Open Bisimulation for Aspects

Radha Jagadeesan Corin Pitcher James Riely
DePaul University
{rjagadeesan,cpitcher,jriely}@cs.depaul.edu

Abstract

We define and study bisimulation for proving contextual equivalence in an aspect extension of the untyped lambda-calculus. To our knowledge, this is the first study of coinductive reasoning principles aimed at proving equality of aspect programs. The language we study is very small, yet powerful enough to encode mutable references and a range of temporal pointcuts (including cflow and regular event patterns).

Examples suggest that our bisimulation principle is useful. For an encoding of higher-order programs with state, our methods suffice to establish well-known and well-studied subtle examples involving higher-order functions with state.

Even in the presence of first class dynamic advice and expressive pointcuts, our reasoning principles show that aspect-aware interfaces can aid in ensuring that clients of a component are unaffected by changes to an implementation. Our paper generalizes existing results given for *open modules* to also include a variety of history-sensitive pointcuts such as cflow and regular event patterns.

Our formal techniques and results suggest that aspects are amenable to the formal techniques developed for stateful higher-order programs.

Categories and Subject Descriptors D.3.1 [*Programming Languages*]: Formal Definitions and Theory—semantics; D.3.3 [*Programming Languages*]: Language Constructs and Features—modules, packages; F.3.2 [*Logics and Meanings of Programs*]: Semantics of Programming Languages—operational semantics

General Terms languages, equational reasoning

Keywords aspect-oriented programming, contextual equivalence, open bisimulation, modularity, modular reasoning

1. Introduction

Aspects have emerged as a powerful tool in the design and development of systems [10, 31, 49, 41, 32, 5]. A (much-overused!) standard profiling example from the AspectJ tutorials suffices to introduce the basic vocabulary. Suppose class *L* realizes a useful library, and we want to obtain timing information about a method `foo()` of *L*. With aspects this can be done by writing *advice* specifying that, whenever `foo` is called, the current time should be logged, `foo` should be executed, and then the current time should again be logged. Aspects permit the profiling code to be localized in the advice, transferring the responsibility for coordinating the advice and base code to a compiler or runtime environment. This ensures that the developer of the library need not worry about advice that may be written in the future — in [20] this is called *obliviousness*. However, in writing the logging advice, one must identify the pieces of code, using *pointcuts*, that need to be logged — in [20] this is called *quantification*. Aspect-orientation ideas for representing and composing crosscutting concerns such as logging are paradigm-independent and have been developed for object-oriented [31, 59] imperative [15] and functional languages [61, 18].

Our focus in this paper is on the intersubstitutivity of programs written in an aspect-oriented extension of a functional language: when can one program fragment be substituted for another without altering the observable behavior of the program? A basic tool that has been used to address this question for other programming paradigms has been coinduction, in the form of bisimulation principles. While the origins of bisimulation trace back to concurrency theory (see [55, 56] for a comprehensive historical survey and detailed bibliography), bisimulation principles have proven to be quite useful to address program equality in several paradigms, e.g., higher-order languages (see [50, 22] for a detailed treatment with historical context), even in the presence of existential types [58] or state [36, 29], and object-oriented languages [23, 34].

This paper brings aspect-based languages within the ambit of this technique. Our formal techniques and results suggest that aspects are no more intractable than stateful higher-order programs. In first order languages with first order references, when reasoning about programs, the environment has only two ways to interact with a program: either via global shared variables or by invoking the program (that can of course result in changes in encapsulated private state of the program). In higher-order languages with higher-order references, a program can also "leak" local state externally via higher-order mechanisms providing the environment a third way to interact with a program. Our results suggest that mechanisms that address this feature of higher-order languages with state may be adapted to an aspect framework with dynamic aspects.

Our main technical contributions are as follows:

Bisimulation. We study a core untyped lambda calculus, enhanced with aspects and named functions. Advice is first class in our calculus: it can be created and added dynamically while a program is running. The language can code mutable higher-order references and expressive pointcuts such as cflow and regular event patterns.

We describe a bisimulation principle based on a labelled transition system for aspect programs. We show that bisimulation is sound and complete for contextual congruence.

We demonstrate the usability of the bisimulation principle via examples using the encoding of mutable variables — we show that several of the program equalities suggested by Meyer and Sieber [45] are validated by our bisimulation principle.

AOSD 07 March 12-16, 2007, Vancouver, Canada.
Copyright © 2007 ACM 1-59593-615-7/07/03. . . $5.00
Reprinted from AOSD 07, [Unknown Proceedings], March 12-16, 2007, Vancouver, Canada., pp. 1–21.

Application to Open Modules. Aspect-Aware Interfaces [33] enhance the usual signature information of modules with the pointcuts that are exported by the module and visible to the clients of the module. This enhancement of traditional signatures facilitates extra reasoning by providing bounds on the use of advice. An Open Module [6] delineates conditions about when it is permissible to replace the implementation of a module with another.

The formal treatment of Open Modules [6] only permits call pointcuts, whereas the implementation of Open Modules in AspectJ [48] also permits cflow pointcuts. Recent research on more expressive pointcut languages motivate the desirability and implementability of more expressive pointcuts, e.g., those match regular patterns against the whole computation history [9]. We use our bisimulation principle to bridge this expressiveness gap.

To address these issues, our core calculus supports mechanisms to delimit the scope of the program where a function can be advised. We do this by providing named primitive pointcuts. Each function and advice declaration is associated with a primitive pointcut. Advice applies to a function only if its associated primitive pointcut is the same as that of the function. We use normal scoping mechanisms to control the knowledge of primitive pointcuts. The use of named primitive pointcuts as a separate construct permits the scope of the "advise"-access to vary separately from the standard scope of direct access to the function reference.

This framework permits the use of our bisimulation principle to establish conditions under which implementations can be changed without affecting clients, even in the presence of dynamic aspects and an expressive collection of history-sensitive pointcuts. The pointcuts addressable by our approach include those that permit triggering of code if the current history matches a nested word language [7, 8] — this includes cflow and regular event patterns.

Organization of this paper. After a discussion of related work, we present the core language in Section 3, including a definition of contextual equivalence and examples. In Section 4, we describe the LTS and our notion of bisimulation; this section also contains examples that illustrate the use of the bisimulation. In Section 5, we state the foundational properties that hold. Section 6 presents an extended example, implementing access control and type enforcement. In this extended abstract, we elide all proofs, referring the reader to the full version [28] for details.

2. Related Work

Core calculi for aspect-based languages have been explored in a variety of settings: e.g., [26, 13] are based on class-oriented calculi; in [14], a parametric description of a wide range of aspect languages, is based on the object calculus [1]; and [51] integrates aspect and object-oriented languages. Our calculus builds on descriptions of aspects in higher-order functional languages [18, 61].

[63] describes a denotational semantics for a calculus with dynamic join points, pointcut designators, and advice. Our focus is on operational reasoning and proof rules: we refer the reader to [36] for a comparison of the operational and denotational approaches to stateful higher-order languages.

[44] provide the semantics of dynamic join points by translating into a core functional language with simple matching features. Our approach complements this work by providing reasoning tools for a core functional language with aspects.

Formal static reasoning via type systems has been explored for functional [42] and object-oriented [27] aspect languages. Typing considerations are orthogonal to our primary focus, and we elide them to lighten the presentation.

Model-checking techniques have been explored to analyze the behavior of individual aspect programs [40, 57, 60]. Our paper is complementary to this research: we envision our study in this paper as providing formal foundations and support to compositional proof principles of use to model-checking tools for aspect programs. The utility of compositional methods in model-checking aspect programs is already suggested in [40].

There has also been research into facilitating reasoning by controlling obliviousness. For example, information flow methods have been used to create type systems that ensure that aspects do not affect the return value [16] — for some security applications, these superficially drastic sounding restrictions are appropriate. In this general spirit, albeit with less impact on obliviousness, the named primitive pointcuts of our calculus can be viewed as ways to control interference between aspects and between aspects and other code. Our primitive pointcuts are directly inspired by Open Modules [6] (see also [47]) and are a formal device to model some features of Aspect-Aware Interfaces [33]. There are two different views about where such names can originate: (a) as programming annotation, written by the programmer (a view arguably in tension with uninhibited obliviousness), or (b) a tool derived annotation, derived from an analysis of the context of the program. In this paper, we do not take a viewpoint on this debate; instead, we focus on the support to reasoning that is afforded by such annotations.

Broadly speaking, bisimulation approaches to higher-order languages fall into two main categories, depending on the kinds of tests that are permitted.

The first approach is usually termed applicative bisimulation. Some of the historical landmarks on this route are the initial definition of applicative bisimulation for lazy lambda calculus [4], the presentation using a labelled transition system [21] and a general method to show that applicative bisimulation is a congruence [24]. In this approach, two terms, say M_1, M_2, that agree on convergence behavior are tested for bisimilarity by providing them identical arguments and testing the resulting computation ($M_1 N$ and $M_2 N$) coinductively for bisimilarity. Applicative bisimulation tests terms only once. However, imperative features may require arguments to be tested multiple times — such extensions were developed by [29].

The second approach, often termed "contextual bisimulation", was initially introduced for higher-order process algebras [52]. [58] develops this approach for a language including existential types; [36] develops this general framework for a higher-order language with imperative features. Class equivalences [35], and the object calculus [34] are also tackled by these methods. In this style, two lambda terms, say M_1, M_2, are tested by providing them arguments that are derived from identical contexts (say $D[\cdot]$) with holes filled by bisimilar terms (say N_1, N_2) and testing the resulting computation ($M_1 D[N_1]$ and $M_2 D[N_2]$) coinductively for bisimilarity. The complexity and number of tests is controlled by restricting attention to value contexts, i.e., $D[\cdot]$ such that $D[N_1]$ and $D[N_2]$ are values.

Our approach is inspired by open bisimulation [54], and ENF-bisimulation [38, 39]. In comparison to applicative bisimulation, the more elementary congruence proofs of our approach suggest that our open-bisimulation based approach addresses stateful features more directly. In contrast to contextual approaches, our methods do not need to address the contextual closure of programs and equivalences of values in this closure. However, the price paid by our approach is the explicit maintenance of extra contexts and transitions for book-keeping mechanisms. We develop congruence results and bisimulation-upto results to lighten this burden. In the following technical sections, we present a detailed comparison of our definitions with the two approaches.

In summary, the examples in the paper suggest that our treatment is good enough to capture and formalize intuitions crystallized by observation of the source code. However, we do not have any results that support the (semi-)automatic derivation of witnessing relations. That investigation remains open to future study.

3. Language

Our calculus builds on descriptions of aspects in higher-order functional languages [18, 61]. Advice may be loaded dynamically; several recent aspect language implementations support such dynamic aspects, eg, [11]. Primitive pointcuts are named and scoped: a programmer may limit the scope over which a function is advisable by controlling the scope of the associated primitive pointcut. In this respect, our language has some of the expressiveness of the module language of [6], in a simpler setting. Each function declaration is associated with a primitive pointcut and advice applies to a function only if its associated primitive pointcut is that of the function. One may view possession of the name of a function as a form of *read access* and possession of the primitive pointcut of a function as a form of *write access*. We formalize this intuition when encoding references in Example 6.

The language is an untyped lambda calculus extended with function declarations in the style of ML and with advice over declared functions. The difference between abstractions and declared functions can be detected contextually. For example, consider $\lambda\_.0$ and $\textbf{fun}\ \mathsf{f@p} = \lambda\_.0\,;\,\mathsf{f}$, which declares f at primitive pointcut p and returns f. The first expression results immediately in an abstraction. The second results in the name f, which is only resolved to an abstraction when applied. The difference is observable when the primitive pointcut p is used to declare advice, as, for example, in the context $\textbf{adv}\ \mathsf{p} = \lambda\_.1\,;\,[\text{–}]\,()$; here $[\text{–}]$ is the "hole" to be filled by a term. The context declares advice at p then applies the hole to the unit value; evaluation results in 0 when the hole is filled with $\lambda\_.0$, but 1 when filled with $\textbf{fun}\ \mathsf{f@p} = \lambda\_.0\,;\,\mathsf{f}$. A function declared at a bound primitive pointcut is unadvisable outside the scope of the binder; thus, $\lambda\_.0$ and $\textbf{pcd}\ \mathsf{p}\,;\,\textbf{fun}\ \mathsf{f@p} = \lambda\_.0\,;\,\mathsf{f}$ are contextually indistinguishable.

In the rest of this section, we formalize the syntax (Section 3.1) and dynamics (Sections 3.2 and 3.3) of this core calculus. Section 3.4 defines contextual equivalence. Section 3.5 provides simple examples to illustrate the definitions. Section 3.6 discusses *open modules* and temporal pointcuts.

3.1 Syntax

We divide names into two countably infinite and mutually disjoint sets: variables and primitive pointcuts. In this study, primitive pointcuts are second-class entities; we discuss the motivation for this decision in Example 10.

Syntax

$f, g, h, x, y, z, \phi, \psi, \theta$	Variable Names
p, q, r	Primitive Pointcut Descriptors
$A, B\ ::=$	Declarations
$\mathsf{pcd}\ p$	Primitive Pointcut Descriptor ($dn = \{p\}$)
$\mathsf{fun}\ f\mathsf{@}p = U$	Function ($dn = \{f\}, f$ bound in U)
$\mathsf{adv}\ p = \lambda z.U$	Advice ($dn = \{\,\}$, z bound in U)
$U, V, W\ ::=$	Values
x	Variable
$\lambda x.M$	Abstraction (x bound in M)
$M, N, L\ ::=$	Terms
U	Value
$A\,;M$	Declaration ($dn(A)$ bound in M)
$\mathsf{let}\ x = M\,;N$	Sequence (x bound in N)
$U\ V$	Application

The name declared by a declaration is given by the function *dn*, defined in the syntax table above. We assume the usual notion of free names, recovered by the function *fn*. We identify terms up to renaming of bound names and write $M[x := U]$ for the capture-avoiding substitution of U for x in M. Thus $\mathsf{pcd}\ p\,;M$ is identical to $\mathsf{pcd}\ q\,;M[p := q]$ for any $q \notin fn(M)$.

We use the following discipline for variable names, when feasible. (The distinctions, while useful in many cases, are blurred when discussing congruence.)

- z is used for *proceed variables* bound in the body of advice;
- x-y are used for variables bound in abstractions and let-expressions, other than as a proceed variable;
- f-h are used for variables bound by function declarations;
- ϕ-θ are used for free function variables.

Variables x-y are resolved, in the standard way, during evaluation (Section 3.3). Variables z and f-h are resolved during function lookup (Section 3.2). The variables ϕ-θ are unresolvable; these are used in the LTS semantics (Section 4).

In examples, we use the unit value $()$, booleans, integers and pairs of values. These can be encoded in the standard way (where $()$ is any value). The extension of the equational theory to distinguish these types is unsurprising and requires additional bookkeeping. We also use other well-known combinators, such as the divergent term Ω and the fixpoint combinator fix.

We use syntax sugar for application in the style of Moggi [46]; for example, $M\ N \triangleq \mathsf{let}\ x = M\,;\,\mathsf{let}\ y = N\,;\,x\ y$. We adopt the same convention for operators on booleans, naturals and pairs. We write $\_$ for a bound variable that does not occur free in its scope; we abbreviate $\mathsf{let}\ \_ = M\,;N$ as $M\,;N$ and $\lambda\_.M$ as $\lambda.M$.

In examples, we sometimes write $\mathsf{fun}\ f = U$ as shorthand for $\mathsf{pcd}\ p\,;\,\mathsf{fun}\ f\mathsf{@}p = U$, when p is not of interest. We also occasionally write declarations as terms, with the meaning that A, as a term, abbreviates $A\,;\,()$.

3.2 Lookup

In this subsection, we describe function lookup, which determines the body of an advised function from a declaration sequence. We write $\vec{A}$ for *declaration sequences*, with "$\cdot$" representing the empty sequence, and "$;$" the element separator. An *evaluation configuration* is a pair of a declaration sequence and a term, written $\vec{A}/M$.

