be defined)} \end{array}$$

Reduction Contexts (call-by-value)

$$\mathcal{C} ::= [\,] \mid \mathcal{C}.m(e) \mid \iota.m(\mathcal{C}) \mid \mathcal{C}.f \mid \mathcal{C}.f = e \mid \iota.f = \mathcal{C} \mid \sigma \cdot \mathcal{C}$$

Figure 2. Syntax and dynamic aspects.

notation $o[f \mapsto \iota]$ denotes the update of o with binding $f \mapsto \iota$. The initial heap contains the object representing null at the fixed address 0: $H_0 = \{0 \mapsto (\mathit{UndefinedObject})\}$.

A stack frame σ is a 4-tuple composed of (1) the target address, (2) the argument address, (3) the method name, and (4) the sender frame. The initial stack frame is $\sigma_0 = (0, 0, \epsilon, \epsilon)$. The trace T is a store mapping identifiers to events. The definition of Event is intentionally missing and is specified by each analysis. The initial trace is $T_0 = \emptyset$. With $\mathcal{C}$ we define evaluation contexts taking the Wright-Felleisen approach of context-sensitive reductions [18].

3.3 Operational semantics

We use a big step operational semantics, given by the relation

$$\to \ \subseteq (\mathit{RExpr} \times \mathit{Heap} \times \mathit{Trace}) \times (\mathit{RExpr} \times \mathit{Heap} \times \mathit{Trace})$$

defined in Figure 3. A computation is started with $\sigma_0 \cdot \mathit{MainBody}(), H_0, T_0$.

The rule NEW creates an object with class c, and initializes all fields of the new object with null (*i.e.*, mapping fields to the address 0). The heap H is extended to the heap H' that additionally contains a binding for the new address ι. As with the following rules, the trace T is passed unchanged.

NULL, the rule for literal variable access (note, we only have one literal, null), replaces the syntactic element null with the reserved address 0. The heap is not modified. THIS and ARG fetch the target address, respectively the argument address, from the current stack frame (the underscore "_" is used as a placeholder for arguments that can be neglected for the present consideration). FIELD-READ looks up the object with address ι and then from this object the field f. FIELD-WRITE updates the heap with the mapping of the address ι to the object with updated mapping of the field f.

MESSAGE-SEND is slightly more complicated. First, the class c is extracted from the target object (the class is stored as the first element of an object tuple). Once c is determined, *MethodLookup* returns the expression e representing the body of the method and the class c' in which this method is implemented. In the second step, a new stack frame σ' is created as the tuple containing the target and argument. If the expression e in the context of the stack frame σ' and the heap H evaluates to the object at address ι'' with heap H', then the message send is replaced with ι'' (return value) and the new heap.

CONTEXT, in conjunction with evaluation contexts $\mathcal{C}$ (see Figure 2) limits the positions in which reduction rules can be applied. This defines how an expression is normalized as successive applications of the rules.

In addition to the reduction rules, we define *log*, which takes a trace T and an event v and returns a tuple consisting of a new trace and the id of the newly added event. The returned trace is the trace T with a new binding of an *id* to the event. (Note, we use the same notation for accessing and updating T as for the heap H.)

Many dynamic analyses model their events to have mutual relationships. Therefore, we represent the trace T as a store, which makes it possible for events to refer to each other through their unique identifier. To maintain the temporal order of events, the *id*s are increasing numbers.

We assume that programs are well-formed, *i.e.*, they do not access undefined fields or send messages that cannot be understood. This is a reasonable assumption given that our goal is to detail the semantics of an analysis.

The reduction rules of our base language L allow us to capture the ten events we selected in the previous Section 2: field read and write, this reference, argument read, literal reference, class instantiation, and message send are directly accessible through the respective reduction rule. The events method entry, method return, and method return value are less obvious because we use a big step operational semantics. These events can be captured in MESSAGE-SEND as this rule not only captures message sending but also already sets up the new stack frame, σ', and as it also captures the returned value, ι''. How to capture these events is shown in Figure 9 (Section 5).

4 INTRODUCTORY EXAMPLE

We demonstrate our approach on a specification of a very simple dynamic analysis: tracing the creation of objects. We capture the address of each new object and the stack frame of the method in which the object is instantiated.

Figure 4 specifies this dynamic analysis as follows. We define an event to be a 2-tuple with elements of the set *Address* and *StackFrame*. The definition of $Event$ completes the definitions of the dynamic aspects of L.

Extended dynamic aspects (in addition to Figure 2)

$v \in Event \quad = \quad Address \times StackFrame$
(address of created object and stack frame)

Figure 4. Event specification for object creation analysis.

Next we adapt the reduction rules as shown in Figure 5 to emit events to the trace. The only rule that we need to modify is NEW. In the modified definition, NEW reduces to a runtime expression with trace T'. The new trace T' is given by the convenience function log. The event that we emit is the tuple consisting of the address of the instantiated object, ι, and the current stack frame, σ.

(NULL), (THIS), (ARG), (FIELD-READ), (FIELD-WRITE), (MESSAGE-SEND), and (CONTEXT) same as in Figure 3

(NEW)

$$\frac{\begin{array}{c}FieldsOf(c) = \{f_1, ..., f_n\} \\ \iota \text{ is fresh in } H \\ H' = H[\iota \mapsto (c, f_1 \mapsto 0, ..., f_n \mapsto 0)] \\ (T', \_) = log(T, (\iota, \sigma))\end{array}}{\sigma \cdot \text{new } c, H, T \rightarrow \sigma \cdot \iota, H', T'}$$

Figure 5. Reduction rules for object creation analysis (differences compared to Figure 3 highlighted in gray).

To illustrate the specification of this analysis given by Figure 4 and Figure 5, we evaluate a simple example program P step by step. Figure 6 defines the program P with two classes. The first class, *IRBuilder*, having one field, the second class, *IRSequence*, having no fields. *MainBody*() is the function that returns the expressions of the main method. This code instantiates *IRBuilder* and sends the message *startNewSeq*. Since in our language L, each method requires exactly one parameter, we use *null*, although in the real code the method takes no parameters. The implementation of the method *startNewSeq* of the class *IRBuilder* creates a new *IRSequence* instance and assigns it to the field *currentSeq* of the target object.

In Figure 7 the reduction steps of executing P are listed. The evaluation of the initial expression requires three reduction steps (1–3). For the MESSAGE-SEND rule of step 3, three more reductions (3.1–3.3) are required as part of the rule's premise.

The rule NEW is applied in reduction steps 1 and 3.2. In the final state, the trace contains two events, one for each object on the heap (except for null). We can see that the first object (its address is ι_1) is created in main, whereas the second object (ι_2) is created in startNewSeq.

This example demonstrates how to use the proposed framework by extending its reduction rules. The example demonstrates that a specification can be quite small (only one single redefinition as shown in Figure 5). The default semantics provide all required definitions for syntax, dynamic aspects, and reduction rules so that the users can focus on the specification of their dynamic analysis.

5 CASE STUDY: TOD

The Trace Oriented Debugger (TOD) is a back-in-time debugger for Java [14], [19]. Back-in-time debuggers make it possible to navigate backwards within a program execution history, drastically improving the task of debugging. A formal description of queries is given to specify operations such as stepping, state and control flow reconstitution. The queries are based on the concepts of filters and cursor operations. While querying is well-defined, the publication does not detail the emission of the events. We apply our approach to evaluate how it can be used to formally specify TOD.

$$\begin{array}{rcl} Class & := & \{IRBuilder, IRSequence\} \\ Field & := & \{currentSeq\} \\ Method & := & \{startNewSeq\} \end{array}$$

MainBody() := **new** *IRBuilder.startNewSeq(null)*
MethodLookup($startNewSeq, IRBuilder$) :=
(*this.currentSeq* = **new** *IRSequence*, $IRBuilder$)

FieldsOf($IRBuilder$) := $\{currentSeq\}$
FieldsOf($IRSequence$) := $\{\}$

Figure 6. Definition of example program P.