Example 1. Let $\vec{\mathsf{A}}$ be defined as follows.

$$\begin{array}{lll} \vec{\mathsf{A}} = \textbf{adv}\ \mathsf{p} = \lambda \mathsf{z}.\mathsf{V}\,; & & \mathsf{V} = \lambda \mathsf{y}.(\mathsf{z}\ \mathsf{y}) + 1 \\ \quad\ \ \textbf{fun}\ \mathsf{f@p} = \mathsf{W}\,; & \text{where} & \mathsf{W} = \lambda.5 \\ \quad\ \ \textbf{adv}\ \mathsf{p} = \lambda \mathsf{z}.\mathsf{U} & & \mathsf{U} = \lambda \mathsf{x}.(\mathsf{z}\ \mathsf{x}) * 3. \end{array}$$

When one looks up f in the context of $\vec{\mathsf{A}}$, the result is

$$\vec{\mathsf{A}}(\mathsf{f}) = \mathsf{U}\big[\mathsf{z} := \mathsf{V}[\mathsf{z} := \mathsf{W}]\big] = \lambda \mathsf{x}.((\lambda \mathsf{y}.((\lambda.5)\ \mathsf{y}) + 1)\ \mathsf{x}) * 3.$$

The top-level term is U: the last (or most recently) declared advice which effects f (via the primitive pointcut p). The proceed variable z of U is bound to the rest of the advice which effects f, in this case V. Substitutions layer in this way to the last piece of advice, which proceeds to the function body, in this case W.

Evaluation of $\mathsf{f}\,()$ proceeds as follows:

$$\begin{array}{rl} \cdot/\vec{\mathsf{A}}\,;\mathsf{f}\,() & \twoheadrightarrow \vec{\mathsf{A}}/((\lambda \mathsf{y}.((\lambda.5)\ \mathsf{y}) + 1)\,())\,*3 \\ & \twoheadrightarrow \vec{\mathsf{A}}/(((\lambda.5)\,()) + 1) * 3 \\ & \twoheadrightarrow \vec{\mathsf{A}}/18. \end{array}$$

Lookup is a partial function on names. For example, using the declarations above, $\vec{\mathsf{A}}(\mathsf{g})$ is undefined, and thus the evaluation configuration $\vec{\mathsf{A}}/\mathsf{g}\,()$ is stuck. □

Example 2. Note that advice may ignore the definition of the underlying function or of other advice — both referenced via z. As an example, consider

$$\begin{array}{lll} \vec{\mathsf{B}} = \textbf{adv}\ \mathsf{p} = \lambda \mathsf{z}.\mathsf{V}\,; & & \mathsf{V} = \lambda.7 \\ \quad\ \ \textbf{fun}\ \mathsf{f@p} = \mathsf{W}\,; & \text{where} & \mathsf{W} = \lambda.5 \\ \quad\ \ \textbf{adv}\ \mathsf{p} = \lambda \mathsf{z}.\mathsf{U} & & \mathsf{U} = \lambda \mathsf{x}.(\mathsf{z}\ \mathsf{x}) * 3. \end{array}$$

In this case

$$\vec{\mathsf{B}}(\mathsf{f}) = \mathsf{U}\big[\mathsf{z} := \mathsf{V}[\mathsf{z} := \mathsf{W}]\big] = \lambda\mathsf{x}.((\lambda.\mathsf{7})\ \mathsf{x}) * \mathsf{3}$$

and evaluation of $\mathsf{f}\,()$ proceeds as follows.

$$\cdot/\vec{\mathsf{B}}\,;\mathsf{f}\,() \twoheadrightarrow \vec{\mathsf{B}}/((\lambda.\mathsf{7})\,()) * \mathsf{3} \twoheadrightarrow \vec{\mathsf{B}}/\mathsf{21} \qquad \square$$

Lookup is defined using two auxiliary functions: *body* and *advise*. Whereas we identify terms up to renaming of bound names, the same does not hold for names declared in a declaration sequence. Instead, we require that declaration sequences be *well formed*, ie, that each name is declared at most once. (This treatment is motivated by the definition of *body*, by which a primitive pointcut may escape its scope.)

Definition 3 (Well formedness). A declaration sequence "$\vec{A}\,;B$" is *well formed* if $dn(B)$ does not occur in $\vec{A}$ and $\vec{A}$ is well formed. The empty sequence is also well formed.

An evaluation configuration $\vec{A}/M$ is *well formed* if $\vec{A}$ is well formed. □

Note that in a well-formed evaluation configuration $\vec{A}/M$, there may be names that occur free in M that are not declared in $\vec{A}$ (cf. Example 1).

The partial function $body(f, \vec{A})$ is defined whenever f is declared in $\vec{A}$; when defined, *body* returns both the value of the function and the primitive pointcut at which f is declared in $\vec{A}$.

$$\begin{aligned}
body(f, \cdot) &\triangleq \text{undefined}\\
body(f, \mathsf{pcd}\cdots;\vec{A}) &\triangleq body(f, \vec{A})\\
body(f, \mathsf{fun}\ f@p = U\,;\vec{A}) &\triangleq \langle p, U\rangle\\
body(f, \mathsf{fun}\ g@p = U\,;\vec{A}) &\triangleq body(f, \vec{A}),\ \text{where } f \neq g\\
body(f, \mathsf{adv}\cdots;\vec{A}) &\triangleq body(f, \vec{A})
\end{aligned}$$

The total function $advise(p, U, \vec{A})$ returns a value that applies to U the advice declared in $\vec{A}$ for p.

$$\begin{aligned}
advise(p, U, \cdot) &\triangleq U\\
advise(p, U, \mathsf{pcd}\cdots;\vec{A}) &\triangleq advise(p, U, \vec{A})\\
advise(p, U, \mathsf{fun}\cdots;\vec{A}) &\triangleq advise(p, U, \vec{A})\\
advise(p, U, \mathsf{adv}\ p = \lambda z.V\,;\vec{A}) &\triangleq advise(p, V[z := U], \vec{A})\\
advise(p, U, \mathsf{adv}\ q = \lambda z.V\,;\vec{A}) &\triangleq advise(p, U, \vec{A}),\ \text{where } p \neq q
\end{aligned}$$

Finally, the partial function $\vec{A}(f)$ is defined as follows.

$$\vec{A}(f) \triangleq \begin{cases} advise(p, V, \vec{A}) & \text{if } body(f, \vec{A}) = \langle p, V\rangle\\ \text{undefined} & \text{otherwise}\end{cases}$$

3.3 Dynamics

Evaluation is defined inductively as a binary relation between well formed configurations, using four axiom schemas. Following [19], the definition uses contexts.

Evaluation $(\vec{A}/M \rightarrow \vec{A}'/M')$

$$\begin{array}{lll}
\mathscr{E},\mathscr{F},\mathscr{G} ::= [-] \mid \mathsf{let}\ x = \mathscr{E}\,;N & & \text{Evaluation Contexts}\\
\vec{A}/\mathscr{E}[B\,;M] & \rightarrow \vec{A}\,;B/\mathscr{E}[M] & \text{if } dn(B) \notin dn(\vec{A}) \cup fn(\mathscr{E})\\
\vec{A}/\mathscr{E}[\mathsf{let}\ x = U\,;N] & \rightarrow \vec{A}/\mathscr{E}\big[N[x := U]\big] & \\
\vec{A}/\mathscr{E}[f\ V] & \rightarrow \vec{A}/\mathscr{E}\big[U\ V\big] & \text{if } \vec{A}(f) = U\\
\vec{A}/\mathscr{E}[(\lambda x.M)\ V] & \rightarrow \vec{A}/\mathscr{E}\big[M[x := V]\big] &
\end{array}$$

The first axiom is structural, regulating the scope of declarations. Recall that we allow renaming of bound variables in terms, but not declaration sequences. Since the set of names is infinite, evaluation configurations of the form $\vec{A}/\mathscr{E}[B\,;M]$ may always reduce, fixing a "fresh" name for $dn(B)$.

The axiom for sequencing is standard, reducing $\mathsf{let}\ x = M\,;N$ only when M is a value.

There are three possibilities for an application $\vec{A}/\mathscr{E}[U\ V]$: (1) If U is a function name f and $\vec{A}(f)$ is defined, then evaluation proceeds to $\vec{A}/\mathscr{E}[\vec{A}(f)\ V]$. (2) If U is an abstraction then evaluation proceeds call-by-value using U; this is the standard beta-reduction axiom. (3) Otherwise evaluation is stuck.

Write $\twoheadrightarrow$ for the reflexive transitive closure of $\rightarrow$.

Example 4. Consider the following evaluation configuration:

$$\cdot/\mathbf{fun}\ \mathsf{id@p} = \lambda\mathsf{x.x}\,;\mathbf{adv}\ \mathsf{p} = \lambda\mathsf{z}.\lambda\mathsf{y.z\ z\ y}\,;(\lambda\mathsf{f.f\ 5})\ \mathsf{id}.$$

Using the axiom for declarations twice and the axiom for application once, this reduces to

$$\mathbf{fun}\ \mathsf{id@p} = \lambda\mathsf{x.x}\,;\mathbf{adv}\ \mathsf{p} = \lambda\mathsf{z}.\lambda\mathsf{y.z\ z\ y}/\mathsf{id\ 5}.$$

Note that id is treated as a pure name when passed as an argument; it is only resolved at the point of application, where the axioms for lookup and beta-reduction yield

$$\begin{aligned}
&\mathbf{fun}\ \mathsf{id@p} = \lambda\mathsf{x.x}\,;\mathbf{adv}\ \mathsf{p} = \lambda\mathsf{z}.\lambda\mathsf{y.z\ z\ y}/(\lambda\mathsf{y}.(\lambda\mathsf{x.x})\ (\lambda\mathsf{x.x})\ \mathsf{y})\ \mathsf{5}\\
\twoheadrightarrow\ &\mathbf{fun}\ \mathsf{id@p} = \lambda\mathsf{x.x}\,;\mathbf{adv}\ \mathsf{p} = \lambda\mathsf{z}.\lambda\mathsf{y.z\ z\ y}/\mathsf{5}. \qquad \square
\end{aligned}$$

3.4 Contextual Equivalence

Contextual equivalence is defined with respect to a primitive notion of observation; two terms are related if they yield the same observations in all contexts. Following [17, 30], we assume a distinguished function name and take a call to this function to be a primitive observation.

Definition 5. A *(general) context* is any term with a single hole:

$$\mathscr{C} ::= [-] \mid A\,;\mathscr{C} \mid \mathsf{let}\ x = \mathscr{C}\,;N \mid \mathsf{let}\ x = M\,;\mathscr{C}.$$

Write M↯ if $M \twoheadrightarrow \mathscr{E}[\mathsf{signal}\ U]$ for some evaluation context $\mathscr{E}$ and value U. For terms M and N in which signal does not occur, define $M \lesssim N$ if for every context $\mathscr{C}$, $\mathscr{C}[M]$↯ implies $\mathscr{C}[N]$↯. Two terms M and N are *contextually equivalent* ($M \equiv N$) if $M \lesssim N$ and $N \lesssim M$. □

As a simple example, consider $(\mathbf{adv}\ \mathsf{p} = \lambda.\lambda.\mathsf{1})\,;\mathsf{f}\,()\,;\Omega$ and $(\mathbf{adv}\ \mathsf{p} = \lambda.\lambda.\mathsf{2})\,;\mathsf{f}\,()\,;\Omega$. In our setting, these can be distinguished by the context

$$(\mathbf{fun}\ \mathsf{g@p} = \mathsf{0})\,;(\mathbf{fun}\ \mathsf{f} = \mathbf{if}\ \mathsf{g}\,() = \mathsf{1}\ \mathbf{then}\ \mathsf{signal}()\ \mathbf{else}\ \Omega)\,;[-].$$

3.5 Simple Examples

Example 6 (References). We show how to code ML-style references as syntax sugar in the language of terms. The example demonstrates the well-known fact that dynamically loaded advice is a form of mutability.

We model references as a pair of functions, where the first is used for reading and the second for writing; the first is locally advisable, whereas the second is not.

$$\begin{aligned}
\mathsf{ref}\ U &\triangleq \mathbf{pcd}\ \mathsf{p}\,;(\mathbf{fun}\ \mathsf{f@p} = \lambda.U)\,;(\mathsf{f}, \lambda\mathsf{x}.\mathbf{adv}\ \mathsf{p} = \lambda.\lambda.\mathsf{x})\\
!\,U &\triangleq (\mathsf{fst}\ U)\,()\\
U := V &\triangleq (\mathsf{snd}\ U)\ V;\,()
\end{aligned}$$

We can code the imperative factorial as

$$\begin{aligned}
&\mathbf{fun}\ \mathsf{fac} = (\lambda\mathsf{x}.(\mathbf{let}\ \mathsf{y} = \mathsf{ref\ 1})\,;(\mathbf{fun}\ \mathsf{loop} = \mathsf{U})\,;\mathsf{loop\ x}),\ \text{where}\\
&\mathsf{U} = \lambda\mathsf{x}.\mathbf{if}\ (\mathsf{x} \leq \mathsf{1})\ \mathbf{then}\ (!\mathsf{y})\ \mathbf{else}\ (\mathsf{y} := !\mathsf{y} * \mathsf{x};\ \mathsf{loop}\ (\mathsf{x} - \mathsf{1})).
\end{aligned}$$

Eliding the definitions of fac, loop, and p, $\mathsf{fac\ 2}$ evaluates as

$$\begin{aligned}
\cdot/\mathsf{fac\ 2} &\twoheadrightarrow \mathbf{fun}\ \mathsf{f@p} = \lambda.\mathsf{1}/\mathsf{loop\ 2}\\
&\twoheadrightarrow \mathbf{fun}\ \mathsf{f@p} = \lambda.\mathsf{1}\,;\mathbf{adv}\ \mathsf{p} = \lambda.\mathsf{2}/\mathsf{loop\ 1}\\
&\twoheadrightarrow \mathbf{fun}\ \mathsf{f@p} = \lambda.\mathsf{1}\,;\mathbf{adv}\ \mathsf{p} = \lambda.\mathsf{2}/\mathsf{f}\,()\\
&\twoheadrightarrow \mathbf{fun}\ \mathsf{f@p} = \lambda.\mathsf{1}\,;\mathbf{adv}\ \mathsf{p} = \lambda.\mathsf{2}/\mathsf{2}.
\end{aligned}$$

Garbage collecting the declarations, the result is 2, as expected. □

Example 7 (Contexts may need to test a value more than once). It is important to note that contexts may store values and test them more than once. For example, the terms

$$\lambda.\mathsf{0} \quad \text{and} \quad \mathbf{let}\ \mathsf{b} = \mathsf{ref\ tru}\,;(\lambda.\mathbf{if}\ !\mathsf{b}\ \mathbf{then}\ \mathsf{b} := \mathsf{fls};\,\mathsf{0}\ \mathbf{else}\ \mathsf{1})$$

can be distinguished by the context

$$\textbf{let}\ \mathsf{x} = [-]\ ;\ \mathsf{x}()\ ;\ \textbf{if}\ \mathsf{x}() = 0\ \textbf{then}\ \mathsf{signal}()\ \textbf{else}\ \Omega. \qquad \Box$$

Example 8 (Contexts can observe advice order). To show some of the subtleties of contextual reasoning, here is an example where a context inserts itself in the middle of an advice list.

$$\mathscr{E} = \textbf{fun}\ \mathsf{f@p} = \mathsf{W}\ ;\ \textbf{let}\ \mathsf{x} = [-]\ ;\ \textbf{adv}\ \mathsf{p} = \lambda \mathsf{z}.\mathsf{V}\ ;\ \mathsf{x}()$$

Consider

$$\mathscr{E}[\textbf{adv}\ \mathsf{p} = \lambda \mathsf{z}.\mathsf{U}_1\ ;\ (\lambda.\textbf{adv}\ \mathsf{p} = \lambda \mathsf{z}.\mathsf{U}_2\ ;\ \mathsf{f}\ 0)]$$

which evaluates to

$$\cdots\ ;\ \mathsf{U}_2\big[\mathsf{z} := \mathsf{V}[\mathsf{z} := \mathsf{U}_1[\mathsf{z} := \mathsf{W}]]\big].$$

Here the context has inserted the advice V between two bits of user advice U_2 and U_1. Using $\mathsf{V} = \lambda \mathsf{x}.\textbf{if}\ \mathsf{x} = 1\ \textbf{then}\ \mathsf{signal}()\ \textbf{else}\ \Omega$, the context can distinguish the following pairs of advice from the term; however, this difference cannot be detected simply by running f without using V.

$$\begin{array}{ll} \mathsf{U}_1 = \lambda \mathsf{x}.\mathsf{z}\ (\mathsf{x}+2) & \mathsf{U}'_1 = \lambda \mathsf{x}.\mathsf{z}\ (\mathsf{x}+1) \\ \mathsf{U}_2 = \lambda \mathsf{x}.\mathsf{z}\ (\mathsf{x}+1) & \mathsf{U}'_2 = \lambda \mathsf{x}.\mathsf{z}\ (\mathsf{x}+2) \qquad \Box \end{array}$$

Example 9 (Indistinguishability of functions). Functions with the same body declared at the same primitive pointcut are indistinguishable. The following terms are contextually equivalent for any M.

$$\begin{array}{l} \textbf{fun}\ \mathsf{f@p} = \lambda \mathsf{x}.M;\ \textbf{fun}\ \mathsf{g@p} = \lambda \mathsf{x}.M;\ (\mathsf{f}, \mathsf{g}) \\ \textbf{fun}\ \mathsf{h@p} = \lambda \mathsf{x}.M;\ (\mathsf{h}, \mathsf{h}) \qquad \Box \end{array}$$

3.6 Open Modules and Temporal Pointcuts

In this subsection, we consider encodings of *open modules*, as proposed by Aldrich [6]. Open modules extend ML-style modules to support two methods for controlling aspects:

- a distinction between internal and external function calls — only external calls are advisable from outside the module; and
- explicit pointcut declaration in module interfaces — only declared pointcuts may be used externally.