$\sigma_0 \cdot \mathsf{new}\ IRBuilder.startNewSeq(\mathsf{null}),$
$\{0 \mapsto (UndefinedObject)\},$
$\{\}$

$\rightarrow^{(1)\ \textsc{New}}$
$\sigma_0 \cdot \iota_1.startNewSeq(\mathsf{null}),$
$\{0 \mapsto .., \iota_1 \mapsto (IRBuilder, currentSeq \mapsto 0)\},$
$\{id_1 \mapsto (\iota_1, \sigma_0)\}$

$\rightarrow^{(2)\ \textsc{Null}}$
$\sigma_0 \cdot \iota_1.startNewSeq(0),$
$\{0 \mapsto .., \iota_1 \mapsto (IRBuilder, currentSeq \mapsto 0)\},$
$\{id_1 \mapsto (\iota_1, \sigma_0)\}$

$(\iota_1, 0, startNewSeq, \sigma_0) \cdot \mathsf{this}.currentSeq := \mathsf{new}\ IRSequence,$
$\{0 \mapsto .., \iota_1 \mapsto (IRBuilder, currentSeq \mapsto 0)\},$
$\{id_1 \mapsto (\iota_1, \sigma_0)\}$

$\rightarrow^{(3.1)\ \textsc{This}}$
$(\iota_1, 0, \text{startNewSeq}, \sigma_0) \cdot \iota_1.currentSeq := \mathsf{new}\ IRSequence,$
$\{0 \mapsto .., \iota_1 \mapsto (IRBuilder, currentSeq \mapsto 0)\},$
$\{id_1 \mapsto (\iota_1, \sigma_0)\}$

$\rightarrow^{(3.2)\ \textsc{New}}$
$(\iota_1, 0, \text{startNewSeq}, \sigma_0) \cdot \iota_1.currentSeq := \iota_2,$
$\{0 \mapsto .., \iota_1 \mapsto (IRBuilder, currentSeq \mapsto 0), \iota_2 \mapsto (IRSequence)\},$
$\{id_1 \mapsto (\iota_1, \sigma_0), id_2 \mapsto (\iota_2, (\iota_1, 0, \text{startNewSeq}, \sigma_0))\}$

$\rightarrow^{(3.3)\ \textsc{Field-write}}$
$(\iota_1, 0, \text{startNewSeq}, \sigma_0) \cdot \iota_2,$
$\{0 \mapsto .., \iota_1 \mapsto (IRBuilder, currentSeq \mapsto \iota_2), \iota_2 \mapsto (IRSequence)\},$
$\{id_1 \mapsto (\iota_1, \sigma_0), id_2 \mapsto (\iota_2, (\iota_1, 0, \text{startNewSeq}, \sigma_0))\}$

$\rightarrow^{(3)\ \textsc{Message-send}}$
$\sigma_0 \cdot \iota_2,$
$\{0 \mapsto .., \iota_1 \mapsto (IRBuilder, currentSeq \mapsto \iota_2), \iota_2 \mapsto (IRSequence)\},$
$\{id_1 \mapsto (\iota_1, \sigma_0), id_2 \mapsto (\iota_2, (\iota_1, 0, \text{startNewSeq}, \sigma_0))\}$

Figure 7. Example evaluation of P with extended operational semantics.

5.1 Requirements for a formal specification of TOD

In the main publication of TOD [14], the following traced events are listed:

- Field write (FW)
- Local variable write (VW)
- Array write (AW)
- Exception (Ex)
- Behavior call (BC)
- Behavior enter (Bn)
- Behavior exit (Bx)

The events in the left column capture side effects to reconstruct object states and the events in the right column capture control flow to revert the debugger to previous call stacks.

For each event some of the following properties are recorded depending on the event type: timestamp, thread id, stack depth, pointer to parent event, source code location, field id, behavior id, local variable id, array index, value, return value, target, exception, arguments.

TOD also provides support for scoped trace capture. This means that the instrumentation scheme described above is selective. It is possible to supply user-defined filters that limit the number of emitted events. TOD supports class selectors, which are predicates on classes that should generate events.

5.2 Understanding the requirements

At first sight the gathering of the events and their properties seems straightforward. When looking more closely, though, the list above (and in the original TOD publication) is rather ambiguous. Closer reading reveals that the authors capture the "pointer to the parent event". However, no further explanation is given regarding the meaning of this parent. We contacted the authors and in private correspondence they explained that a parent event "is the Bn event that corresponds to the entry into the current method. It permits us to reconstitute the call."

In other words, any Bn event will have a parent. However, looking even deeper it is still unclear why the authors capture both a BC and a Bn, given that they appear redundant. When asked, the authors answered: "The reason we have both BC and Bn is that not all the program is instrumented. So sometimes instrumented code is called from non-instrumented code (so you only have Bn), and sometimes non-instrumented code is called by instrumented code (and you have only BC)."

When writing the specification, this answer triggered another question — a hidden edge case of scoping in relation to the parent event. Let us assume a Bn event is created in response to a message sent from non-instrumented code. What is the parent event of this Bn event? The answer is not obvious since the calling method is out of scope and hence does not have an associated Bn event. There are two possible solutions. Either the parent event pointer is undefined or it is the last triggered Bn event (that is, the Bn event of the method that called into code that is out of scope). How this is solved influences the query results. The latter solution seems to make more sense as it facilitates connecting two sequences of events — even if there are unknown method calls in between.

The above reveals two of the ambiguities present in the description of TOD's original publication [14]. Although all the events and their properties are listed, the paper does not define how this data is generated and hence lacks the completeness and precision that would be required to understand the approach in detail.

5.3 Specification

Representation of events. In a first step of the formal specification we define how to represent events. Figure 8 defines *Event* as a set of tuples of size 9. The elements of an event tuple are its parameters. At the first position the event type is given, then the stack depth, the parent event id, field name, *etc.* Fields and methods are given by their name, whereas field values, return values, the target and method argument are all object addresses. Depending on the event type, some elements of events are undefined, like in the informal specification in the TOD publication.

We have omitted the event *Ex* as our framework does not support exceptions without extension of the syntax

and reduction rules. For the sake of simplicity we have also omitted the event types of local and array writes. We do not miss significant precision because these two events are very similar to the field write event (*i.e.*, the local variable event has a variable index property instead of a field index property). From the list of properties we have omitted thread id and source code location as our formalization does not model multiple threads and the location of the source code is part of the static model (a mapping from expressions to source code).

Extending dynamic aspects. Extracting the values of the properties given in Figure 8 is straightforward except for the parent event id. The difficulty is that the parent event of a new event is created in a different execution context (or, more precisely, in a different reduction rule in the formalization). To specify the parent event property, we extend the definition of *StackFrame* of the base semantics (given in Figure 3) to associate an event id with each stack frame (see bottom of Figure 8).

Extended reduction rules. The event id of a stack frame is defined to hold the parent event id for this execution context. For instance, when creating the field write event in the rule (FIELD-WRITE), we can extract the id p from the current stack frame σ (see Figure 9) and pass it to the newly created log entry. Another special property stored in an event is the stack depth that is computed by the function *depth(σ)* (counting the number of frames in the current call stack). The other properties (field name, field value, target) can be directly accessed in the reduction rule.

The rule (FIELD-WRITE) illustrates how we specify structural scoping (see Figure 9). Depending on the predicate *inscope(σ)* a new event and trace are generated. If the current stack frame is out of scope, the trace is left unmodified. *inscope(σ)* is defined to depend on whether the class of the current target object is instrumented.

The reduction rule (MESSAGE-SEND) is similar. It models the events behavior call (BC), behavior enter (Bn), and behavior exit (Bx). The event BC is triggered when a message is sent, while Bn is triggered when the method is actually called. Since we do not have two rules for message send and method call, we capture both events in the (MESSAGE-SEND) reduction rule. The difference is nevertheless clear as BC is triggered in the calling context (σ) and Bn is triggered in the newly created context (σ'). Therefore, it is possible that only one or none of the two events is triggered — depending on whether their current context is in scope. The properties of the three events are the same except for the Bx event that additionally captures the returned value.