The first feature is handled in the operational semantics of [6] by renaming the function and creating a fresh declaration of the original name to invoke it. This kind of renaming can be achieved in compilation; here, we write programs directly in the form such a compiler would produce.

The second feature is more subtle, and we address it in two ways.

- We provide distinct binders for functions and primitive pointcuts; these may be viewed respectively as read and write capabilities, which may be handled independently. We treat primitive pointcuts as second class, since they are intended to delimit the static scope of mutability.
- We allow dynamically loaded advice. In addition to encoding state (discussed in the previous example), dynamically loaded advice allows us to create expressive "pointcuts" and to communicate them selectively (as abstractions) — Examples 12, 13.

Example 10 (Open Modules). To get a sense of our approach, consider a concrete example: a math module with one advisable function fac. Internal and external calls to fac are distinguished so that only external calls may be advised.

```
module type MATH = sig
  val fac : int → int
  pointcut pfac : int → int
end;;
module Math : MATH = struct
  let rec fac = fun n → if n<1 then 1 else n*fac(n-1)
  pointcut pfac = call(fac)
end;;
open Math;;
let main = fun _→ fac 5;;
```

We view the module as providing two functions: the first is fac itself; the second is the pointcut pfac. A call to pfac will place advice on external calls to fac. In a module system, the calls to pfac occur in the compiler, rather than at runtime, but this phase distinction is an implementation convenience rather than a necessity.

The example can be coded in our language as follows.

```
fun Math = λ.
  fun fac′ = λn.if n<1 then 1 else n*fac′(n-1);
  pcd pfac′;
  fun fac@pfac′ = fac′;
  (fac, λy.adv pfac′ = λz.λx.y z x);
let (fac, pfac) = Math ();
fun main = λ.fac 5
```

The functions fac and pfac, recovered from Math, correspond exactly to the functions provided by the module above. For example, to count the number of calls to fac, one might call in main:

```
let c = ref 0;
pfac (λz.λx.c := !c+1; z x)
```

□

Remark 11 (Modularity results). In the above example, whereas fac is publicly advisable, fac′ is private to Math. To see that internal calls to fac′ are unadvisable, note that one could exchange the body of fac′ given here with that from Example 6 and the result would be contextually equivalent to the original. In fact the following general result holds. Let

$$\begin{array}{l} \mathscr{C} = \textbf{pcd}\ \mathsf{p};\ \textbf{fun}\ \mathsf{f@p} = [-];\ \mathsf{f} \\ \mathscr{D} = \textbf{fun}\ \mathsf{g@q} = [-]. \end{array}$$

Then,

$$\mathscr{C}[\mathsf{U}] \equiv \mathscr{C}[\mathsf{V}] \text{ implies } \mathscr{D}\big[\mathscr{C}[\mathsf{U}]\big] \equiv \mathscr{D}\big[\mathscr{C}[\mathsf{V}]\big].$$

This follows immediately from the fact that $\equiv$ is a congruence (section 5). This general result allows any function to be defined in such a way that external calls are advisable, while internal ones are not. The remarkable power of contextual reasoning guarantees that the internal body can be substituted with any locally equivalent body without effecting the overall observable behavior. □

The previous encoding can be extended to richer pointcut languages, while still maintaining the modularity results[1].

Example 12 (cflow). The AspectJ pointcut call(f) && cflow(g) detects calls to f in the context of a call to g. Such a pointcut is exported from the following module.

```
fun FcflowG = λ.
  pcd pf; fun f@pf = ··· ;
  pcd pg; fun g@pg = ··· ;
  let b = ref fls; // call to g active
  adv pg = λz.λx.let b′ = !b; b := tru; let y = z x; b := b′; y;
  (f, g, λy.adv pf = λz.λx.if !b then y z x else z x);
let (f,g,pf_cflow_g) = FcflowG ();
```

The local boolean reference b is used to record whether a call to g is active. Whenever g is called, the advice at pg sets b to tru,

[1] **A comment on modelling subclassing.** Enrich pointcuts with a preorder. If one takes $p \leq q$ to mean that advice placed on q applies equally for p, then correct behavior with respect to subclassing is achieved by ensuring that overriding methods are defined at smaller roles.

proceeds to the body of g, then resets b. Whenever f is called the advice at pf first checks b before proceeding to the body of f.

A user may advise "f in the context of g", by calling the pf_cflow_g with advice λz.λx.⋯. However, no other pointcuts are exposed. This generalizes the technique of Aldrich, and indeed the congruence results (c.f. Remark 11) apply equally to such terms. □

Nested word languages [8, 7] are a subset of context free languages with good closure properties that capture sensitivity to both the call-stack (as in cflow) and other history (as in regular patterns [9]). Pointcuts based on nested word languages arise naturally in examples in security (access control) and document processing (XML transducers). Since the operational semantics of nested word languages pushes exactly one stack symbol upon reading a call symbol and pops exactly one stack symbol upon reading a return symbol, such pointcut languages are addressable by implementation methods developed for cflow and regular patterns. The next example illustrates the ingredients of a systematic translation from temporal pointcuts specified via nested-word languages.

Example 13 (History-sensitive access control). Abadi and Fournet [2] argue for history-sensitive access control mechanisms more expressive than the stack inspection mechanisms found in Java and C#. For example, consider a policy stating that advice on a sensitive function rm (e.g., for file deletion) should be executed only if an (untrusted) function un has never been invoked in the past, *and* no call to f is still active. This policy for an access control failure is specified as a nested word language over symbols drawn from calls to, and returns from, un, rm and f. Using EBNF syntax:

$$\begin{array}{rcl} \mathsf{balanced} & ::= & \big((\mathit{call}(.)\ \mathsf{balanced}\ \mathit{ret}(.))\big)* \\ \mathsf{opencalls} & ::= & \big(\mathsf{balanced} \mid \mathit{call}(.)\big)* \end{array}$$

The specified property can then be written as:

$$\big((\underbrace{.*\ \mathit{call}(\mathsf{un})\ \ .*}_{\mathsf{un}\text{ called}}) \mid (\underbrace{\mathsf{opencalls}\ \mathit{call}(\mathsf{f})\ \mathsf{opencalls}}_{\mathit{call}(\mathsf{f})\text{ active}})\big)\ \mathit{call}(\mathsf{rm}).$$

Following Example 12, we can export a pointcut matching the negation of this property of the call history.

```
fun Hsac = λ.
  pcd pun ; fun un@pun = ⋯ ;
  pcd pf ; fun f@pf = ⋯ ;
  pcd prm ; fun rm@prm = ⋯ ;
  let b₁ = ref fls ; // call to f active
  adv pf = λz.λx.let b′ = ! b₁ ; b₁ := tru ; let y = z x ; b₁ := b′ ; y ;
  let b₂ = ref fls ; // call un occurred
  adv pun = λz.λx.b₂ := tru ; z x ;
  (f, un, rm, λy.adv prm = λz.λx.if ! b₁ or ! b₂ then z x else y z x) ;
let (f, un, rm, pf_hsac) = Hsac () ;
```

Advice attached using pf_hsac applies only in the specified conditions, and no other pointcuts are exposed. The congruence results (c.f. Remark 11) apply equally to such terms. □

4. Labeled Transition System and Bisimulation

In this section, we present the bisimulation relation following the LTS style pioneered by Gordon [21, 22], in particular in the style of presentation of Jeffrey and Rathke for Concurrent ML [29]. In contrast to this prior work, our intuitions are guided by open bisimulation and address aspect features. The technical consequence of this difference is that our proof that bisimulation is a congruence is a direct proof based on a direct analysis of substitutions rather than following these papers in being based on Sangiorgi [53] or Howe [24].

The rest of this section is organized as follows. In Section 4.1, we describe the ideas of our LTS for the restricted case of the pure untyped lambda calculus without aspects or declarations. This treatment of a familiar calculus is intended to motivate the LTS use of symbolic functions and advice that are defined by the environment and provide core intuitions for the following subsections. In Section 4.2 we adapt the operational semantics of earlier sections to deal with symbolic data such as functions and advice. In the Section 4.3, we provide a description of the LTS for the full calculus, and follow with a definition of the bisimulation relation in Section 4.4. Section 4.5 makes the intuitions of our model concrete by a series of examples.

4.1 An introduction to open bisimulation

In this subsection, we provide an snapshot of our approach by briefly describing an LTS for the pure untyped call-by-value lambda calculus.

We briefly recall the LTS approach [21] to applicative bisimulation for the pure untyped call-by-value lambda calculus

- A non-value term M has a τ transition to M' if M reduces in one step to M'.
- A value U (eg. $\lambda x.M$) has a transition labeled U' to the application $U\ U'$.

Two terms are bisimilar if the associated transition systems are bisimilar, i.e., if their convergence properties agree and each applicative test yields bisimilar terms.

Our approach is inspired by open bisimulation [54], and ENF-bisimulation [38, 39]. (The reader can view this subsection, in isolation, as a presentation of ENF-bisimulation-upto-η using an LTS.) Following our conventions, we use ϕ and ψ for variables that occur free in terms.

- A non-value term M has a τ transition to M' if M reduces in one step to M'.
- Values U have transitions labeled ϕ (where ϕ is fresh) to the application $U\ \phi$ — applicative tests are carried out with fresh names.
- Terms can now be of the form $\mathscr{E}[\phi\ U]$, for some evaluation context $\mathscr{E}$, where ϕ is an uninterpreted symbol. These terms have additional transitions:
  - A transition labeled fcall ϕ to U
  - Transitions labeled ret ψ to $\mathscr{E}[\psi]$ for a fresh environment variable ψ.

Again, two terms are bisimilar if the associated transition systems are bisimilar. The second rule is crucial to enforce the idea (similar to ENF-bisimulation [38, 39]) that if the application $\phi\ U$ is bisimilar to the application $\psi\ V$ then $\phi = \psi$ and U is bisimilar to V.

4.2 Symbolic functions and symbolic advice

The LTS must allow functions and advice to be defined by the environment, influencing a term. To accommodate context functions, we need only extend our notion of well-formedness to allow occurrences of free variables representing these functions. As noted earlier, we use ϕ, ψ to indicate these free variables; we sometimes refer to these as *symbolic function names* because they are uninterpreted in the term.

To accommodate context advice, we assume a countably infinite set of *symbolic advice names*, α, β, disjoint from the sets of variable names and primitive pointcuts.

SYMBOLIC ADVICE

α, β Symbolic Advice Names

$A,B ::= \cdots \mid \mathsf{adv}\ p = \alpha$	Symbolic Advice Declaration
$U,V,W ::= \cdots \mid \alpha\texttt{<}U\texttt{>}$	Symbolic Advice Call

Note that if $A = \mathsf{fun}\ f\texttt{@}p = \phi$, then by our previous definition of lookup $\vec{A}(f) = \langle p, \phi \rangle$; thus no extensions are required to handle symbolic functions. For symbolic advice, we extend the definition of *advise* as follows.

$$\begin{aligned} &\mathit{advise}(p, U, \mathsf{adv}\ p = \alpha\,;\vec{A}) \triangleq \mathit{advise}(p, \alpha\texttt{<}U\texttt{>}, \vec{A}) \\ &\mathit{advise}(p, U, \mathsf{adv}\ q = \alpha\,;\vec{A}) \triangleq \mathit{advise}(p, U, \vec{A}),\ \text{where } p \neq q \end{aligned}$$

Example 14 (Evaluation with symbolic names). Let

$$\vec{\mathsf{A}} = \mathbf{pcd}\ \mathsf{p}\,;\mathbf{fun}\ \mathsf{f}\texttt{@}\mathsf{p} = \phi\,;\mathbf{adv}\ \mathsf{p} = \alpha\,;\mathbf{adv}\ \mathsf{p} = \lambda \mathsf{z}.\lambda \mathsf{x}.(\mathsf{z}\ \mathsf{x}) * 3.$$

Evaluation of $\mathsf{f}()$ proceeds as follows.

$$\begin{aligned} \vec{\mathsf{A}}/\mathsf{f}() &\twoheadrightarrow \vec{\mathsf{A}}/(\lambda \mathsf{x}.(\alpha\texttt{<}\mathsf{f}\texttt{>}\ \mathsf{x}) * 3)() \\ &\twoheadrightarrow \vec{\mathsf{A}}/(\alpha\texttt{<}\mathsf{f}\texttt{>}()) * 3 \end{aligned}$$

At this point, evaluation is stuck. Intuitively, control is given to the context that defined α. The LTS presented next will provide transitions which cover such cases, potentially exposing f. Note that if evaluation arrives at an application $\mathsf{f}()$, the result will be ϕ; again evaluation is stuck, this time giving the context control through the undefined body of ϕ. □

4.3 The LTS

For namespace management, we define a *symbol environment*, which binds all symbolic function and advice names, and an *symbol declaration*, which may declare primitive pointcuts, functions and advice.

LTS SYNTAX

$\mathbf{M},\mathbf{N} ::= \vec{A}/\vec{\mathscr{E}}/M/\vec{U}$	Configuration
$\Gamma ::= \cdot \mid \phi,\Gamma \mid \alpha,\Gamma$	Symbol Environment
$\Delta ::= \cdot \mid A,\Delta$	Symbol Declaration

$\varkappa ::= \tau \mid \kappa$	All Labels
$\kappa ::=$	Visible Labels
$\mathsf{fcall}\ \phi$	Term calls context function ϕ
$\mathsf{acall}\ \alpha$	Term calls context advice α
$\mathsf{ret}\ \phi$	Context returns to term with result ϕ ($dn = \{\phi\}$)
$\mathsf{app}\ \phi$	Context calls term with argument ϕ ($dn = \{\phi\}$)
put	Context saves value
$\mathsf{get}\ i$	Context restores value
$\mathsf{fun}\, f\texttt{@}p = \phi$	Context declares function ($dn = \{f, \phi\}$)
$\mathsf{adv}\ p = \alpha$	Context declares advice ($dn = \{\alpha\}$)

In a configuration $\vec{A}/\vec{\mathscr{E}}/M/\vec{U}$, we refer to M as the *active term.*

With respect to evaluation configurations, the new elements are the list of contexts $\vec{\mathscr{E}}$ and the list of values $\vec{U}$. The contexts $\vec{\mathscr{E}}$ model the call stack: it will be used in a manner consistent with the stack discipline. The list $\vec{U}$ includes all values that have been released/leaked to the environment during evaluation of the term. Thus, the values in $\vec{U}$ are available for the environment to inspect and use. Formally, $\vec{U}$ is a way to account for the imperative/state features of the calculus. These modelling ideas follow prior research [29, 30, 58, 36].

We define the LTS relative to a *symbol environment* Γ and *symbol declaration* Δ. In Section 4.4, we will define bisimilarity as $\Gamma;\Delta \vdash \mathbf{M} \sim \mathbf{N}$. The symbol environment is used to manage names in the LTS, in particular to ensure two bisimilar terms may always make transitions with the same labels. The symbol declaration, likewise, ensures that both contexts in a bisimulation have the same observation power. (We describe how to derive an initial state from a term in Definition 17.)

The target symbol environment/declaration of a transition is determined by the source symbol environment/declaration and the label of the transition.

Definition 15 (LTS state). In a configuration $\vec{A}/\vec{\mathscr{E}}/M/\vec{U}$, $dn(\vec{A})$ are bound in $\vec{\mathscr{E}}/M/\vec{U}$. (The let binders in $\vec{\mathscr{E}}$ are not in scope in M or $\vec{U}$ and thus are not binding.)

A *state* of the LTS is a triple $\Gamma;\Delta \vdash \mathbf{M}$, where the names listed in Γ are bound in Δ and $\mathbf{M}$ and $dn(\Delta)$ are bound in $\mathbf{M}$. A state is *well formed* if no name occurs free, and no name is declared more than once in $\Gamma, \Delta, \vec{A}$. □

By way of contrast with evaluation configurations, note that we require a well formed LTS state to be closed. In the sequel, we assume that all LTS states are well-formed.

LTS

$$\begin{aligned}
&\Gamma;\Delta \vdash \vec{A}/\vec{\mathscr{E}}/M/\vec{U} \xrightarrow{\tau} \Gamma;\Delta \vdash \vec{B}/\vec{\mathscr{E}}/N/\vec{U} \quad \text{if } \Delta,\vec{A}/M \rightarrow \Delta,\vec{B}/N \\
&\Gamma;\Delta \vdash \vec{A}/\vec{\mathscr{E}}/\mathscr{F}[\phi\ V]/\vec{U} \xrightarrow{\mathsf{fcall}\ \phi} \Gamma;\Delta \vdash \vec{A}/\vec{\mathscr{E}},\mathscr{F}/V/\vec{U} \quad \text{if } \phi \in \Gamma \\
&\Gamma;\Delta \vdash \vec{A}/\vec{\mathscr{E}}/\mathscr{F}[\alpha\texttt{<}V\texttt{>}\ W]/\vec{U} \xrightarrow{\mathsf{acall}\ \alpha} \Gamma;\Delta \vdash \vec{A}/\vec{\mathscr{E}},\mathscr{F}/W/\vec{U},V \\
&\qquad \text{if } \alpha \in \Gamma \\
&\Gamma;\Delta \vdash \vec{A}/\vec{\mathscr{E}},\mathscr{F}/V/\vec{U} \xrightarrow{\mathsf{ret}\ \phi} \Gamma,\phi;\Delta \vdash \vec{A}/\vec{\mathscr{E}}/\mathscr{F}[\phi]/\vec{U} \\
&\Gamma;\Delta \vdash \vec{A}/\vec{\mathscr{E}}/V/\vec{U} \xrightarrow{\mathsf{app}\ \phi} \Gamma,\phi;\Delta \vdash \vec{A}/\vec{\mathscr{E}}/V\ \phi/\vec{U} \\
&\Gamma;\Delta \vdash \vec{A}/\vec{\mathscr{E}}/V/\vec{U} \xrightarrow{\mathsf{put}} \Gamma;\Delta \vdash \vec{A}/\vec{\mathscr{E}}/V/\vec{U},V \\
&\Gamma;\Delta \vdash \vec{A}/\vec{\mathscr{E}}/V/\vec{U} \xrightarrow{\mathsf{get}\ i} \Gamma;\Delta \vdash \vec{A}/\vec{\mathscr{E}}/U_i/\vec{U} \quad \text{if } 1 \leq i \leq |\vec{U}| \\
&\Gamma;\Delta \vdash \vec{A}/\vec{\mathscr{E}}/V/\vec{U} \xrightarrow{\mathsf{fun}\, f\texttt{@}p=\phi} \Gamma,\phi;\Delta,\mathsf{fun}\ f\texttt{@}p = \phi \vdash \vec{A}/\vec{\mathscr{E}}/V/\vec{U},f \\
&\qquad \text{if } p \in \Gamma \cup dn(\Delta) \\
&\Gamma;\Delta \vdash \vec{A}/\vec{\mathscr{E}}/V/\vec{U} \xrightarrow{\mathsf{adv}\ p=\alpha} \Gamma,\alpha;\Delta \vdash \vec{A}\,;\mathsf{adv}\ p = \alpha/\vec{\mathscr{E}}/V/\vec{U} \\
&\qquad \text{if } p \in \Gamma \cup dn(\Delta)
\end{aligned}$$

The fact that configurations must be well-formed ensures that, in the rules for ret and app transitions, the name ϕ must be fresh (i.e., must not occur in $\Gamma \cup dn(\Delta) \cup dn(\vec{A})$; likewise for the names ϕ and f in the rule for fun and α in the rule for adv.

Call-By-Value invariant. The LTS rules enforce a call-by-value invariant. This is seen by noting that precedence is afforded to internal reductions of the term. So, all rules except the first three are applicable to state $\Gamma;\Delta \vdash \vec{A}/\vec{\mathscr{E}}/M/\vec{U}$ only if M is a value.

Applicative tests. $\mathsf{app}\ \phi$ performs applicative tests. Rather than providing a term as an argument for the applicative test, this rule provides a fresh symbolic argument ϕ.

Stack of evaluation contexts. In the pure lambda calculus setting of Section 4.1, the rules for fcall and ret reflect the absence of interference between the caller and the callee in a purely functional language — the testing of the evaluation context and the callee argument is done separately. Thus, there was no need to track the evaluation context in the LTS for the pure lambda calculus.

In contrast, the LTS for the full calculus has to permit the environment an opportunity to inspect the arguments before the term continues evaluation — this is meaningful for the full calculus because of state changes caused by the dynamic laying down of advice. This is done in our LTS by the use of the stack of evaluation contexts $\vec{\mathscr{E}}$.

$\mathsf{fcall}\ \phi$ pushes the current evaluation context into $\vec{\mathscr{E}}$. The active term becomes the argument to the call, V, $\mathsf{ret}\ \phi$ returns a symbolic value ϕ to the top evaluation frame, $\mathscr{F}$, of the stack $\vec{\mathscr{E}},\mathscr{F}$ and moves it into the current-term position, popping $\mathscr{F}$ from the top of the stack. (This stack discipline would have to be liberalized to address a language with control operators.)

Note that calls to signal (from Section 3.4) are treated like any other call, and thus generate labels of the form $\mathsf{fcall\ signal}$.

Symbolic advice tests. In the rule for acall, since environment advice is invoked with the arguments V, they are added to the list $\vec{U}$ of values that are available for the environment to inspect and use. As in the case for fcall, the active term is changed to the argument, in this case W.

Environment value tests. put and get enable the movement of values between $\vec{U}$, the list of values leaked to the environment, and the active position of the configuration. put permits an evaluated value to be saved for use by the environment. get permits the environment to interact with a saved argument by moving it into the active term position. This rule leaves a copy of the restored term in $\vec{U}$. The label on this rule carries the position i in $\vec{U}$ that is being restored. Conceptually, put and get ensure that $\vec{U}$ is closed under structural rules.

New name tests. The rules for fun and adv permit the environment to add new function names and new advice. The first rule is necessary for bookkeeping; it allows the context to create an unbounded number of new function names; new names are added to the list of values $\vec{U}$ to maintain the invariant that functions declared in Δ can be inspected by the environment. The second rule is needed for more than bookkeeping. Since the order of advice matters, the rule for $\mathsf{adv}\ p = \alpha$ also has to insert it into the list of advice declarations being carried in $\vec{A}$.

4.4 Bisimulation

Define $\twoheadrightarrow$ to be the reflexive transitive closure of $\xrightarrow{\tau}$. On visible labels define the weak labeled transition relation $\overset{\kappa}{\twoheadrightarrow}$ as $\twoheadrightarrow\xrightarrow{\kappa}$.

Note in the definition of the LTS ($\Gamma;\Delta \vdash \mathbf{M} \xrightarrow{\varkappa} \Gamma';\Delta' \vdash \mathbf{M}'$), that the symbol environment and declaration in the residual ($\Gamma';\Delta'$) are uniquely determined by the initial state ($\Gamma;\Delta$) and label ($\varkappa$). This leads us to define bisimulation as a family of relations between configurations, written $\Gamma;\Delta \vdash \mathbf{M} \sim \mathbf{N}$. It is technically convenient to require that bisimilar configurations have equal length lists of contexts and values. (Alternatively, we could prove that these invariants hold for bisimulations derived from the initial configurations of Definition 17.)

Definition 16. We say that a configuration $\vec{A}/\vec{\mathscr{E}}/M/\vec{U}$ has sort $\langle \Gamma, \Delta, m, n\rangle$ if $\Gamma;\Delta \vdash \vec{A}/\vec{\mathscr{E}}/M/\vec{U}$ is well-formed, the length of $\vec{\mathscr{E}}$ is m, and the length of $\vec{U}$ is n.

We define similarity, $\lesssim$, as the largest family of $\langle \Gamma, \Delta, m, n\rangle$-indexed relations over configurations such that

$$\Gamma;\Delta \vdash \mathbf{M} \lesssim \mathbf{N} \text{ and } \Gamma;\Delta \vdash \mathbf{M} \overset{\kappa}{\twoheadrightarrow} \Gamma';\Delta' \vdash \mathbf{M}'$$

imply that for some $\mathbf{N}'$

$$\Gamma;\Delta \vdash \mathbf{N} \overset{\kappa}{\twoheadrightarrow} \Gamma';\Delta' \vdash \mathbf{N}' \text{ and } \Gamma';\Delta' \vdash \mathbf{M}' \lesssim \mathbf{N}'.$$

$\Gamma;\Delta$-*bisimilarity*, $\sim$ is defined as two way similarity:

$$\Gamma;\Delta \vdash \mathbf{M} \sim \mathbf{N} \ \text{ if } \ \Gamma;\Delta \vdash \mathbf{M} \lesssim \mathbf{N} \ \text{ and } \ \Gamma;\Delta \vdash \mathbf{N} \lesssim \mathbf{M}. \qquad \Box$$

Bisimulation is insensitive to the addition of irrelevant new names to Γ, i.e., If

$$\Gamma;\Delta \vdash \vec{A}/\vec{\mathscr{E}}/M/U,\vec{U} \ \sim\ \vec{B}/\vec{\mathscr{F}}/N/V,\vec{V}$$

and $\Gamma' \cap \Gamma = \emptyset$, then:

$$\Gamma,\Gamma';\Delta \vdash \vec{A}/\vec{\mathscr{E}}/M/U,\vec{U},U \ \sim\ \vec{B}/\vec{\mathscr{F}}/N/V,\vec{V},V$$

Symmetrically, bisimulation is also insensitive to the removal of irrelevant new names from Γ, i.e., names in Γ that are not free in the rest of the configuration can be removed.

As usual, indexed-bisimulation can be formalized as the greatest fixed point of a product lattice [50]. Bisimulation on configurations relates to terms as follows.

Definition 17. Write $\Gamma;\Delta \vdash M \sim N$ if

$$\Gamma;\Delta \ \vdash\ \cdot/\cdot/M/\vec{f} \ \sim\ \cdot/\cdot/N/\vec{f}$$

where $\vec{f}$ are the function names bound in Δ, in declaration order.

Let $fn(M,N) = \{\vec{\phi}, \vec{p}\}$. Write $M \sim N$ if

$$\vec{\phi},\vec{\alpha};\mathsf{pcd}\ \vec{p};\ \mathsf{adv}\ \vec{p} = \vec{\alpha} \vdash M \sim N. \qquad \Box$$

The function symbols $\vec{\phi}$ detect function calls by the term. The primitive pointcut declarations $\mathsf{pcd}\ \vec{p}$ bind the free primitive pointcuts in the term. The advice declarations $\mathsf{adv}\ \vec{p} = \vec{\alpha}$ detect any call to a new function declared at a visible primitive pointcut (by the term). Functions can be introduced by fun transitions to detect any new advice declared at a visible primitive pointcut (by the term).

4.5 Simple Examples

The first examples show that bisimulation yields a β_v, η_v theory.

Example 18 (β_v preserves bisimilarity). A standard LTS proof shows that prefixing by τ preserves bisimilarity. So, since:

$$\Gamma;\Delta \vdash \vec{A}/\vec{\mathscr{E}}/(\lambda x.M)\ U/\vec{U} \xrightarrow{\tau} \Gamma;\Delta \vdash \vec{A}/\vec{\mathscr{E}}/M[x := U]/\vec{U}$$

we have:

$$\Gamma;\Delta \vdash \vec{A}/\vec{\mathscr{E}}/(\lambda x.M)\ U/\vec{U} \sim \vec{A}/\vec{\mathscr{E}}/M[x := U]/\vec{U}$$

Thus, ${\beta_v}^2$ preserves bisimilarity. □

Example 19 (η_v preserves bisimilarity). η_v holds, i.e., in the case where x is not free in U:

$$\Gamma;\Delta \vdash \vec{A}/\vec{\mathscr{E}}/U/\vec{U} \sim \vec{A}/\vec{\mathscr{E}}/\lambda x.Ux/\vec{U}$$

The key case in this proof is to note that the transition

$$\Gamma;\Delta \vdash \vec{A}/\vec{\mathscr{E}}/U/\vec{U} \xrightarrow{\mathsf{app}\ \phi} \Gamma,\phi;\Delta \vdash \vec{A}/\vec{\mathscr{E}}/U\ \phi/\vec{U}$$

on the LHS is matched by the following sequence from the RHS:

$$\Gamma;\Delta \vdash \vec{A}/\vec{\mathscr{E}}/\lambda x.Ux/\vec{U} \xrightarrow{\mathsf{app}\ \phi} \Gamma,\phi;\Delta \vdash \vec{A}/\vec{\mathscr{E}}/(\lambda x.Ux)\ \phi/\vec{U}$$

and

$$\Gamma,\phi;\Delta \vdash \vec{A}/\vec{\mathscr{E}}/(\lambda x.Ux)\ \phi/\vec{U} \xrightarrow{\tau} \Gamma,\phi;\Delta \vdash \vec{A}/\vec{\mathscr{E}}/U\ \phi/\vec{U} \qquad \Box$$

Bisimulation is not a trivial relation: for example, it distinguishes the Church booleans from one another, and likewise the Church numerals. In combination with the two examples above, this provides some justification for our use of the traditional encoding of algebraic datatypes such as booleans and natural numbers.

As demonstrated in Example 18, the order of evaluation and multiplicity of use of "internal" functions are not necessarily detectable. Bisimulation can, however, detect the order and multiplicity of calls to symbolic functions created by the environment.

Example 20 (Detecting order). Consider the following terms.

$$\mathbf{let}\ \mathsf{x} = \phi\,()\ ;\ \mathbf{let}\ \mathsf{y} = \psi\,()\ ;\ ()$$
$$\mathbf{let}\ \mathsf{y} = \psi\,()\ ;\ \mathbf{let}\ \mathsf{x} = \phi\,()\ ;\ ()$$

The LTSs for these terms are immediately distinguished by the initially enabled transition, namely $\mathsf{fcall}\ \phi$ for the first term and $\mathsf{fcall}\ \psi$ for the second. □

Example 21 (Detecting multiplicity). Consider the following terms.

$$\mathbf{let}\ \mathsf{x} = \phi\,()\ ;\ \mathbf{let}\ \mathsf{y} = \phi\,()\ ;\ ()$$
$$\mathbf{let}\ \mathsf{y} = \phi\,()\ ;\ ()$$

The LTSs for the first term may perform the following sequence of transitions: $\mathsf{fcall}\ \phi$, $\mathsf{ret}\ \psi$, $\mathsf{fcall}\ \phi$. The second term can match the first two of these transitions, but not the third. □

[2] We use β_v, η_v for the call-by-value versions of β, η.

These distinctions hold even if all terms involved in the above examples are purely functional, i.e., no aspects. Thus, even for this fragment, our approach makes more distinctions relative to applicative bisimulation and contextual bisimulation for a purely functional language.

Of course, these distinctions are motivated and necessary for the full language with imperative features.