Note how the edge case is treated when the called method is not in scope. In this case the previous parent event id p is used for the new id p'. In this way the most recent parent id is carried along through non-instrumented method calls.

Extended dynamic aspects (in addition to Figure 2)

$$
\begin{array}{rcll}
Event & = & D_1 \times D_2 \times ... \times D_9 & \\
D_1 & = & \{FW, BC, Bn, Bx\} & \text{(event kind)} \\
D_2 & = & \mathbb{N} & \text{(stack depth)} \\
D_3 & = & ID & \text{(parent event id)} \\
D_4 & = & Field & \text{(field name)} \\
D_5 & = & Method & \text{(method name)} \\
D_6 & = & Address & \text{(field value)} \\
D_7 & = & Address & \text{(return value)} \\
D_8 & = & Address & \text{(target)} \\
D_9 & = & Address & \text{(argument)}
\end{array}
$$

$$StackFrame = Addr. \times Addr. \times Method \times Sender \times \mathbb{N}$$

Figure 8. TOD events and extension of *StackFrame*.

5.4 Evaluation

This case study has shown that a significant part of TOD can be formalized by modeling events, extending stack frames, and redefining four reduction rules. We left out three of seven events, but still managed to specify the crucial part of the analysis. Important questions arose that we had not thought about before and that could only be answered by the authors of TOD.

The required specification takes the space of half a page, and about one full page if the base syntax and dynamic aspects (Figure 2) are included (with explanations). That is, the specification of TOD could have been added as part of the original publication [14].

This case study indicates that our approach:

- provides a compact but appropriate basis to write new specifications with moderate effort,
- supports the most critical object-oriented features but lacks support for other events like exceptions that may be relevant in some cases,
- forces one to be precise and define important edge cases that otherwise are easily overlooked (an example is the parent event of a Bn called from non-instrumented code), and
- is flexible enough to be adapted to new requirements that are not readily supported (to extend the formalization to pass the parent event id between reduction steps required only minor adaptations).

6 CASE STUDY: OFA

In this second case study we present how our approach can be applied to Object Flow Analysis (OFA), a more complex analysis than the one of TOD. OFA is a dynamic analysis for object-oriented programs that captures how objects are passed through the system. OFA proved essential for building a scalable virtual machine (VM) for supporting practical back-in-time debugging [3]. In contrast to previous approaches, like TOD, Object Flow Analysis is not based on a sequential trace of events but on an extended model of the heap that keeps track

(NEW), (NULL), (FIELD-READ), and (CONTEXT) same as in Figure 3

(THIS)
$$\frac{\sigma = (\iota, \_, \_, \_, \_)}{\sigma \cdot \mathsf{this}, H, T \rightarrow \sigma \cdot \iota, H, T}$$

(ARG)
$$\frac{\sigma = (\_, \iota, \_, \_, \_)}{\sigma \cdot \mathsf{x}, H, T \rightarrow \sigma \cdot \iota, H, T}$$

(FIELD-WRITE)
$$\frac{\begin{array}{c} H' = H[\iota \mapsto H(\iota)[f \mapsto \iota']] \\ (\_, \_, \_, \_, p) = \sigma \\ (T', \_) = \begin{cases} log(T, (FW, depth(\sigma), \\ \quad p, f, \bot, \iota', \bot, \iota, \bot)) & \text{if } inscope(\sigma) \\ (T', \epsilon) & \text{else} \end{cases} \end{array}}{\sigma \cdot \iota.f = \iota', H, T \rightarrow \sigma \cdot \iota', H', T'}$$

(MESSAGE-SEND)
$$\frac{\begin{array}{c} (c, \_) = H(\iota) \\ (\_, \_, \_, \_, p) = \sigma \\ (T', \_) = \begin{cases} log(T, (BC, depth(\sigma), \\ \quad p, \bot, m, \bot, \bot, \iota, \iota')) & \text{if } inscope(\sigma) \\ (T, \epsilon) & \text{else} \end{cases} \\ \mathit{MethodLookup}(m, c) = (e, c') \\ p' \text{ is fresh in } T \\ \sigma' = (\iota, \iota', m, \sigma, p') \\ (T'', p') = \begin{cases} log(T', (Bn, depth(\sigma'), \\ \quad p, \bot, m, \bot, \bot, \iota, \iota')) & \text{if } inscope(\sigma') \\ (T', p) & \text{else} \end{cases} \\ \sigma' \cdot e, H, T'' \rightarrow \sigma' \cdot \iota'', H', T''' \\ (T'''', \_) = \begin{cases} log(T''', (Bx, depth(\sigma'), \\ \quad p, \bot, m, \bot, \iota'', \iota, \iota')) & \text{if } inscope(\sigma') \\ (T''', \epsilon) & \text{else} \end{cases} \end{array}}{\sigma \cdot \iota.m(\iota'), H, T \rightarrow \sigma \cdot \iota'', H', T''''}$$

$$inscope(\sigma) ::= c \in \mathsf{InstrumentedClasses} \text{ where } (c, \_) = H(\iota) \text{ and } \sigma = (\iota, \_, \_, \_, \_)$$

$$depth(\sigma) ::= \begin{cases} 1 & \text{if } (\_, \_, \_, \epsilon, \_) = \sigma \\ depth(\sigma') + 1 & \text{if } (\_, \_, \_, \sigma', \_) = \sigma \end{cases}$$

Figure 9. TOD Reduction rules of operational semantics (differences compared to Figure 3 highlighted in gray).

of historical runtime data. Object references are first-class entities (called *aliases*) and they hold additional information, such as the origin of the reference. OFA was also used for several approaches in reverse engineering and program understanding [2].

6.1 Requirements for a formal specification of OFA

The key step of OFA is the tracking of the transfer of object references at runtime. The underlying principle is to explicitly represent object references and to capture the relationships between these references. For example, when writing to a field or when passing an object as method argument, object flow analysis tracks the newly created references (the one stored in the field, respectively the one stored in the method argument). In this model the entity representing object references is called *Alias*. Relationships among aliases represent the transfer of references by capturing from which other alias an alias originates. By tracking the origins one can then accurately reconstruct the flow of each object in the system. Furthermore, the predecessor relationship of field aliases models side effects. For a detailed description of the model we refer the reader to our previous work [3].

OFA is a challenging analysis to formalize because it not only generates events that are stored in a trace (like in the case of TOD), but it requires modifications to the interpretation of the program because the structure of the heap needs to be modified. Since OFA requires significant modifications to the interpretation of the base language, we want to formally prove that the original semantics of the language are preserved. With this case study we demonstrate how our framework supports formal reasoning about the properties of dynamic analysis.

6.2 The specification

To formally specify OFA, we introduce a level of indirection between runtime expressions and objects. Object addresses in the extended language are represented by an alias record. Conceptually, an alias in the OFA model represents an *object reference*. Similar to objects, aliases are referred to by an address and are stored, separate from the main heap, in an alias store. The syntax and basic structure of the reduction rules are kept unchanged.

Figure 10 shows the extended syntax and dynamic aspects of L_{a}. In comparison to L (Section 3) we replace trace T with an alias store A. We renamed this store to make its use in the following explanations more clear. Expressions are unchanged except for the symbol κ that we use to refer to an address of an alias. Runtime expressions do not contain addresses to objects anymore.

6.2.1 Dynamic aspects

Addresses κ in runtime expressions refer to a binding in the alias store A, which maps addresses to aliases. An alias is a tuple $(\iota, \kappa_{orig}, \kappa_{pred}, \sigma)$, where ι is the address of the actual object, κ_{orig} is the address of the origin alias, κ_{pred} is the address of the predecessor alias, and σ is the stack frame. Depending on the class of an alias, κ_{orig} and κ_{pred} can be undefined ($\bot$). For conciseness, the class of an alias is not stored.

We define the following convenience function that yields the object address that an alias wraps.

Definition 1 *(Object of alias)*

$$o(\kappa, A) := \iota \quad \textit{where} \quad A(\kappa) = (\iota, \_, \_, \_)$$

Intuitively, the function o takes an alias address and an alias store, looks up the alias tuple in the alias store and then yields the object address located at the first position.

$$\text{(NULL)}\ \frac{(\kappa, A') = literalA(\sigma, A, 0)}{\sigma \cdot \mathsf{null}, H, A \to \sigma \cdot \kappa, H, A'} \qquad \text{(THIS)}\ \frac{\sigma = (\kappa, \_, \_, \_)}{\sigma \cdot \mathsf{this}, H, A \to \sigma \cdot \kappa, H, A} \qquad \text{(ARG)}\ \frac{\sigma = (\_, \kappa, \_, \_)}{\sigma \cdot \mathsf{x}, H, A \to \sigma \cdot \kappa, H, A}$$

$$\text{(NEW)}\ \frac{\begin{array}{c} FieldsOf(c) = \{f_1, ..., f_n\} \\ (\kappa_1, A_1) = writeA(\sigma, A_0, 0, \bot, \bot), ..., \\ (\kappa_n, A_n) = writeA(\sigma, A_{n-1}, 0, \bot, \bot) \\ \iota \text{ is fresh in } H \\ H' = H[\iota \mapsto (c, f_1 \mapsto \kappa_1, ..., f_n \mapsto \kappa_n)] \\ (\kappa, A') = allocA(\sigma, A_n, \iota) \end{array}}{\sigma \cdot \mathsf{new}\ c, H, A_0 \to \sigma \cdot \kappa, H', A'} \qquad \text{(CONTEXT)}\ \frac{\sigma \cdot e_r, H \to \sigma \cdot e_r', H'}{\sigma \cdot \mathcal{C}[e_r], H \to \sigma \cdot \mathcal{C}[e_r'], H'}$$

$$\text{(FIELD-READ)}\ \frac{\kappa_{orig} = H(o(\kappa, A))(f) \quad (\kappa'', A') = readA(\sigma, A, \kappa_{orig})}{\sigma \cdot \kappa.f, H, A \to \sigma \cdot \kappa'', H, A'}$$

$$\text{(FIELD-WRITE)}\ \frac{\begin{array}{c} \iota = o(\kappa, A) \\ \kappa_{pred} = H(\iota)(f) \\ (\kappa'', A') = writeA(\sigma, A, o(\kappa', A), \kappa', \kappa_{pred}) \\ H' = H[\iota \mapsto H(\iota)[f \mapsto \kappa'']] \end{array}}{\sigma \cdot \kappa.f = \kappa', H, A \to \sigma \cdot \kappa', H', A'}$$

$$\text{(MESSAGE-SEND)}\ \frac{\begin{array}{c} (c, \_) = H(o(\kappa, A)) \\ MethodLookup(m, c) = (e, c') \\ \kappa'' \text{ is fresh in } A \\ \sigma' = (\kappa, \kappa'', m, \sigma) \\ (\kappa'', A') = paramA(\sigma', A, \kappa') \\ \sigma' \cdot e, H, A' \to \sigma' \cdot \kappa''', H', A'' \\ (\kappa_4, A''') = \begin{cases} returnA(\sigma, A'', \kappa''') & if\ \kappa''' \neq \kappa \\ (\kappa, A'') & else \end{cases} \end{array}}{\sigma \cdot \kappa.m(\kappa'), H, A \to \sigma \cdot \kappa_4, H', A'''}$$

(Alias creation functions)

$$\begin{array}{rcl} allocA(\sigma, A, \iota) & := & (\kappa, A[\kappa \mapsto (\iota, \bot, \bot, \sigma)]) \\ literalA(\sigma, A, \iota) & := & (\kappa, A[\kappa \mapsto (\iota, \bot, \bot, \sigma)]) \\ paramA(\sigma, A, \kappa_{orig}) & := & (\kappa, A[\kappa \mapsto (o(\kappa_{orig}, A), \kappa_{orig}, \bot, \sigma)]) \\ returnA(\sigma, A, \kappa_{orig}) & := & (\kappa, A[\kappa \mapsto (o(\kappa_{orig}, A), \kappa_{orig}, \bot, \sigma)]) \\ readA(\sigma, A, \kappa_{orig}) & := & (\kappa, A[\kappa \mapsto (o(\kappa_{orig}, A), \kappa_{orig}, \bot, \sigma)]) \\ writeA(\sigma, A, \iota, \kappa_{orig}, \kappa_{pred}) & := & (\kappa, A[\kappa \mapsto (\iota, \kappa_{orig}, \kappa_{pred}, \sigma)]) \\ & & \text{...where } \kappa \text{ fresh in } A \end{array}$$

Figure 11. Extended reduction rules (differences to Figure 3 are highlighted in gray).

Source and runtime expressions

$$\begin{array}{rcll} e \in Expr & ::= & ... & \\ e_r \in RExpr & ::= & e & \textit{(source expressions)} \\ & | & \kappa & \textit{(alias addresses)} \\ & | & \sigma \cdot e_r & \textit{(nested call)} \end{array}$$

Dynamic aspects

$$\begin{array}{rcl} o \in Object & = & Class \times (Field \to Address) \\ a \in Alias & = & Address \times Address \times \\ & & Address \times StackFrame \\ \iota, \kappa \in Address & = & \mathbb{N} \\ H \in Heap & = & Address \to Object \\ A \in AliasStore & = & Address \to Alias \\ \sigma \in StackFrame & = & Address \times Address \times Method \times Sender \\ Sender & = & StackFrame \mid \epsilon \end{array}$$

Reduction Contexts (call-by-value)

$$\mathcal{C} ::= [\,] \mid \mathcal{C}.m(e) \mid \kappa.m(\mathcal{C}) \mid \mathcal{C}.f \mid \mathcal{C}.f = e \mid \kappa.f = \mathcal{C} \mid \sigma \cdot \mathcal{C}$$

Figure 10. Extended syntax and dynamic aspects (differences to Figure 2 highlighted in gray).

6.2.2 Operational semantics

The runtime semantics is given by the extended relation:

$$\to\ \subseteq (RExpr \times Heap \times AliasStore) \times (RExpr \times Heap \times AliasStore)$$

This relation is defined by the rules from Figure 11.

NEW creates two types of aliases. First, for each field in the new object, a write alias is created using the function $writeA()$ defined in Figure 11. The context of the write alias is the stack frame σ in which the object is created with new. Second, an allocation alias is created pointing to the new object ι and to the same stack frame σ.

NULL creates a literal alias for the object null. The rules THIS and ARG do not have to be modified, except for the address symbol ι that is replaced with κ to point out that the addresses in the stack frame point into the alias store instead of the heap.

FIELD-READ first extracts ι, the address of the actual object, which is stored at the first position of the alias tuple. Then the value of the field of the object is looked up and a read alias is created for this reference transfer. The origin alias of the field read alias is the alias κ_{orig} currently stored in the field.

FIELD-WRITE also first extracts the address of the actual object. It then looks up the current value of the field that is going to be changed. This value, aliased by κ_{pred}, is then remembered as the predecessor in the new field write alias. The origin of the field write alias is κ', the right hand side of the assignment. What this rule also shows is that the result of the assignment is the alias κ', that is, the original right hand side value rather than the newly created field write alias.

In the MESSAGE-SEND rule again two types of aliases are created. First, before the new method body is evaluated, an alias is created for the parameter. Note that the context σ' is the new stack frame, not the one in which the message m is sent. Furthermore, the target of the message send, the object address represented by the alias κ, is directly used as this in the new stack frame; no new alias is created in this case.