Example 22 (The use of get and put rules). Consider:

$$\begin{array}{ll} \mathsf{M} = \vec{\mathsf{A}}\,;\mathsf{U} & \vec{\mathsf{A}} = \textbf{pcd}\ \mathsf{p}\,;\ \textbf{fun}\ \mathsf{f@p} = \lambda.\mathsf{fls}\,; \\ \mathsf{N} = \lambda.\mathsf{tru} & \mathsf{U} = \lambda.\textbf{let}\ \mathsf{x} = \textbf{not}(\mathsf{f}\ ())\,;(\textbf{adv}\ \mathsf{p} = \lambda.\lambda.\mathsf{x})\,;\mathsf{x} \end{array}$$

Because of the state changes caused by the aspect in U, M is distinguished from N via the context

$$\mathscr{E} = \textbf{let}\ \mathsf{y} = [-]\,;\mathsf{y}()\,;\mathsf{y}()$$

since $\mathscr{E}[\mathsf{M}]$ yields fls and $\mathscr{E}[\mathsf{N}]$ yields tru.

Clearly, this distinction relies crucially on the use of M twice. Applicative bisimulation thus fails to distinguish the terms because it only tests the functions against identical arguments once. In the contextual-bisimulation based work of Koutavas and Wand [36], applicative tests are made against arguments in the contextual closure of the putative bisimulation and the terms are distinguished. In the following example, we essentially show that the LTS is expressive enough to code the distinguishing context $\mathscr{E}$ by using $\mathsf{put}, \mathsf{get}$ tests to permit multiple tests of terms.

Using the definitions above, the behavior of $\mathscr{E}$ can be simulated in the LTS using the $\mathsf{put}, \mathsf{get}$ rules as follows. Consider the initial configuration $\cdot;\cdot \vdash \cdot/\cdot/\mathsf{M}/\cdot$, which has τ transitions to $\cdot;\cdot \vdash \vec{\mathsf{A}}/\cdot/\mathsf{U}/\cdot$. This configuration in turn has a put labeled transition to:

$$\cdot;\cdot \vdash \vec{\mathsf{A}}/\cdot/\mathsf{U}/\mathsf{U}$$

which in turn has an $\mathsf{app}\ \phi$ labeled transition to:

$$\phi;\cdot \vdash \vec{\mathsf{A}}/\cdot/\mathsf{U}\ \phi/\mathsf{U}$$

A few τ transitions from this configuration yields:

$$\phi;\cdot \vdash \vec{\mathsf{A}}\,;\textbf{adv}\ \mathsf{f} = \lambda.\lambda.\mathsf{tru}/\cdot/\mathsf{tru}/\mathsf{U}$$

To reevaluate U, we use a get 1 transition to get:

$$\phi;\cdot \vdash \vec{\mathsf{A}}\,;\textbf{adv}\ \mathsf{f} = \lambda.\lambda.\mathsf{tru}/\cdot/\mathsf{U}/\mathsf{U}$$

An $\mathsf{app}\ \psi$ labeled transition yields:

$$\phi,\psi;\cdot \vdash \vec{\mathsf{A}}\,;\textbf{adv}\ \mathsf{f} = \lambda.\lambda.\mathsf{tru}/\cdot/\mathsf{U}\ \psi/\mathsf{U}$$

This second evaluation of U takes place in the context of the aspect that has been laid down. A few τ transitions from this configuration yields:

$$\phi,\psi;\cdot \vdash \vec{\mathsf{A}}\,;\textbf{adv}\ \mathsf{f} = \lambda.\lambda.\mathsf{tru}\,;\textbf{adv}\ \mathsf{f} = \lambda.\lambda.\mathsf{fls}/\cdot/\mathsf{U}\ \psi/\mathsf{U} \qquad \square$$

Much of the related work is formalized in terms of references, rather than advisable functions. In the next example, we discuss some of the subtleties, using the work of Meyer and Sieber [45] as the basis for comparison.

Example 23 (References versus advisable functions). For a free reference variable x, Meyer-Sieber [45] validate the equivalence $\mathsf{!x}\,;\mathsf{!x} \overset{\text{MS}}{=} \mathsf{!x}$. In our language, this translates roughly to the *in*equivalence demonstrated in Example 21. The difference arises from the weak assumptions one can make about functions relative to references; indeed the equivalence is valid in our language for *bound* references, where stronger assumptions are manifest:

$$\textbf{let}\ \mathsf{x} = \mathsf{ref}\ 0\,;\mathsf{!x}\,;\mathsf{!x} \sim \textbf{let}\ \mathsf{x} = \mathsf{ref}\ 0\,;\mathsf{!x}.$$

Unwinding the definition of references, this is roughly

$$\textbf{pcd}\ \mathsf{p}\,;\textbf{fun}\ \mathsf{f@p} = \lambda.0\,;\mathsf{f}()\,;\mathsf{f}() \sim \textbf{pcd}\ \mathsf{p}\,;\textbf{fun}\ \mathsf{f@p} = \lambda.0\,;\mathsf{f}().$$

But the equivalence does not hold when p is available to the context, since calls to f are then observable. Let $\Delta = \textbf{pcd}\ \mathsf{p}, \textbf{fun}\ \mathsf{f@p} = \lambda.0$. Then

$$\cdot;\Delta \vdash \mathsf{f}()\,;\mathsf{f}() \not\sim \mathsf{f}().$$

Interestingly, the equivalence does hold after an assignment, ie, declaration of non-proceeding advice. Let $\mathsf{A} = \textbf{adv}\ \mathsf{p} = \lambda.\lambda.1$, then

$$\cdot;\Delta \vdash \mathsf{A}\,;\mathsf{f}()\,;\mathsf{f}() \sim \mathsf{A}\,;\mathsf{f}()$$

which corresponds to $(\mathsf{x} := 1\,;\mathsf{!x}\,;\mathsf{!x}) \overset{\text{MS}}{=} (\mathsf{x} := 1\,;\mathsf{!x})$.

Note also that for pure references $\mathsf{!x}\,;\Omega \overset{\text{MS}}{=} \Omega$, whereas the corresponding result for functions does not hold: $\mathsf{f}()\,;\Omega \overset{\text{MS}}{\neq} \Omega$. $\square$

4.6 A reasoning principle

To simplify reasoning about bisimilarity, we develop an upto-principle that eliminates the need to:

- Include terms that do not interact with the state, if they occur in the same position on each side of the bisimulation.
- Replicate values in bisimulations, e.g., arising from a get 1 then a put transition.

The following definition formalizes the replication of values to a relation on configurations.

Definition 24. $\mathscr{R}^{\bullet}_{\mathtt{dup}} \supseteq \mathscr{R}$ is defined inductively as follows. If $\Gamma;\Delta \vdash \vec{A}/\vec{\mathscr{E}}/M/U,\vec{U}\ \ \mathscr{R}\ \ \vec{B}/\vec{\mathscr{F}}/N/V,\vec{V}$, then:

- $\Gamma,\Gamma';\Delta \vdash \vec{A}/\vec{\mathscr{E}}/M/U,\vec{U},U\ \ \mathscr{R}^{\bullet}_{\mathtt{dup}}\ \ \vec{B}/\vec{\mathscr{F}}/N/V,\vec{V},V$
- $\Gamma,\Gamma';\Delta \vdash \vec{A}/\vec{\mathscr{E}}/M/\vec{U}\ \ \mathscr{R}^{\bullet}_{\mathtt{dup}}\ \ \vec{B}/\vec{\mathscr{F}}/N/\vec{V}$ $\square$

We say that a term (resp. evaluation context) is state-free over a symbol environment $\Gamma;\Delta$ if every free name is contained in Γ and the term (resp. evaluation context) contains *no* declaration subterms. The following definition formalizes the addition of identical state-free evaluation contexts / values to a relation on configurations.

Definition 25. $\mathscr{R}^{\bullet}_{\mathtt{sf}} \supseteq \mathscr{R}$ is defined inductively as follows. If L (resp. W, $\mathscr{E}$) is a state-free term (resp. value, context) for $\Gamma,\Gamma';\Delta$, and $\Gamma;\Delta \vdash \vec{A}/\vec{\mathscr{E}}/M/\vec{U}\ \ \mathscr{R}\ \ \vec{B}/\vec{\mathscr{F}}/N/\vec{V}$, then:

- $\Gamma,\Gamma';\Delta \vdash \vec{A}/\vec{\mathscr{E}}/L/\vec{U}\ \ \mathscr{R}^{\bullet}_{\mathtt{sf}}\ \ \vec{B}/\vec{\mathscr{F}}/L/\vec{V}$.
- $\Gamma,\Gamma';\Delta \vdash \vec{A}/\vec{\mathscr{E}}/M/W,\vec{U}\ \ \mathscr{R}^{\bullet}_{\mathtt{sf}}\ \ \vec{B}/\vec{\mathscr{F}}/N/W,\vec{V}$.
- $\Gamma,\Gamma';\Delta \vdash \vec{A}/\mathscr{E},\vec{\mathscr{E}}/M/\vec{U}\ \ \mathscr{R}^{\bullet}_{\mathtt{sf}}\ \ \vec{B}/\mathscr{E},\vec{\mathscr{F}}/N/\vec{V}$. $\square$

Let $\mathscr{R}^{\bullet} = \mathscr{R}^{\bullet}_{\mathtt{dup}} \cup \mathscr{R}^{\bullet}_{\mathtt{sf}}$. Let $\twoheadleftarrow\!\!\!\twoheadrightarrow$ be the reflexive, transitive closure of the least symmetric relation containing $\xrightarrow{\tau}$. The following upto-technique is used to prove equivalences in Section 4.7.

Lemma 26. *Let $\mathscr{R}$ be a $\langle \Gamma,\Delta,m,n \rangle$-indexed relation on configurations. Suppose:*

$$\Gamma;\Delta \vdash \mathbf{M}\ \ \mathscr{R}\ \ \mathbf{N}\ \textit{and}\ \Gamma;\Delta \vdash \mathbf{M} \overset{\kappa}{\twoheadrightarrow} \Gamma';\Delta' \vdash \mathbf{M}'$$

implies there exists $\mathbf{N}'$ such that:

$$\Gamma;\Delta \vdash \mathbf{N} \overset{\kappa}{\twoheadrightarrow} \Gamma';\Delta' \vdash \mathbf{N}'\ \textit{and}\ \Gamma';\Delta' \vdash \mathbf{M}'\ \ (\twoheadleftarrow\!\!\!\twoheadrightarrow;\mathscr{R}^{\bullet};\twoheadleftarrow\!\!\!\twoheadrightarrow)\ \ \mathbf{N}'.$$

Then $\twoheadleftarrow\!\!\!\twoheadrightarrow;\mathscr{R}^{\bullet};\twoheadleftarrow\!\!\!\twoheadrightarrow \subseteq \sim$. $\square$

One very useful consequence of the lemma is that $\sim^{\bullet} \subseteq \sim$[3].

4.7 Examples with local store and higher-order functions

Examples 27 and 28 illustrate equivalences involving local state and higher-order functions—originally due to Meyer and Sieber

[3] **A remark on a closure property of bisimulation.** The results of section 5 imply that $\sim$ is sound for a more general version of definition 25: i.e., if $fn(U),\mathscr{E}$ are bound in Γ,Δ and $\Gamma;\Delta \vdash \vec{A}/\vec{\mathscr{E}}/M/\vec{U}\ \sim\ \vec{B}/\vec{\mathscr{F}}/N/\vec{V}$ then, $\Gamma;\Delta \vdash \vec{A}/\mathscr{E},\vec{\mathscr{E}}/M/U,\vec{U}\ \sim\ \vec{B}/\mathscr{E},\vec{\mathscr{F}}/N/U,\vec{V}$. However, this more general property of $\sim$ is not necessarily sound as part of an upto-proof technique.

[45]. The proofs provided here exemplify the techniques needed to address examples 1–5 and example 7 from [45]. Example 6 involves the equality of locations, and requires extra machinery to code and reason about. To better illustrate the LTS, examples are written in our language directly rather than using the syntactic sugar for references in Example 6.

Example 27 (Local Store). Recall that dynamic aspects generalize local store. This example shows that local declaration of a primitive pointcut and function at that primitive pointcut(providing local store) does not affect computation. Consider the terms:

$$\mathsf{M} = \mathsf{x} \qquad\qquad \mathsf{N} = \mathbf{pcd}\ \mathsf{p}\,;\ \mathbf{fun}\ \mathsf{f@p} = \lambda\,.\,0\,;\,\mathsf{x}$$

We wish to prove $\lambda \mathsf{x}.\mathsf{M} \sim \lambda \mathsf{x}.\mathsf{N}$. By congruence, lemma 32, it suffices to show $\mathsf{M} \sim \mathsf{N}$. Define the relation $\mathscr{R}$ as :

$$\mathsf{x};\cdot \vdash (\cdot/\cdot/\mathsf{x}/\cdot)\ \ \mathscr{R}\ \ (\vec{\mathsf{A}}/\cdot/\mathsf{x}/\cdot)$$

where $\vec{\mathsf{A}} = \mathbf{pcd}\ \mathsf{p}\,;\ \mathbf{fun}\ \mathsf{f@p} = \lambda\,.\,0$.

The only possible transition labels are $\mathsf{app}\ \phi$ and put.

$$\begin{tikzcd}
\mathsf{x};\cdot \vdash \cdot/\cdot/\mathsf{x}/\cdot \arrow[r, "\mathsf{app}\ \phi"] \arrow[d, squiggly, no head, "\mathscr{R}"'] & \mathsf{x},\phi;\cdot \vdash \cdot/\cdot/\mathsf{x}\ \phi/\cdot \arrow[d, squiggly, no head, "\mathscr{R}^{\bullet}_{\mathtt{sf}}"] \\
\mathsf{x};\cdot \vdash \vec{\mathsf{A}}/\cdot/\mathsf{x}/\cdot \arrow[r, "\mathsf{app}\ \phi"] & \mathsf{x},\phi;\cdot \vdash \vec{\mathsf{A}}/\cdot/\mathsf{x}\ \phi/\cdot
\end{tikzcd}$$

$$\begin{tikzcd}
\mathsf{x};\cdot \vdash \cdot/\cdot/\mathsf{x}/\cdot \arrow[r, "\mathsf{put}"] \arrow[d, squiggly, no head, "\mathscr{R}"'] & \mathsf{x};\cdot \vdash \cdot/\cdot/\mathsf{x}/\mathsf{x} \arrow[d, squiggly, no head, "\mathscr{R}^{\bullet}_{\mathtt{sf}}"] \\
\mathsf{x};\cdot \vdash \vec{\mathsf{A}}/\cdot/\mathsf{x}/\cdot \arrow[r, "\mathsf{put}"] & \mathsf{x};\cdot \vdash \vec{\mathsf{A}}/\cdot/\mathsf{x}/\mathsf{x}
\end{tikzcd}$$

By Lemma 26, $\mathsf{x};\cdot \vdash (\cdot/\cdot/\mathsf{x}/\cdot) \sim (\cdot/\cdot/\vec{\mathsf{A}}\,;\,\mathsf{x}/\cdot)$. □

Example 28 (Higher-Order Functions). This example demonstrates reasoning about a call to an unknown procedure.

$$\begin{aligned}
\mathsf{M} &= \mathsf{x}\,(\lambda\,.\,())\,;\ ()\\
\mathsf{N} &= \mathbf{pcd}\ \mathsf{p}\,;\ \mathbf{fun}\ \mathsf{f@p} = \lambda\,.\,0\,;\\
&\quad\ \mathsf{x}\,(\lambda\,.\,(\mathbf{let}\ \mathsf{y} = \mathsf{f}\ ()\,;\,(\mathbf{adv}\ \mathsf{p} = \lambda\,.\,\lambda\,.\,\mathsf{y}+2)\,;\ ()))\,;\\
&\quad\ \mathbf{if}\ ((\mathsf{f}\ ()\ \mathbf{mod}\ 2) = 0)\ \mathbf{then}\ ()\ \mathbf{else}\ \Omega
\end{aligned}$$

In M, the external procedure x is invoked with a functional argument without side effects. In N, x is invoked with an argument that advises the local function f—corresponding to incrementing a local reference by two—thus maintaining the invariant that a call to f yields an even number.