Under the condition that the address returned from the evaluation of e is not identical to this, a return alias is created. The rationale to not create a return alias on each return from a method call is that we want to capture only cases in which a different value than this is returned. This is especially important in languages like Smalltalk that implicitly return this. Also important to notice with return aliases is that their context σ is the stack frame to which they are returned, rather than the one from which they originate.

6.3 Example evaluation

We illustrate the extended operational semantics by evaluating the example program introduced in Section 4. The source code is given in Figure 6. At the left side of Figure 12 the reduction steps are listed. The initial state is an empty stack frame and the source expression of the main method body. The heap is initialized with the unique instance of null, and the alias store is empty. The evaluation of the initial expression requires three reduction steps (1–3). For the MESSAGE-SEND rule of step 3, three more reductions (3.1–3.3) are required as part of the rule's premise.

(1) In the first reduction step, (NEW), the target of the messsage send is reduced, which produces the new object with address ι_1 on the heap. This address is represented by the allocation alias κ_2. The field of the new object is initialized with a write alias κ_1 pointing to null.

(2) In the second reduction step, the literal null used as method parameter, is evaluated. This step produces a literal alias κ_3 but does not modify the heap.

(3) In the third reduction step, the method startNewSeq is called. For passing the parameter κ_3, a new parameter alias κ_4 is created, which is stored together with the target κ_2 in the new stack frame. The following three reduction steps are evaluated in this new context.

(3.1) This step simply applies the rule (THIS), which does not modify the heap or alias stack.

(3.2) This step reduces the right hand side of the assignment. It instantiates IRSequence, producing a new heap binding with address ι_2. In the runtime expression this object is represented by the allocation alias κ_5.

(3.3) The rule (FIELD-WRITE) is evaluated, producing a write alias κ_6 as follows. First the predecessor alias κ_{pred} is obtained by looking up the current alias stored in the field. The origin alias of the field write alias is the right hand side of the assignment since the object flows from there into the field.

Eventually, the resulting value of reduction step 3 is the return alias κ_7. At the right of Figure 12 the heap and the alias store of this example evaluation is illustrated. It shows the three objects created on the heap and the 7 aliases on the alias store. The different arrows indicate references between objects and aliases. Each alias points to its object (value), and the parameter, write, and return aliases point to their origin. The write alias in addition points to the alias previously stored in the field (predecessor). The field currentSeq first points to the write alias κ_1, and later to the write alias κ_6.

6.4 Proving that original semantics are preserved

Dynamic analysis is based on instrumentation of a program. This process should never alter its behavior. In a trivial case like logging of events it is obvious that the interpretation of the program is not modified. However, in the case of OFA, because it implies deep changes in the language model and in the virtual machine implementation, the preservation of the underlying semantics is not at all clear. In this case, we would need to be able to prove this constraint.

The formalization of OFA enables us to proof that an execution of a program with the original language L is behaviorally equivalent to an execution in L_a — that is, it merely generates additional data (aliases) but except for that the language semantics are the same. Intuitively, our proof shows that at any step of the execution of a program in the extended language, its flattened heap and flattened runtime expression is identical to the heap and runtime expression of the same program being executed up to this step in the original language.

To compare the similarity of a state s of a program in L_a with a state t of a program in L we define the relation F that relates s to t by flattening the heap (and runtime expression). Figure 13 illustrates the flattening of the final state of the example evaluation shown in Figure 12.

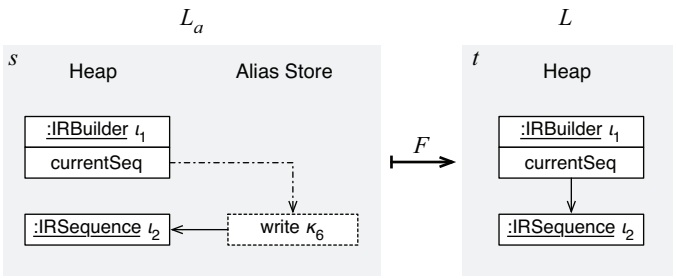

Figure 13. Relation F maps state s in language L_a to state t in language L.

We have defined the two languages as state transition systems. L_a is defined as $(S, \rightarrow)$ where $S = (RExpr \times Heap \times AliasStore)$ and L is defined as $(T, \rightarrow)$ where $T = (RExpr \times Heap)$. The definitions of $\rightarrow$ are given by the reduction rules in Figure 11. We now define the *simulation preorder* F, a relation between L_a and L, to show that each reduction step in L_a can be matched by a step in L. The simulation we define is a *strong simulation* (or lock-step simulation) as each step in L_a is matched by exactly one step in L.

$\sigma_0 \cdot (\text{new } IRBuilder).startNewSeq(\text{null}),$
$\{0 \mapsto (UndefinedObject)\},$
$\{\}$

$\rightarrow^{(1)\ \text{NEW}}$ $\sigma_0 \cdot \kappa_2.startNewSeq(\text{null}),$
$\{0 \mapsto .., \iota_1 \mapsto (IRBuilder, currentSeq \mapsto \kappa_1)\},$
$\{\kappa_1 \mapsto (0, ..), \kappa_2 \mapsto (\iota_1, ..)\}$

$\rightarrow^{(2)\ \text{NULL}}$ $\sigma_0 \cdot \kappa_2.startNewSeq(\kappa_3),$
$\{0 \mapsto .., \iota_1 \mapsto (IRBuilder, currentSeq \mapsto \kappa_1)\},$
$\{\kappa_1 \mapsto (0, ..), \kappa_2 \mapsto (\iota_1, ..), \kappa_3 \mapsto (0, ..)\}$

$(\kappa_2, \kappa_4) \cdot \text{this}.currentSeq := \text{new } IRSequence,$
$\{0 \mapsto .., \iota_1 \mapsto (IRBuilder, currentSeq \mapsto \kappa_1)\},$
$\{\kappa_1 \mapsto (0, ..), \kappa_2 \mapsto (\iota_1, ..), \kappa_3 \mapsto (0, ..), \kappa_4 \mapsto (0, \kappa_3, ..)\}$

$\rightarrow^{(3.1)\ \text{THIS}}$ $(\kappa_2, \kappa_4) \cdot \kappa_2.currentSeq := \text{new } IRSequence,$
$\{0 \mapsto .., \iota_1 \mapsto (IRBuilder, currentSeq \mapsto \kappa_1)\},$
$\{\kappa_1 \mapsto (0, ..), \kappa_2 \mapsto (\iota_1, ..), \kappa_3 \mapsto (0, ..), \kappa_4 \mapsto (0, \kappa_3, ..)\}$

$\rightarrow^{(3.2)\ \text{NEW}}$ $(\kappa_2, \kappa_4) \cdot \kappa_2.currentSeq := \kappa_5,$
$\{0 \mapsto .., \iota_1 \mapsto (IRBuilder, currentSeq \mapsto \kappa_1), \iota_2 \mapsto (IRSequence)\},$
$\{\kappa_1 \mapsto (0, ..), \kappa_2 \mapsto (\iota_1, ..), \kappa_3 \mapsto (0, ..), \kappa_4 \mapsto (0, \kappa_3, ..), \kappa_5 \mapsto (\iota_2, ..)\}$

$\rightarrow^{(3.3)\ \text{FLD-W.}}$ $(\kappa_2, \kappa_4) \cdot \kappa_5,$
$\{0 \mapsto .., \iota_1 \mapsto (IRBuilder, currentSeq \mapsto \kappa_6), \iota_2 \mapsto (IRSequence)\},$
$\{\kappa_1 \mapsto (0, ..), \kappa_2 \mapsto (\iota_1, ..), \kappa_3 \mapsto (0, ..), \kappa_4 \mapsto (0, \kappa_3, ..), \kappa_5 \mapsto (\iota_2, ..),$
$\kappa_6 \mapsto (\iota_2, \kappa_5, \kappa_1, ..)\}$

$\rightarrow^{(3)\ \text{MSG-SEND}}$ $\sigma_0 \cdot \kappa_7,$
$\{0 \mapsto .., \iota_1 \mapsto (IRBuilder, currentSeq \mapsto \kappa_6), \iota_2 \mapsto (IRSequence)\},$
$\{\kappa_1 \mapsto (0, ..), \kappa_2 \mapsto (\iota_1, ..), \kappa_3 \mapsto (0, ..), \kappa_4 \mapsto (0, \kappa_3, ..), \kappa_5 \mapsto (\iota_2, ..),$
$\kappa_6 \mapsto (\iota_2, \kappa_5, \kappa_1, ..), \kappa_7 \mapsto (\iota_2, \kappa_5, ..)\}$

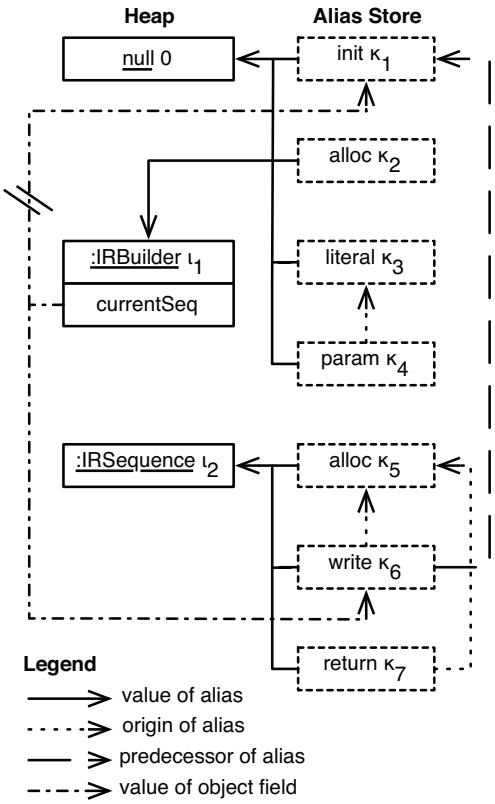

Figure 12. Left: example evaluation with extended semantics. Right: resulting heap and alias store.

Proposition 1 *The relation $F \subseteq (S \times T)$ is a simulation, that is, $(s,t) \in F$ implies that for s' with $s \rightarrow s'$ there is t' such that $t \rightarrow t'$ and $(s',t') \in F$.*

Figure 14 illustrates Proposition 1. Solid lines indicate hypotheses and dashed lines indicate conclusions. To verify our proposition, we first define the relation F.

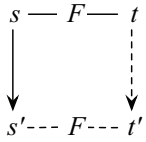

Figure 14. Simulation diagram

The relation F flattens a runtime expression and a heap with respect to a given alias store.

Definition 2 *(Simulation preorder F)*

$$F := \{(RExpr, H, A), (f_r(RExpr, A), f_h(H, A))\} \quad \textit{where}$$

$$f_r(e_r, A) := \begin{cases} e & \textit{if} \quad e_r = e \\ o(\kappa, A) & \textit{if} \quad e_r = \kappa \\ f_\sigma(\sigma, A) \cdot f_r(e'_r, A) & \textit{if} \quad e_r = \sigma \cdot e'_r \end{cases}$$

$$\textit{where} \quad f_\sigma((\kappa, \kappa'), A) := (o(\kappa, A), o(\kappa', A))$$

$$\textit{and}$$

$$f_h(H, A) := \{\iota_1 \mapsto (C, f_1 \mapsto o(\kappa, A), ...), ...\iota_n \mapsto ...\}$$

$$\textit{where} \quad H = \{\iota_1 \mapsto (C, f_1 \mapsto \kappa, ...), ...\iota_n \mapsto ...\}$$

Intuitively, f_r takes a runtime expression and replaces each occurrence of an alias address κ with the corresponding object address $o(\kappa, A)$. And f_h replaces all alias addresses referred to by fields of objects on the heap with corresponding object addresses. Details of the proof are provided in the appendix.

6.5 Evaluation

The OFA case study shows that:

- Our approach supports the specification of sophisticated dynamic analyses with relative ease. The introduction of an alias store is a major change of how the runtime works as direct addressing is replaced by an additional level of indirection. Yet, specification requires no fundamental rewrite of the dynamic aspects and rules of the base language.
- Our approach provides a good framework to reason about properties of the dynamic analysis. Since the implementation of OFA potentially has inadvertent side effects on the behavior of programs, it is worthwhile to show that the analysis is correct, *i.e.*, to prove that the semantics are not altered.

7 Related Work

Operational semantics, pioneered by Plotkin [17], is a well known way of specifying language semantics in

terms of sequences of computational steps. A large body of research exists that uses operational semantics. Our approach uses language semantics as a way to formally define and reason about dynamic analysis.

The minimal object language on which our framework is based is influenced by established approaches [9], [10], [11]. The main difference in the syntax and reduction rules compared to the approach by Drossopoulou *et al.* [11] is that we unify primitive type values and reference values. In our language, we refer to primitive values, such as null, in runtime expressions through addresses. That is, the value null is represented as an instance on the heap. This unification simplifies our extended reduction rules because an explicit distinction between primitive type values and addresses can be avoided. Furthermore, message send is specified as a big step operational semantics as in a recent publication of Clarke and Drossopoulou [10], which makes passing of parameters and return values more explicit.

Operational semantics is being used as a foundation to formalize and reason about dynamic analyses. Several publications in the field of runtime verification, like the work of Flanagan *et al.* [6], [7], [8], use dedicated operational semantics. Their approach is to select a small set of operations relevant for the problem at hand (atomicity checking) and to define additional "instrumentation rules" that are applied to a sequence of operations. In comparison to our framework, their approach does not define complete semantics (because it is not needed for their problem), *i.e.*, it lacks source and runtime expressions, objects, message sending, etc. Their formalization assumes the existence of a trace of operations but does not define how this trace is generated. Their approach also differs in that it uses a layer of instrumentation rules whereas we propose to modify the relevant reduction rules directly. Being able to do "invasive" modifications is necessary for dynamic analyses that require changes in the execution of the program.

Numerous researchers have investigated the semantics of AOP, often building on the work of Walker *et al.* [20]. Walker *et al.* present a core calculus for first-class aspects, which identifies program points for instrumentation through explicit labels. In contrast to this static approach that uses labels, the formalization of Wand [21] *et al.*, a denotational semantics, avoids an intermediate level and formalizes events that happen at runtime. A notable related work in this area is the formal framework of Klose and Ostermann for the comparison of pointcut languages [22]. Their framework allows one to specify joinpoints on a reduced object-oriented language. Joinpoints, which locate program points that can be used to trace runtime data, are specified using syntactic elements and reduction contexts of a small-step operational semantics. While focusing on joinpoint models and pointcut languages, the underlying language semantics are left unspecified. In contrast to our approach, their object language does not define a heap nor syntax and semantics for field write, argument access, *etc.*. The key difference is that with our approach the generation of events is specified by extending the reduction rules of the operational semantics whereas in their approach joinpoints are defined at a higher level of abstraction. Hence, our approach leaves more freedom to define where and exactly what data is logged, which is important as the application of our framework is intended to be as open as possible to be applicable to future, yet unknown, dynamic analyses.

Another interesting related work in the area of AOP formalization is TraceMatches [23] as it proposes an extension to AspectJ [24] that allows programmers to trigger code depending on patterns of events in an execution trace. The execution trace is given by enter and exit events from standard AOP joinpoints. TraceMatches provides a framework based on a formal semantics that can potentially also be used to define dynamic analyses. The main difference from our approach is that TraceMatches does not allow one to specify how the execution trace is generated. The possible patterns are limited by the events provided by the joinpoint model. Hence, TraceMatches is not generic enough to be used for dynamic analyses that depend on specialized traces (like the presented example of the Trace Oriented Debugger that passes an id along non-instrumented execution paths) or analyses that are based on a dedicated runtime instrumentation like Object Flow Analysis.