In our proof, we prove the local invariant of evenness separately, without referring to the external function call. The bisimulation principle allows us to modularly add the external function.

By lemma 32, to prove $\lambda \mathsf{x}.\mathsf{M} \sim \lambda \mathsf{x}.\mathsf{N}$ it suffices to show that $\mathsf{M} \sim \mathsf{N}$. Let:

$$\begin{aligned}
&\mathsf{U} = \lambda\,.\,()\\
&\mathscr{E} = [-]\,;\ ()\\
&\vec{\mathsf{A}} = \mathbf{pcd}\ \mathsf{p}\,;\ \mathbf{fun}\ \mathsf{f@p} = \lambda\,.\,0\\
&\mathsf{V} = \lambda\,.\,(\mathbf{let}\ \mathsf{y} = \mathsf{f}\ ()\,;\,(\mathbf{adv}\ \mathsf{p} = \lambda\,.\,\lambda\,.\,\mathsf{y}+2)\,;\ ())\\
&\mathscr{F} = [-]\,;\ \mathbf{if}\ ((\mathsf{f}\ ()\ \mathbf{mod}\ 2) = 0)\ \mathbf{then}\ ()\ \mathbf{else}\ \Omega\\
&\vec{\mathsf{B}}_0 \text{ is the empty advice list}\\
&\vec{\mathsf{B}}_n = \vec{\mathsf{B}}_{n-1}\,;\,(\mathbf{adv}\ \mathsf{p} = \lambda\,.\,\lambda\,.\,2\mathsf{n})
\end{aligned}$$

So, $\mathsf{M} = \mathscr{E}[\mathsf{x}\ \mathsf{U}]$ and $\mathsf{N} = \vec{\mathsf{A}}\,;\,\mathscr{F}[\mathsf{x}\ \mathsf{V}]$. We first prove two purely local results without the external call, to show that the tests under consideration (as given by $\mathscr{E},\mathscr{F}$) do not distinguish $\vec{\mathsf{A}};\vec{\mathsf{B}}_m$ and $\vec{\mathsf{A}};\vec{\mathsf{B}}_n$ for any m,n.

- For any m,n, the configurations $\cdot;\cdot \vdash \vec{\mathsf{A}},\vec{\mathsf{B}}_m/\mathscr{E}/\mathsf{V}/\mathsf{V}$ and $\cdot;\cdot \vdash \vec{\mathsf{A}},\vec{\mathsf{B}}_n/\mathscr{F}/\mathsf{V}/\mathsf{V}$ are bisimilar.
- For any m,n, the configurations $\cdot;\cdot \vdash \vec{\mathsf{A}},\vec{\mathsf{B}}_m/\mathscr{E}/\mathsf{V}/\mathsf{V}$ and $\cdot;\cdot \vdash \vec{\mathsf{A}},\vec{\mathsf{B}}_n/\mathscr{E}/\mathsf{U}/\mathsf{U}$ are bisimilar.

Let m,n range over all non-negative integers. Define:

$$\cdot;\cdot \vdash (\vec{\mathsf{A}},\vec{\mathsf{B}}_m/\mathscr{F}/\mathsf{V}/\mathsf{V})\ \ \mathscr{R}\ \ (\vec{\mathsf{A}},\vec{\mathsf{B}}_n/\mathscr{E}/\mathsf{V}/\mathsf{V}) \tag{1}$$

$$\cdot;\cdot \vdash (\vec{\mathsf{A}},\vec{\mathsf{B}}_m/\mathscr{E}/\mathsf{V}/\mathsf{V})\ \ \mathscr{R}\ \ (\vec{\mathsf{A}},\vec{\mathsf{B}}_n/\mathscr{E}/\mathsf{U}/\mathsf{U}) \tag{2}$$

There are three possibilities for the transition system labels that we discuss below. For each, we address 1. The proof for 2 is identical and omitted.

Case $\mathsf{put},\mathsf{get}\ 1$**:** For $\kappa \in \{\mathsf{put},\mathsf{get}\ 1\}$:

$$\begin{tikzcd}
\cdot;\cdot \vdash \vec{\mathsf{A}},\vec{\mathsf{B}}_m/\mathscr{E}/\mathsf{V}/\mathsf{V} \arrow[r, "\kappa"] \arrow[d, squiggly, no head, "\mathscr{R}"'] & \cdot;\cdot \vdash \vec{\mathsf{A}},\vec{\mathsf{B}}_m/\mathscr{E}/\mathsf{V}/\mathsf{V},\mathsf{V} \arrow[d, squiggly, no head, "\mathscr{R}^{\bullet}_{\mathrm{dup}}"] \\
\cdot;\cdot \vdash \vec{\mathsf{A}},\vec{\mathsf{B}}_n/\mathscr{F}/\mathsf{V}/\mathsf{V} \arrow[r, "\kappa"] & \cdot;\cdot \vdash \vec{\mathsf{A}},\vec{\mathsf{B}}_n/\mathscr{F}/\mathsf{V}/\mathsf{V},\mathsf{V}
\end{tikzcd}$$

Case $\mathsf{app}\ \phi$**:** Use the operational semantics. Since:

$$\begin{aligned}
\cdot;\cdot \vdash \vec{\mathsf{A}},\vec{\mathsf{B}}_n/\mathscr{F}/\mathsf{V}/\mathsf{V} &\overset{\mathsf{app}\ \phi}{\twoheadrightarrow} \cdot;\cdot \vdash \vec{\mathsf{A}},\vec{\mathsf{B}}_{n+1}/\mathscr{F}/()/\mathsf{V}\\
\cdot;\cdot \vdash \vec{\mathsf{A}},\vec{\mathsf{B}}_m/\mathscr{E}/\mathsf{V}/\mathsf{V} &\overset{\mathsf{app}\ \phi}{\twoheadrightarrow} \cdot;\cdot \vdash \vec{\mathsf{A}},\vec{\mathsf{B}}_{m+1}/\mathscr{E}/()/\mathsf{V}
\end{aligned}$$

$$\begin{tikzcd}
\cdot;\cdot \vdash \vec{\mathsf{A}},\vec{\mathsf{B}}_m/\mathscr{E}/\mathsf{V}/\mathsf{V} \arrow[r, "\mathsf{app}\ \phi"] \arrow[d, squiggly, no head, "\mathscr{R}"'] & \phi;\cdot \vdash \vec{\mathsf{A}},\vec{\mathsf{B}}_{m+1}/\mathscr{E}/()/\mathsf{V} \arrow[d, squiggly, no head, "(\twoheadleftarrow\!\!\twoheadrightarrow;\mathscr{R}^{\bullet};\twoheadleftarrow\!\!\twoheadrightarrow)"] \\
\cdot;\cdot \vdash \vec{\mathsf{A}},\vec{\mathsf{B}}_n/\mathscr{F}/\mathsf{V}/\mathsf{V} \arrow[r, "\mathsf{app}\ \phi"] & \phi;\cdot \vdash \vec{\mathsf{A}},\vec{\mathsf{B}}_{n+1}/\mathscr{F}/()/\mathsf{V}
\end{tikzcd}$$

Case $\mathsf{ret}\ \phi$**:** Use the invariant that for any m, the function call $\mathsf{f}\ ()$ in advice context $\vec{\mathsf{A}},\vec{\mathsf{B}}_m$ evaluates to an even number.

$$\begin{tikzcd}
\cdot;\cdot \vdash \vec{\mathsf{A}},\vec{\mathsf{B}}_m/\mathscr{E}/\mathsf{V}/\mathsf{V} \arrow[r, "\mathsf{ret}\ \phi"] \arrow[d, squiggly, no head, "\mathscr{R}"'] & \phi;\cdot \vdash \vec{\mathsf{A}},\vec{\mathsf{B}}_m/\mathscr{E}/()/\mathsf{V} \arrow[d, squiggly, no head, "(\twoheadleftarrow\!\!\twoheadrightarrow;\mathscr{R}^{\bullet};\twoheadleftarrow\!\!\twoheadrightarrow)"] \\
\cdot;\cdot \vdash \vec{\mathsf{A}},\vec{\mathsf{B}}_n/\mathscr{F}/\mathsf{V}/\mathsf{V} \arrow[r, "\mathsf{ret}\ \phi"] & \phi;\cdot \vdash \vec{\mathsf{A}},\vec{\mathsf{B}}_n/\mathscr{F}/()/\mathsf{V}
\end{tikzcd}$$

Therefore, by Lemma 26, $\mathscr{R}$, and hence $\mathscr{R}^{\bullet}$, is contained in bisimilarity. Now, using transitivity of bisimilarity yields:

$$\cdot;\cdot \vdash (\vec{\mathsf{A}}/\mathscr{E}/\mathsf{U}/\cdot) \sim (\vec{\mathsf{A}}/\mathscr{F}/\mathsf{V}/\cdot)$$

Since x is not free in either configuration, we have:

$$\mathsf{x};\cdot \vdash (\vec{\mathsf{A}}/\mathscr{E}/\mathsf{U}/\cdot) \sim (\vec{\mathsf{A}}/\mathscr{F}/\mathsf{V}/\cdot)$$

From example 27, since $\sim^{\bullet}_{\mathtt{sf}} \subseteq \sim$, and $\mathscr{E},\mathsf{U}$ are state-free for x:

$$\mathsf{x};\cdot \vdash (\vec{\mathsf{A}}/\mathscr{E}/\mathsf{U}/\cdot) \sim (\cdot/\mathscr{E}/\mathsf{U}/\cdot)$$

Using transitivity of $\sim$:

$$\mathsf{x};\cdot \vdash (\vec{\mathsf{A}}/\mathscr{F}/\mathsf{V}/\cdot) \sim (\cdot/\mathscr{E}/\mathsf{U}/\cdot)$$

Since

$$\begin{aligned}
\mathsf{x};\cdot \vdash \cdot/\cdot/\mathscr{E}[\mathsf{x}\ \mathsf{U}]/\cdot &\overset{\mathsf{fcall}\ \mathsf{x}}{\twoheadrightarrow} \mathsf{x};\cdot \vdash \cdot/\mathscr{E}/\mathsf{U}/\cdot\\
\mathsf{x};\cdot \vdash \cdot/\cdot/\vec{\mathsf{A}}\,;\,\mathscr{F}[\mathsf{x}\ \mathsf{V}]/\cdot &\overset{\mathsf{fcall}\ \mathsf{x}}{\twoheadrightarrow} \mathsf{x};\cdot \vdash \vec{\mathsf{A}}/\mathscr{F}/\mathsf{V}/\cdot
\end{aligned}$$

the required result,

$$\mathsf{x};\cdot \vdash (\cdot/\cdot/\vec{\mathsf{A}}\,;\,\mathscr{F}[\mathsf{x}\ \mathsf{V}]/\cdot) \sim (\cdot/\cdot/\mathscr{E}[\mathsf{U}]/\cdot)$$

follows since both sides have only weak $\mathsf{fcall}\ \mathsf{x}$ transitions to bisimilar targets. □

5. Results

Bisimilarity is sound and complete relative to observational congruence. The proofs are found in the full paper (see [28]). In this section, we merely give the reader a very high level tour of the results.

The soundness proof has three parts.

- First, we prove that the η_v-relation is a precongruence. This permits us to assume that all values in the $\vec{U}$ portion of the configuration are abstractions. Several of the later proofs are simplified by this assumption.

- Secondly, we prove a substitution lemma that validates substitution of equals-for-equals for contexts that do not capture variables: the reader might want to view this semantically as an instance of the composition principles underlying game semantics [3, 25], and syntactically as our (admittedly peculiar!) variant of the delayed substitutions of the SECD machine [37].
- With this key ingredient in place, the rest of the soundness proof becomes manageable, and dare we say, largely self-explanatory.

The following notion of *compatibility* captures some useful properties of the initial configurations of Definition 17 and those reachable from them.

Definition 29. A pair of LTS configurations $\Gamma;\Delta \vdash \vec{A}/\vec{\mathscr{E}}/M/\vec{U}$ and $\Gamma;\Delta \vdash \vec{B}/\vec{\mathscr{F}}/N/\vec{V}$ are *compatible* if: (a) All advice in Δ is symbolic advice of the form $\mathsf{adv}\ p = \alpha$. (b) If $\mathsf{pcd}\ p \in \Delta$, then there exists $\mathsf{adv}\ p = \alpha \in \Delta$. (c) If $\mathsf{fun}\ f@p = \phi \in \Delta$ then there exists $1 \leq i \leq \min(|\vec{U}|, |\vec{V}|)$ such that $\vec{U}_i = \vec{V}_i = f$ □

The next two lemmas provide the infrastructure required to reason separately about the active term and the remaining pieces of a configuration. Lemma 30 permits the substitution of identical terms for values in the active term spot of bisimilar configurations, while maintaining bisimilarity. Lemma 31 is dual.

Lemma 30. *Suppose* $\Gamma;\Delta \vdash \vec{A}/\vec{\mathscr{E}}/U/\vec{U}$ *and* $\Gamma;\Delta \vdash \vec{B}/\vec{\mathscr{F}}/V/\vec{V}$ *are compatible and* $fn(L) \subseteq \Gamma \cup dn(\Delta)$*. Then:*

$$\Gamma;\Delta \vdash \vec{A}/\vec{\mathscr{E}}/U/\vec{U} \sim \vec{B}/\vec{\mathscr{F}}/V/\vec{V}$$

implies:

$$\Gamma;\Delta \vdash \vec{A}/\vec{\mathscr{E}}/L/\vec{U} \sim \vec{B}/\vec{\mathscr{F}}/L/\vec{V}. \qquad \square$$

Lemma 31. *Suppose* $\Gamma;\Delta \vdash \cdot/\cdot/M/\vec{U}$ *and* $\Gamma;\Delta \vdash \cdot/\cdot/N/\vec{V}$ *are compatible and* $\Gamma;\Delta \vdash \vec{A}/\vec{\mathscr{E}}/()/\vec{W}$ *is well-formed. Then:*

$$\Gamma;\Delta \vdash \cdot/\cdot/M/\vec{U} \sim \cdot/\cdot/N/\vec{V}$$

implies:

$$\Gamma;\Delta \vdash \vec{A}/\vec{\mathscr{E}}/M/\vec{U},\vec{W} \sim \vec{A}/\vec{\mathscr{E}}/N/\vec{V},\vec{W}. \qquad \square$$

These lemmas constitute the basic machinery of the proof that bisimilarity is a congruence (and is therefore sound for contextual equivalence).

Theorem 32 (Congruence of Bisimilarity). *Let* $U_1 \sim U_1'$*,* $U_2 \sim U_2'$ *and* $U \sim U'$*. Let* $M \sim M'$*,* $M_1 \sim M_1'$ *and* $M_2 \sim M_2'$*. Then:*

- $U_1\ U_2 \sim U_1'\ U_2'$
- $\lambda x.M \sim \lambda x.M'$
- $\mathsf{let}\ x = M_1\ ;\ M_2 \sim \mathsf{let}\ x = M_1'\ ;\ M_2'$
- $\mathsf{fun}\ f@p = U\ ;\ M \sim \mathsf{fun}\ f@p = U'\ ;\ M'$
- $\mathsf{pcd}\ p\ ;\ M \sim \mathsf{pcd}\ p\ ;\ M'$
- $\mathsf{adv}\ p = \lambda z.U\ ;\ M \sim \mathsf{adv}\ p = \lambda z.U'\ ;\ M'$ □

The following theorem states that bisimilarity is sound and complete for observational equivalence. The soundness follows immediately from lemma 32. Completeness proceeds via a definablity argument: we show that every distinguishing trace (= finite sequence of visible levels) between two terms, we can construct a context that witnesses the trace. This construction proceeds via an analysis of normal forms for such traces.

Theorem 33 (Completeness). $M \equiv N$ *if and only if* $M \sim N$*.* □

6. Access Control and Type Enforcement

In this section we demonstrate how Type Enforcement (TE) [12, 62] policies—a form of history-sensitive mandatory access control popularized in the NSA's Security-Enhanced Linux (SELinux) [43]—can be encoded as temporal advice and how *security properties* of the resulting system can be established using open bisimulation. TE policies associate types with code and other resources to be protected; henceforth we call these "TE types" to avoid confusion with the usual notion of type found in programming languages. Also, the runtime system associates a current TE type with running code, which determines its privileges: access control decisions are based upon the current TE type and the TE type associated with the resource being accessed. The current TE type evolves as new code is invoked, based upon the current TE type, the TE type associated with the new code, the TE policy, and constraints imposed by the caller. The mechanism permits access control policies that are sensitive to the history of the code that has been executed and constraints imposed by that code.