In the book "Virtual Machines", Iain D. Craig provides specifications for VMs of simple imperative languages [25] using a transition system similar to our operational semantics. While his specification operates on single instructions, our reduction rules work at the level of source expressions. As such, our framework is independent of a specific implementation model (*e.g.*, stack- vs. register-based VMs). The abstraction level of source expressions in our view is more appropriate for specifying dynamic analyses as one is usually not interested in tracing, for instance, single operations of the operand stack.

Belblidia *et al.* propose operational semantics for the Java VM [26]. For each bytecode instruction a reduction rule is specified. The reduction rules are at a lower level of abstraction and have to take the idiosyncrasies of Java into account. The semantics also captures exceptions and multi-threading, which we have omitted for sake of conciseness. This work shows how our approach could be extended to include these language features.

8 Conclusions

We propose that dynamic analysis approaches are formally specified by extending language semantics. We propose the L language, which models key OO concepts, including a heap and field assignment. L is concise, therefore it can be understood and applied to new dynamic analyses with modest effort. The semantics of L is expressed by means of operational semantics

— an approach established in the field of static analysis. We show how dynamic analyses can be specified by extending its reduction rules. We demonstrated how two complex analyses can be formally specified using L.

The TOD case study dealt with logging of events. The authors neglected to explain what exactly the parent of an event is, although this is key to their analysis. Even if apparently a small oversight, this leads to important ambiguities regarding the exact collection of traces. The L specification exposed the ambiguity and enabled it to be resolved.

The OFA case study showed how we can capture the semantics of the runtime objects' dynamics. Given the complex changes it performs on the core of the language, it is no longer obvious whether the semantics of the underlying language is preserved. Thus, we have shown how our formalization provides the basis to reason about a dynamic analysis, and eventually to prove that it does not alter the language semantics. In the case of OFA the actual implementation of the VM followed closely the described model, and this proof is particularly important because it conceptually ensured that the changes in the VM preserved the behavior of existing programs.

Overall, the two case studies are significantly distinct and thus show that our language covers an important area of analysis. Furthermore, even if these analyses concern many details, the specifications are still succinct.

ACKNOWLEDGMENT

We gratefully acknowledge the financial support of the Swiss National Science Foundation for the projects "Bringing Models Closer to Code" (No. 200020-121594, Oct. 2008 - Sept. 2010) and "Synchronizing Models and Code" (No. 200020-131827, Oct. 2010 - Sept. 2012).

REFERENCES

[1] B. Cornelissen, A. Zaidman, A. van Deursen, L. Moonen, and R. Koschke, "A systematic survey of program comprehension through dynamic analysis," *IEEE Transactions on Software Engineering*, vol. 35, no. 5, pp. 684–702, 2009.

[2] A. Lienhard, S. Ducasse, and T. Gîrba, "Taking an object-centric view on dynamic information with object flow analysis," *Journal of Computer Languages, Systems and Structures*, vol. 35, no. 1, pp. 63–79, 2009.

[3] A. Lienhard, T. Gîrba, and O. Nierstrasz, "Practical object-oriented back-in-time debugging," in *Proceedings of the 22nd European Conference on Object-Oriented Programming (ECOOP'08)*, ser. LNCS, vol. 5142. Springer, 2008, pp. 592–615, ECOOP distinguished paper award.

[4] J. Quante and R. Koschke, "Dynamic object process graphs," *Journal of Systems and Software*, vol. 81, no. 4, pp. 481–501, 2008.

[5] ——, "Dynamic object process graphs," in *Proceedings 10th European Conference on Software Maintenance and Reengineering (CSMR'06)*. IEEE Computer Society Press, 2006, pp. 81–90.

[6] C. Flanagan, S. N. Freund, and J. Yi, "Velodrome: a sound and complete dynamic atomicity checker for multithreaded programs," in *PLDI '08: Proceedings of the 2008 ACM SIGPLAN conference on Programming language design and implementation*. New York, NY, USA: ACM, 2008, pp. 293–303.

[7] C. Flanagan and S. N. Freund, "FastTrack: efficient and precise dynamic race detection," in *PLDI '09: Proceedings of the 2009 ACM SIGPLAN conference on Programming language design and implementation*. New York, NY, USA: ACM, 2009, pp. 121–133.

[8] ——, "Adversarial memory for detecting destructive races," in *PLDI '10: Proceedings of the 2010 ACM SIGPLAN conference on Programming language design and implementation*. New York, NY, USA: ACM, 2010, pp. 244–254.

[9] N. R. Cameron, S. Drossopoulou, J. Noble, and M. J. Smith, "Multiple ownership," in *Proceedings of the 22nd annual ACM SIGPLAN conference on Object oriented programming systems and applications (OOPSLA'07)*. New York, NY, USA: ACM, 2007, pp. 441–460.

[10] D. Clarke and S. Drossopoulou, "Ownership, encapsulation and the disjointness of type and effect," in *Proceedings of the 17th ACM SIGPLAN conference on Object-oriented programming, systems, languages, and applications (OOPSLA'02)*. New York, NY, USA: ACM, 2002, pp. 292–310.

[11] S. Drossopoulou, A. Francalanza, P. Müller, and A. Summers, "A unified framework for verification techniques for object invariants," in *Proceedings of 22nd European Conference on Object-Oriented Programming (ECOOP'08)*, ser. Lecture Notes in Computer Science, Jul. 2008, pp. 412–437.

[12] B. M. Cantrill, M. W. Shapiro, and A. H. Leventhal, "Dynamic instrumentation of production systems," in *Proceedings of USENIX 2004 Annual Technical Conference*. Berkeley, CA, USA: USENIX Association, 2004, pp. 15–28.

[13] W. De Pauw, D. Kimelman, and J. Vlissides, "Modeling object-oriented program execution," in *Proceedings of the European Conference on Object-Oriented Programming (ECOOP'94)*, ser. LNCS, M. Tokoro and R. Pareschi, Eds., vol. 821. Bologna, Italy: Springer-Verlag, Jul. 1994, pp. 163–182.

[14] G. Pothier, E. Tanter, and J. Piquer, "Scalable omniscient debugging," *Proceedings of the 22nd Annual SCM SIGPLAN Conference on Object-Oriented Programming Systems, Languages and Applications (OOPSLA'07)*, vol. 42, no. 10, pp. 535–552, 2007.

[15] D. Rayside and L. Mendel, "Object ownership profiling: a technique for finding and fixing memory leaks," in *Proceedings of the twenty-second IEEE/ACM international conference on Automated software engineering (ASE'07)*. New York, NY, USA: ACM, 2007, pp. 194–203.

[16] S. Pheng and C. Verbrugge, "Dynamic data structure analysis for java programs," in *Proceedings of the 14th IEEE International Conference on Program Comprehension (ICPC'06)*. Washington, DC, USA: IEEE Computer Society, 2006, pp. 191–201.

[17] G. Plotkin, "A structural approach to operational semantics," University of Aarhus, Denmark, Tech. Rep., 1981.

[18] A. K. Wright and M. Felleisen, "A syntactic approach to type soundness," *Inf. Comput.*, vol. 115, no. 1, pp. 38–94, 1994.

[19] G. Pothier and É. Tanter, "Back to the future: Omniscient debugging," *IEEE Software*, vol. 26, no. 6, pp. 78–85, 2009.

[20] D. Walker, S. Zdancewic, and J. Ligatti, "A theory of aspects," in *ICFP '03: Proceedings of the eighth ACM SIGPLAN international conference on Functional programming*. New York, NY, USA: ACM, 2003, pp. 127–139.

[21] M. Wand, G. Kiczales, and C. Dutchyn, "A semantics for advice and dynamic join points in aspect-oriented programming," *ACM Trans. Program. Lang. Syst.*, vol. 26, no. 5, pp. 890–910, 2004.

[22] K. Klose and K. Ostermann, "A classification framework for pointcut languages in runtime monitoring," in *Objects, Components, Models and Patterns, Proceedings of TOOLS Europe 2009*, ser. LNBIP, vol. 33. Springer-Verlag, 2009, pp. 289–307.