Example 34 (Web-Server). As an example policy, consider a web-server permitted to listen on ports 80 and 8080 if run by a system administrator, but only upon port 8080 if executed by an ordinary user. When fine-grained access control policies are available, the system administrator might also be prohibited from using any other program to listen on ports 80 or 8080. In this scenario, access privileges depend on both the original identity (system administrator or user) and the code (the web-server) that is running.

To encode the web-server policy, we allow the current TE type to range over $\{\mathsf{adm}, \mathsf{usr}, \mathsf{ws\_adm}, \mathsf{ws\_usr}\}$, the TE type for the web-server code is $\mathsf{ws\_exe}$, and the TE types associated to the ports are $\{\mathsf{port}_{80}, \mathsf{port}_{8080}\}$. Initially the current TE type is adm or usr, then when the web-server is executed, the policy causes the current TE type to change from adm to $\mathsf{ws\_adm}$, or from usr to $\mathsf{ws\_usr}$. In addition, the policy permits

- adm and usr to execute code of TE type $\mathsf{ws\_exe}$;
- $\mathsf{ws\_adm}$ to access ports of TE type $\mathsf{port}_{80}, \mathsf{port}_{8080}$;
- $\mathsf{ws\_usr}$ to access ports of TE type port_{8080}.

With this policy, we expect that code running as usr cannot be influenced by new connections on port 80, even after executing other code. □

The TE mechanism can be implemented with advice—when protected resources are functions that can be advised. To define the advice, we require:

- A finite set of current TE types T and a finite set of TE types E for executable code, assumed disjoint without loss of generality.
- An "allow" relation $allow \subseteq T \times E \times T$ describes when code can execute/access a function and transition to a new TE type, i.e., if the current TE type is t then a function marked with TE code type e can be invoked successfully and transition to current TE type t' if $allow(t, e, t')$.
- An "automatic transition" map $auto : T \times E \to T$ describes TE type transitions that occur automatically when a new function is executed[4], i.e., if the current TE type is t and a function marked with TE code type e is invoked successfully, then it is executed with TE type $auto(t, e)$.
- A finite set of primitive pointcuts Q and a map $type : Q \to E$.

We consider declarations $\mathsf{A}_{\mathsf{curr}}(t)$ representing a private variable curr storing the current TE type initialized with t. The scope of the private variable curr extends over advice A_q, one for each primitive pointcut q, which checks whether a call to a function at q is permitted and updates the current TE type before the call takes

[4] For reasons of space, we do not model the behavior of the SELinux API (`setexeccon`) that allows a caller to choose, subject to "allow", a TE type other than the default "automatic" TE type. However, there are no inherent problems with such modeling.

place. A free variable fail is invoked when an access control check fails. The coding for updating the current TE type uses the same strategy adopted for cflow in Example 12, i.e., the caller's current TE type is stored before proceeding, and restored afterwards.

$$\begin{aligned}
\mathsf{A}_{\mathsf{curr}}(t) &\triangleq \mathsf{pcd}\ p, \mathsf{fun\ curr@}p = \lambda.t \\
\vec{\mathsf{A}}_Q &\triangleq (\mathsf{A}_q \,|\, q \in Q) \\
\mathsf{A}_q &\triangleq \mathsf{adv}\ q = \lambda z.\lambda x.\mathsf{L}_{z,x,q} \\
\mathsf{L}_{z,x,q} &\triangleq \mathsf{let\ next} = \mathit{auto}(\,!\mathsf{curr}, \mathit{type}(q))\,; \\
&\quad\ \mathsf{if}\ \mathit{allow}(\,!\mathsf{curr}, \mathit{type}(q), \mathsf{next})\ \mathsf{then} \\
&\quad\quad\ \mathsf{let\ prev} = \,!\mathsf{curr}\,;\ \mathsf{curr} := \mathsf{next}\,; \\
&\quad\quad\ \mathsf{let\ y} = z\ x\,;\ \mathsf{curr} := \mathsf{prev}\,;\ \mathsf{y} \\
&\quad\ \mathsf{else}\quad \mathsf{fail}\ (\,)
\end{aligned}$$

With the TE policy described in Example 34, and using TE code types as primitive pointcuts, suppose we are given a function webserver@ws_exe that starts a webserver on a port given as an argument, and functions $\mathsf{listen}_{80}\mathsf{@port}_{80}$, $\mathsf{listen}_{8080}\mathsf{@port}_{8080}$ that create listening sockets on ports 80 and 8080 respectively. In such a context, the advice implementing the TE policy prevents the webserver from accessing port 80 when invoked with a current TE type of usr, i.e., if webserver attempts to invoke listen_{80} in the following program, the advice implementing the TE policy will cause fail to be invoked instead, because the invocation of webserver will cause the current TE type to change to ws_usr.

$$\mathsf{A}_{\mathsf{curr}}(\mathsf{usr})\,;\ \vec{\mathsf{A}}_Q\,;\ \mathsf{webserver}\ (80)$$

In this example, we see that the body of $\mathsf{listen}_{80}\mathsf{@port}_{80}$ is irrelevant to computation beginning with TE type usr. To formalize this non-interference property, we first define reachability $\mathit{reach}(t,e)$ of a TE type e from a TE type t to be the least relation such that:

- $\exists t'.\ \mathit{allow}(t,e,t')$ implies $\mathit{reach}(t,e)$
- $\exists t', e'.\ \mathit{allow}(t,e',t')$ and $\mathit{reach}(t',e)$ implies $\mathit{reach}(t,e)$

Reachability $\mathit{reach}(t,q)$ of a primitive pointcut from a TE type t is then defined to hold exactly when $\mathit{reach}(t,\mathit{type}(q))$. In the example above, the TE code type port_{80} is not reachable from usr.

Now Proposition 35 demonstrates that we can take a program that declares functions at public primitive pointcuts, impose aspects for type enforcement on those public primitive pointcuts, then arbitarily change the bodies of functions declared at primitive pointcuts unreachable from the initial TE type without changing the behavior of the program.

Proposition 35. *Consider a list of variables Γ, a TE type $t_{init} \in T$, a finite set of function names F, a map $pcd : F \to Q$, and values U_f, U'_f for each $f \in F$ such that:*

- $\mathsf{fail} \in \Gamma$
- $\vec{A}_1 = (\mathsf{pcd}\ q \,|\, q \in Q)$
- $\vec{A}_2 = \mathsf{A}_{\mathsf{curr}}(t_{init}), \vec{\mathsf{A}}_Q$
- $\vec{B} = (\mathsf{fun}\ f\mathsf{@}pcd(f) = U_f \,|\, f \in F)$
- $\vec{B}' = (\mathsf{fun}\ f\mathsf{@}pcd(f) = U'_f \,|\, f \in F)$
- *For $f \in F$, $\mathit{fn}(U_f) \cup \mathit{fn}(U'_f) \subseteq \Gamma \cup Q$*
- *For $f \in F$, if $\mathit{reach}(t_{init}, pcd(f))$ then $U_f = U'_f$.*
- $\mathit{fn}(M) \subseteq \Gamma \cup Q \cup F$

Then:

$$\Gamma; \vec{A}_1 \vdash \vec{A}_2\,;\, \vec{B}\,;\, M \sim \vec{A}_2\,;\, \vec{B}'\,;\, M \qquad \square$$

PROOF (SKETCH). By open bisimulation in combination with results from Section 5. Recall that an advice declaration is symbolic if it has form $\mathsf{adv}\ q = \alpha$ and that a function declaration $\mathsf{fun}\ f\mathsf{@}q = U$ is symbolic if U is a variable. The relation $\mathscr{R}$ contains:

$$\Gamma; \vec{A}_1, \vec{C}_1 \vdash (\vec{A}_2, \vec{B}, \vec{C}_2 / \cdot / N / \vec{V})\ \ \mathscr{R}\ \ (\vec{A}_2, \vec{B}', \vec{C}_2 / \cdot / N' / \vec{V}')$$

Whenever Γ, $\vec{A}_1$, and $\vec{A}_2$ satisfy the conditions in the statement of the result, and there exists a TE type $t \in T$, a set $F_1 \subseteq F$, and binary relations on values S_1, S_2 such that:

- $\vec{C}_1$ consists of symbolic function and advice declarations with free primitive pointcuts in Q and free variables in Γ.
- For $f \in F$, $\mathit{reach}(t, pcd(f))$ iff $f \notin F_1$.
- $\vec{B} = (\mathsf{fun}\ f\mathsf{@}pcd(f) = U_f \,|\, f \in F_1)$
- $\vec{B}' = (\mathsf{fun}\ f\mathsf{@}pcd(f) = U'_f \,|\, f \in F_1)$
- For $f \in F_1, \mathit{fn}(U_f) \cup \mathit{fn}(U'_f) \subseteq \Gamma \cup Q$
- $\vec{C}_2$ consists of symbolic advice with free primitive pointcuts in Q and free variables in Γ, interleaved with advice updating curr such that !curr returns t.
- S_1 is the least set such that:
  - $x \in \Gamma$ implies $(x,x) \in S_1$
  - $\alpha \in \Gamma$ and $(W,W') \in S_1$ implies $(\alpha\texttt{<}W\texttt{>}, \alpha\texttt{<}W'\texttt{>}) \in S_1$
  - $f \notin F_1$ and $(W,W') \in S_1$ implies $(\lambda x.\mathsf{L}_{z,x,q}[z := W], \lambda x.\mathsf{L}_{z,x,q}[z := W']) \in S_1$
  - $f \in F_1$ and $\mathit{fn}(W) \cup \mathit{fn}(W') \subseteq \Gamma \cup Q$ implies $(\lambda x.\mathsf{L}_{z,x,q}[z := W], \lambda x.\mathsf{L}_{z,x,q}[z := W']) \in S_1$
- $S_2 = S_1 \cup \{(f,f) \,|\, f \in dn(\vec{C}_1) \cup F_1\}$
- $(N,N') \in S_2$ or there exists $x \in \Gamma$ and $(W,W') \in S_2$ such that $N = W\ x$ and $N' = W'\ x$.
- $\vec{V}$ and $\vec{V}'$ have the same length, and, for all i, $(V_i, V'_i) \in S_2$.

It can be verified that $\mathscr{R}^\bullet$ is a bisimulation. To prove the main result, we reason backwards, with the aim of reducing the result to an instance of the bisimulation $\mathscr{R}^\bullet$ established above. The first step is to reduce the desired conclusion to:

$$\Gamma_1; \vec{A}_1, \vec{C}_0 \vdash (\cdot/\cdot/\vec{A}_2, \vec{B}, M/\cdot) \sim (\cdot/\cdot/\vec{A}_2, \vec{B}', M/\cdot)$$

Symbolic advice on public primitive pointcuts is added. Note that there are no function declarations in $\vec{A}_1$, so no function names need to be placed into the value lists. Fresh variables are added to Γ: $\Gamma_1 = \Gamma, (\alpha_q \,|\, q \in Q)$ and $\vec{C}_0 = (\mathsf{adv}\ q = \alpha_q \,|\, q \in Q)$. Now, since reduction is included in bisimilarity, it suffices to show:

$$\Gamma_1; \vec{A}_1, \vec{C}_0 \vdash (\vec{A}_2, \vec{B}/\cdot/M/\cdot) \sim (\vec{A}_2, \vec{B}'/\cdot/M/\cdot)$$

Without loss of generality we assume that the function declarations $\vec{B}$ and $\vec{B}'$ factor as $\vec{B} = \vec{B}_1, \vec{B}_2$ and $\vec{B} = \vec{B}'_1, \vec{B}'_2$, where $\vec{B}_1$ and $\vec{B}'_1$ consist of function declarations at primitive pointcuts reachable from t_{init} (i.e., if f is declared in $\vec{B}_1$ or $\vec{B}'_1$, then $\mathit{reach}(t_{init}, pcd(f))$), and $\vec{B}_2$ and $\vec{B}'_2$ consist of function declarations at primitive pointcuts unreachable from t_{init}. By hypothesis, $\vec{B}_1 = \vec{B}'_1$.

$$\Gamma_1; \vec{A}_1, \vec{C}_0 \vdash (\vec{A}_2, \vec{B}_1, \vec{B}_2/\cdot/M/\cdot) \sim (\vec{A}_2, \vec{B}_1, \vec{B}'_2/\cdot/M/\cdot)$$

We introduce fresh variables $\Gamma_2 = \Gamma_1, (x_f \,|\, f \in dn(\vec{B}_1))$, symbolic function definitions $\vec{B}_3 = (\mathsf{fun}\ f\mathsf{@}pcd(f) = x_f \,|\, f \in dn(\vec{B}_1))$, and a value list with the original common function bodies $\vec{W} = (U_f \,|\, f \in dn(\vec{B}_1))$. Using the substitution result used in the the proof of congruence (see [28]), we need only show:

$$\Gamma_2; \vec{A}_1, \vec{C}_0 \vdash (\vec{A}_2, \vec{B}_3, \vec{B}_2/\cdot/M/\vec{W}) \sim (\vec{A}_2, \vec{B}_3, \vec{B}'_2/\cdot/M/\vec{W})$$

A simple bisimulation proof shows that the function declarations and the policy advice declarations can be swapped, and by moving $\vec{A}_2, \vec{B}_3, \vec{B}_2$ and $\vec{A}_2, \vec{B}_3, \vec{B}'_2$ back to M, Lemma 31 can be applied because $\mathit{fn}(\vec{W}) \subseteq \Gamma \cup Q$. This yields that it suffices to show:

$$\Gamma_2; \vec{A}_1, \vec{C}_0 \vdash (\vec{A}_2, \vec{B}_3, \vec{B}_2/\cdot/M/\cdot) \sim (\vec{A}_2, \vec{B}_3, \vec{B}'_2/\cdot/M/\cdot)$$

A simple bisimulation proof shows that this follows from (adding function names from $dn(\vec{B}_1)$ to the value lists to ensure compatibility):

$$\Gamma_2; \vec{A}_1, \vec{C}_1 \vdash (\vec{A}_2, \vec{B}_2/\cdot/M/dn(\vec{B}_3)) \sim (\vec{A}_2, \vec{B}'_2/\cdot/M/dn(\vec{B}_3))$$

where we take $\vec{C}_1 = \vec{C}_0, \vec{B}_3$. Now Lemma 30 cannot be applied immediately, because $\mathit{fn}(M) \cap \mathit{dn}(\vec{B}_2)$ may not be empty (note that $\mathit{dn}(\vec{B}_2) = \mathit{dn}(\vec{B}'_2)$), so we separate those function names by introducing fresh variables $\Gamma_3 = \Gamma_2, (y_f \,|\, f \in \mathit{dn}(\vec{B}_2))$ and considering L such that $\mathit{fn}(L) \subseteq \mathit{fn}(\Gamma_3) \cup \mathit{dn}(\vec{A}_1, \vec{C}_1)$ and $M = L[(y_f \,|\, f \in \mathit{dn}(\vec{B}_2)) := (f \,|\, f \in \mathit{dn}(\vec{B}_2))]$. Again, by the substitution result used in the proof of congruence, we need only show:

$$\Gamma_2; \vec{A}_1, \vec{C}_1 \vdash (\vec{A}_2, \vec{B}_2/\cdot/L/\mathit{dn}(\vec{B}_3), \mathit{dn}(\vec{B}_2)) \sim (\vec{A}_2, \vec{B}'_2/\cdot/L/\mathit{dn}(\vec{B}_3), \mathit{dn}(\vec{B}_2))$$

Or equivalently, regarding F as a list:

$$\Gamma_2; \vec{A}_1, \vec{C}_1 \vdash (\vec{A}_2, \vec{B}_2/\cdot/L/F) \sim (\vec{A}_2, \vec{B}'_2/\cdot/L/F)$$

Since $\mathit{fn}(L) \subseteq \mathit{fn}(\Gamma_3) \cup \mathit{dn}(\vec{A}_1, \vec{C}_1)$, Lemma 30 tells us that it suffices to prove, for some $x \in \Gamma_2$ (known to be non-empty):

$$\Gamma_2; \vec{A}_1, \vec{C}_1 \vdash (\vec{A}_2, \vec{B}_2/\cdot/x/F) \sim (\vec{A}_2, \vec{B}'_2/\cdot/x/F)$$

This follows from the fact that $\mathscr{R}^\bullet$ is a bisimulation. □

Thus we have shown how a non-interference property of advice implementing a history-sensitive access control policy can be established via open bisimulation.