[23] C. Allan, P. Avgustinov, A. S. Christensen, L. Hendren, S. Kuzins, O. Lhoták, O. de Moor, D. Sereni, G. Sittampalam, and J. Tibble, "Adding trace matching with free variables to AspectJ," in *OOPSLA '05: Proceedings of the 20th annual ACM SIGPLAN conference on Object-oriented programming systems and applications*. New York, NY, USA: ACM, 2005, pp. 345–364.

[24] "AspectJ home page," http://eclipse.org/aspectj/.

[25] I. D. Craig, *Virtual machines*. Berlin, Germany: Springer Verlag, 2005.

[26] N. Belblidia and M. Debbabi, "A dynamic operational semantics for JVML," *Journal of Object Technology*, vol. 6, no. 3, 2007.

APPENDIX

Proof of Proposition 1. We case split on the reduction rules $\rightarrow$ of L_a. For each rule, we first take the path to the right and then to the bottom in Figure 14 (that is, s mapped to t reduced to t') and then we take the other path (s reduced to s' mapped to t') to show that we obtain the identical t' along both paths.

Case (NULL): We start with this case because it is not trivial but also not the most complex one. The two listings below show each step at the left and how a step was derived at its right. It has to be proven that both resulting states are equal.

$(\sigma \cdot \mathsf{null}, H, A)$	
$(f_r(\sigma \cdot \mathsf{null}, A), f_h(H, A))$	Definition 2
$(f_\sigma(\sigma, A) \cdot \mathsf{null}, f_h(H, A))$	Definition 2
$(f_\sigma(\sigma, A) \cdot 0, f_h(H, A))$	(NULL) in L

$(\sigma \cdot \mathsf{null}, H, A)$	
$(\sigma \cdot \kappa, H, A')$ where $\kappa = (0, \bot, \bot, \sigma)$	(NULL) in L_a
$(f_\sigma(\sigma, A) \cdot o(\kappa, A), f_h(H, A))$	Definition 2
$(f_\sigma(\sigma, A) \cdot 0, f_h(H, A))$	Definition 1

Case (NEW): To proof this case we introduce:

Lemma 1. $f_h(H[\iota \mapsto (c, f \mapsto \kappa)], A) = f_h(H, A)[\iota \mapsto (c, f \mapsto o(\kappa, A))]$

This lemma says that updating the binding of a heap in L_a and then flattening this heap is equivalent to first flattening the heap and then updating the binding with the same object but with its alias addresses being unwrapped. This lemma follows directly from f_h in Definition 2.

Like in the previous case, we follow the two paths to show that they lead to the identical state:

$\sigma \cdot \mathsf{new}\ c, H, A$	
$f_\sigma(\sigma, A) \cdot \mathsf{new}\ c, f_h(H, A)$	Definition 2
$f_\sigma(\sigma, A) \cdot \iota, H'$	(NEW) in L
where $H' = f_h(H, A)[\iota \mapsto (c, f_1 \mapsto 0, ..., f_n \mapsto 0)]$	

$\sigma \cdot \mathsf{new}\ c, H, A$	
$\sigma \cdot \kappa, H', A'$	(NEW) in L_a
where $H' = H[\iota \mapsto (c, f_1 \mapsto \kappa_1, ..., f_n \mapsto \kappa_n)]$	
and $A' = A[\kappa_1 \mapsto (0, ...)]...[\kappa_n \mapsto (0, ...)][\kappa \mapsto (\iota, ...)]$	
$f_\sigma(\sigma, A) \cdot o(\kappa, A'), f_h(H', A')$	Definition 2
$f_\sigma(\sigma, A) \cdot \iota, f_h(H', A')$	Definition 1
$f_\sigma(\sigma, A) \cdot \iota, H''$	Lemma 1
where $H'' = f_h(H, A')[$	
$\iota \mapsto (c, f_1 \mapsto o(\kappa_1, A'), ..., f_n \mapsto o(\kappa_n, A')]$	
$f_\sigma(\sigma, A) \cdot \iota, H''$	Definition 1
where $H'' = f_h(H, A')[\iota \mapsto (c, f_1 \mapsto 0, ..., f_n \mapsto 0)]$	

The differences between A and A' are the new bindings for $\kappa_1, ..., \kappa_n$ and κ, which all are fresh in A, and hence $f_h(H, A) = f_h(H, A')$. Therefore, both states are equivalent.

Case (THIS) and (ARG): Follow directly from reduction rules and Definition 2.

Case (FIELD-READ): We first proof the following lemma, which is the counterpart to Lemma 1.

Lemma 2. $f_h(H, A)(\iota)(f) = o(H(\iota)(f), A)$

This lemma says that flattening a heap and then looking up a field yields the same object address as first looking up the field and then unwrapping the returned alias.

let $H = \{..., \iota \mapsto (\_, f \mapsto \kappa, ...), ...\}$ be a heap in L_a, then:

$f_h(H, A)(\iota)(f)$	
$H'(\iota)(f)$	Definition 2 and H
where $H' = \{..., \iota \mapsto (\_, f \mapsto o(\kappa, A), ...), ...\}$	
$o(\kappa, A)$	field lookup in H'
$o(H(\iota)(f), A)$	binding of κ in H

Having introduced Lemma 2, we can now again proof that both paths in the case of the rule (FIELD-READ) are equivalent.

$\sigma \cdot \kappa.f, H, A$	
$f_\sigma(\sigma) \cdot o(\kappa, A).f, f_h(H, A)$	Definition 2
$f_\sigma(\sigma) \cdot f_h(H, A)(o(\kappa, A))(f), f_h(H, A)$	(FIELD-READ) in L
$f_\sigma(\sigma) \cdot o(H(o(\kappa, A))(f), A), f_h(H, A)$	Lemma 2

$\sigma \cdot \kappa.f, H, A$	
$\sigma \cdot \kappa'', H, A'$	(FIELD-R.) in L_a
where $A' = A[k'' \mapsto (o(H(o(\kappa, A))(f), A), ...)]$	
$f_\sigma(\sigma) \cdot o(\kappa'', A'), f_h(H, A')$	Definition 2
$f_\sigma(\sigma) \cdot o(H(o(\kappa, A))(f), A), f_h(H, A')$	

The difference between A and A' is the update of κ'', which is fresh in A, and hence $f_h(H, A) = f_h(H, A')$. Therefore, both end states are equivalent.

Case (FIELD-WRITE): This case is analogous to (FIELD-READ), except that Lemma 1 instead of Lemma 2 is used.

Case (MESSAGE-SEND): For the intermediate reduction that is part of the rule's premise, we have to show that its left hand side is equivalent with respect to F to the left hand side of the same rule in L. In particular, this means to show that for σ' in L_a $f_\sigma(\sigma')$ is equivalent to σ' in L. This is straightforward because it requires only to show that the parameter alias is an alias of the object passed as argument.

Case (CONTEXT): Follows directly since the rules are identical in both languages and the reduction contexts defined by $\mathcal{C}$ preserve the reduction order.

□

Adrian Lienhard holds a PhD in Computer Science from the University of Bern, Switzerland. He is development lead at Cmsbox, a startup he co-founded. Cmsbox provides a novel web content management system, and has been awarded by the Nielsen Norman Group.

Tudor Gîrba attained his PhD in 2005, and he now works as a consultant on software and data assessment. Since 2003, he leads the work on the Moose analysis platform. He coined the term *humane assessment*, and he is currently helping companies to assess and manage large software systems and data sets.

Oscar Nierstrasz is full Professor of Computer Science at the Institute of Computer Science (IAM) of the University of Bern, where he founded the Software Composition Group in 1994. He holds a BMath from the University of Waterloo (1979), and a MSc (1981) and PhD (1984) in Computer Science from the University of Toronto. His current research focuses on various aspects of software evolution.