7. Conclusion

This paper is a step towards leveling the formal playing field between aspects and other programming paradigms.

We have described a first (to our knowledge) description of bisimulation for aspect languages. As an indication of its use, we have demonstrated its utility towards bridging a formal gap that exists between the foundations and realizations of Open Modules.

Our bisimulation principle combines techniques used to address mobile processes (open bisimulation), names in the nu-calculus (via tracking leaked secrets in the LTS) and the lambda calculus (ENF-bisimulation). To this mixture, we contribute new techniques to show that bisimilarity is a congruence. Even though we have taken a purely untyped and operational view in this paper, the infrastructure that we have developed holds promise as foundations to address issues of semantic types and logical relation based reasoning for aspect languages.

Our results suggest that aspects are no more difficult to address formally and reason about than well-studied classical issues of higher-order imperative programs. These results complement ongoing research in the aspect community on the design and implementation of aspect languages.

Acknowledgements. We gratefully acknowledge suggestions by anonymous referees. James Riely was supported by NSF Career 0347542. Radha Jagadeesan and Corin Pitcher were supported by NSF Cybertrust 0430175.

References

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

[2] M. Abadi and C. Fournet. Access control based on execution history. In *Proceedings of the Network and Distributed System Security Symposium Conference*, 2003.

[3] S. Abramsky, R. Jagadeesan, and P. Malacaria. Full abstraction for PCF. *Inf. Comput.*, 163(2):409–470, 2000.

[4] S. Abramsky and C.-H. L. Ong. Full abstraction in the lazy lambda calculus. *Inf. Comput.*, 105(2):159–267, 1993.

[5] M. Aksit, K. Wakita, J. Bosch, L. Bergmans, and A. Yonezawa. Abstracting object-interactions using composition-filters. In *Object-based distributed processing, LNCS*, 1993.

[6] J. Aldrich. Open modules: Modular reasoning about advice. In A. P. Black, editor, *ECOOP*, volume 3586 of *Lecture Notes in Computer Science*, pages 144–168. Springer, 2005.

[7] R. Alur. The benefits of exposing calls and returns. In M. Abadi and L. de Alfaro, editors, *CONCUR*, volume 3653 of *Lecture Notes in Computer Science*, pages 2–3. Springer, 2005.

[8] R. Alur and P. Madhusudan. Adding nesting structure to words. In O. H. Ibarra and Z. Dang, editors, *Developments in Language Theory*, volume 4036 of *Lecture Notes in Computer Science*, pages 1–13. Springer, 2006.

[9] P. Avgustinov, E. Bodden, E. Hajiyev, L. Hendren, O. Lhoták, O. de Moor, N. Ongkingco, D. Sereni, G. Sittampalam, and J. Tibble. Aspects for trace monitoring. In K. Havelund, M. Nunez, G. Rosu, and B. Wolff, editors, *Formal Approaches to Testing Systems and Runtime Verification (FATES/RV)*, Lecture Notes in Computer Science. Springer, 2006.

[10] L. Bergmans. *Composing Concurrent Objects - Applying Composition Filters for the Development and Reuse of Concurrent Object-Oriented Programs*. Ph.D. thesis, University of Twente, 1994.

[11] C. Bockisch, M. Haupt, M. Mezini, and K. Ostermann. Virtual machine support for dynamic join points. In *AOSD*, pages 83–92, 2004.

[12] W. E. Boebert and R. Y. Kain. A practical alternative to hierarchical integrity policies. In *Proceedings of the Eighth National Computer Security Conference*, 1985.

[13] C. Clifton and G. T. Leavens. MiniMAO$_1$: An imperative core language for studying aspect-oriented reasoning. *Science of Computer Programming*, 2006. To appear.

[14] C. Clifton, G. T. Leavens, and M. Wand. Parameterized aspect calculus: A core calculus for the direct study of aspect-oriented languages. At `http://www.cs.iastate.edu/~cclifton/papers/TR03-13.pdf`, 2003.

[15] Y. Coady, G. Kiczales, M. J. Feeley, and G. Smolyn. Using AspectC to improve the modularity of path-specific customization in operating system code. In *ESEC / SIGSOFT FSE*, pages 88–98, 2001.

[16] D. S. Dantas and D. Walker. Harmless advice. In J. G. Morrisett and S. L. P. Jones, editors, *POPL*, pages 383–396. ACM, 2006.

[17] R. De Nicola and M. Hennessy. Testing equivalences for processes. *Theoretical Computer Science*, 34(1–2):83–133, Nov. 1984.

[18] C. Dutchyn, D. B. Tucker, and S. Krishnamurthi. Semantics and scoping of aspects in higher-order languages. *Science of Computer Programming*, 2006. To appear. Preliminary version "Pointcuts and advice in higher-order languages" in AOSD 03.

[19] M. Felleisen, D. P. Friedman, E. Kohlbecker, and B. Duba. A syntactic theory of sequential control. *Theor. Comput. Sci.*, 52(3):205–237, 1987.

[20] R. Filman and D. Friedman. Aspect-oriented programming is quantification and obliviousness. In *Workshop on Advanced Separation of Concerns*, 2000.

[21] A. D. Gordon. Bisimilarity as a theory of functional programming. *Electr. Notes Theor. Comput. Sci.*, 1, 1995.

[22] A. D. Gordon. Operational equivalences for untyped and polymorphic object calculi. In A. D. Gordon and A. M. Pitts, editors, *Higher-Order Operational Techniques in Semantics*, Publications of the Newton Institute, pages 9–54. Cambridge University Press, 1998.

[23] A. D. Gordon and G. D. Rees. Bisimilarity for a first-order calculus of objects with subtyping. In *POPL*, pages 386–395, 1996.

[24] D. J. Howe. Proving congruence of bisimulation in functional programming languages. *Inf. Comput.*, 124(2):103–112, 1996.

[25] J. M. E. Hyland and C.-H. L. Ong. On full abstraction for PCF: I, II, and III. *Inf. Comput.*, 163(2):285–408, 2000.

[26] R. Jagadeesan, A. Jeffrey, and J. Riely. An untyped calculus of aspect oriented programs. In *Conference Record of ECOOP 03: The European Conference on Object-Oriented Programming*, volume 2743 of *Lecture Notes in Computer Science*, 2003.

[27] R. Jagadeesan, A. Jeffrey, and J. Riely. Typed parametric polymorphism for aspects. *Science of Computer Programming*, 2006. To appear.

[28] R. Jagadeesan, C. Pitcher, and J. Riely. Open bisimulation for aspects (full version). Available at http://www.teasp.org/bisimulation, 2007.

[29] A. Jeffrey and J. Rathke. A theory of bisimulation for a fragment of concurrent ML with local names. *Theor. Comput. Sci.*, 323(1-3):1–48, 2004. Preliminary version appeared in IEEE LICS 1999.

[30] A. Jeffrey and J. Rathke. Java Jr: Fully abstract trace semantics for a core Java language. In *ESOP*, volume 3444 of *LNCS*, pages 423–438. Springer, 2005.

[31] G. Kiczales, E. Hilsdale, J. Hugunin, M. Kersten, J. Palm, and W. G. Griswold. An overview of AspectJ. *Lecture Notes in Computer Science*, 2072:327–355, 2001.

[32] G. Kiczales, J. Lamping, A. Mendhekar, C. Maeda, C. V. Lopes, J.-M. Loingtier, and J. Irwin. Aspect-oriented programming. In *European Conference on Object-Oriented Programming (ECOOP)*, 1997.

[33] G. Kiczales and M. Mezini. Aspect-oriented programming and modular reasoning. In *ICSE '05: Proceedings of the 27th international conference on software engineering*, pages 49–58, New York, NY, USA, 2005. ACM Press.

[34] V. Koutavas and M. Wand. Bisimulations for untyped imperative objects. In P. Sestoft, editor, *Proc. ESOP 2006*, volume 3924 of *Lecture Notes in Computer Science*, pages 146–161. Springer, Mar. 2006.

[35] V. Koutavas and M. Wand. Proving class equivalence. submitted for publication, July 2006.

[36] V. Koutavas and M. Wand. Small bisimulations for reasoning about higher-order imperative programs. In J. G. Morrisett and S. L. P. Jones, editors, *POPL*, pages 141–152. ACM, 2006.

[37] P. J. Landin. The mechanical evaluation of expressions. *Computer Journal*, 6(4):308–320, Jan. 1964.

[38] S. Lassen. Eager normal form bisimulation. In *LICS*, pages 345–354. IEEE Computer Society, 2005.

[39] S. Lassen. Head normal form bisimulation for pairs and the lambda-mu calculus. In *LICS*, 2006. In the proceedings of the 21st IEEE Symposium on Logic in Computer Science (LICS 2006). To appear.

[40] H. C. Li, S. Krishnamurthi, and K. Fisler. Modular verification of open features using three-valued model checking. *Autom. Softw. Eng.*, 12(3):349–382, 2005.

[41] K. J. Lieberherr. *Adaptive Object-Oriented Software: The Demeter method with propagation patterns*. PWS Publishing Company, 1996.

[42] J. Ligatti, D. Walker, and S. Zdancewic. A type-theoretic interpretation of pointcuts and advice. *Science of Computer Programming*, 2006. To appear.

[43] P. A. Loscocco and S. D. Smalley. Meeting critical security objectives with Security-Enhanced Linux. In *Proceedings of the 2001 Ottawa Linux Symposium*, 2001.

[44] H. Masuhara, G. Kiczales, and C. Dutchyn. A compilation and optimization model for aspect-oriented programs. In G. Hedin, editor, *CC*, volume 2622 of *Lecture Notes in Computer Science*, pages 46–60. Springer, 2003.

[45] A. R. Meyer and K. Sieber. Towards fully abstract semantics for local variables. In *POPL*, pages 191–203, 1988.

[46] E. Moggi. Notions of computation and monads. *Information and Computation*, 93(1):55–92, 1991.

[47] S. Nakajima and T. Tamai. Lightweight formal analysis of aspect-oriented models. In *UML2004 Workshop on Aspect-Oriented Modeling*, 2004.

[48] N. Ongkingco, P. Avgustinov, J. Tibble, L. Hendren, O. de Moor, and G. Sittampalam. Adding open modules to AspectJ. In *AOSD '06: Proceedings of the 5th international conference on Aspect-oriented software development*, pages 39–50, New York, NY, USA, 2006. ACM Press.

[49] H. Ossher and P. Tarr. Multi-dimensional separation of concerns and the hyperspace approach. In *Proceedings of the Symposium on Software Architectures and Component Technology: The State of the Art in Software Development*, 2001.

[50] A. M. Pitts. Operationally-based theories of program equivalence. In P. Dybjer and A. M. Pitts, editors, *Semantics and Logics of Computation*, Publications of the Newton Institute, pages 241–298. Cambridge University Press, 1997.

[51] H. Rajan and K. J. Sullivan. Classpects: unifying aspect- and object-oriented language design. In G.-C. Roman, W. G. Griswold, and B. Nuseibeh, editors, *ICSE*, pages 59–68. ACM, 2005.

[52] D. Sangiorgi. *Expressing Mobility in Process Algebras: First-Order and Higher-Order Paradigms*. PhD thesis CST–99–93, Department of Computer Science, University of Edinburgh, 1992.

[53] D. Sangiorgi. *Expressing Mobility in Process Algebras: First Order and Higher Order Paradigms*. PhD thesis, University of Edinburgh, 1993.

[54] D. Sangiorgi. A theory of bisimulation for the pi-calculus. *Acta Inf.*, 33(1):69–97, 1996.

[55] D. Sangiorgi. Bisimulation: From the origins to today. In *LICS*, pages 298–302. IEEE Computer Society, 2004.

[56] D. Sangiorgi. The bisimulation proof method: Enhancements and open problems. In R. Gorrieri and H. Wehrheim, editors, *FMOODS*, volume 4037 of *Lecture Notes in Computer Science*, pages 18–19. Springer, 2006.

[57] M. Sihman and S. Katz. Model checking applications of aspects and superimpositions. In *Foundations of Aspect Languages*, 2003.

[58] E. Sumii and B. C. Pierce. A bisimulation for type abstraction and recursion. In J. Palsberg and M. Abadi, editors, *POPL*, pages 63–74. ACM, 2005.

[59] P. L. Tarr and H. Ossher. Hyper/J: Multi-dimensional separation of concerns for Java. In *ICSE*, pages 729–730, 2001.

[60] N. Ubayashi and T. Tamai. Aspect-oriented programming with model checking. In *AOSD '02: Proceedings of the 1st international conference on Aspect-oriented software development*, pages 148–154, New York, NY, USA, 2002. ACM Press.

[61] D. Walker, S. Zdancewic, and J. Ligatti. A theory of aspects. In C. Runciman and O. Shivers, editors, *ICFP*, pages 127–139. ACM, 2003.

[62] K. M. Walker, D. F. Sterne, M. L. Badger, M. J. Petkac, D. L. Shermann, and K. A. Oostendorp. Confining root programs with Domain and Type Enforcement (DTE). In *Proceedings of the Sixth USENIX UNIX Security Symposium*, 1996.

[63] M. Wand, G. Kiczales, and C. Dutchyn. A semantics for advice and dynamic join points in aspect-oriented programming. *TOPLAS*, 26(5):890–910, September 2004.

A. Overview of Proofs

This section provides a sketch of the soundness and completeness results for the bisimulation relative to observational congruence. The remaining sections of the appendix contains detailed sketches of all relevant results. In this section, we focus on the technical novelties of our analysis.

A.1 Soundness

In this subsection, we show that $\sim$ is a congruence. From this it is straightforward to show that $M \sim N$ implies $M \equiv N$.

This proof has three parts. Appendix B proves that the η-relation is a precongruence. This permits us to assume that all values in the $\vec{U}$ portion of the configuration are abstractions. Several of the proofs in this subsection rely on this assumption. Secondly, we prove a substitution lemma that validates substitution of equals-for-equals for contexts that do not capture variables: the reader might want to view this semantically as an instance of the composition principles underlying game semantics [3, 25], and syntactically as our (admittedly peculiar!) variant of the delayed substitutions of the SECD machine [37]. With this key ingredient in place, the rest of the soundness proof becomes manageable, and dare we say, largely self-explanatory.

A substitution result. The substitutions that we consider provide two kinds of substitution information on a LTS configuration:

- What value from the list of values in a configuration is substituted for a variable? This information is indicated by the positional index in $\vec{U}$ in a LTS configuration.
- Which contexts in the the list of evaluation contexts need to be substituted into the enclosing context? This information is specified by an integer stack.

Definition 36. An *extended substitution*, σ, is a pair of a partial function from variables to integers and an integer stack.

If ϕ is in the domain of the partial function of σ, we will use $\sigma(\phi)$ for the value of the partial function of σ. We will use $\sigma \uplus (\phi \mapsto k)$ for the operation of extending the domain of the partial function of σ to include ϕ: this operation is undefined if ϕ is already in the domain of the partial function.

We use $\mathit{empty}(\sigma)$ to return the emptiness of the stack; $\mathit{top}(\sigma)$ to return the top value of the stack; and $\mathit{sum}(\sigma)$ to return the sum of the values on the stack. We use $\mathit{pushone}(\sigma)$ to return a new stack with 1 pushed onto the top; $\mathit{pop}(\sigma)$ to return a new stack without the top element. □

We define the function Z_m to compress the top (rightmost) m elements of a context sequence; the return value is a pair with the compressed context and remainder. We also define the function Z_σ, which compresses a context sequence iteratively using the stack in σ — the argument to Z_σ must be a sequence of evaluation contexts of length $\mathit{sum}(\sigma)$; the result is a sequence whose length equals the length of σ. For example, suppose the stack of σ is $n_1, 1, n_2$, where n_i is the length of $\vec{\mathscr{E}}_i$ and $\mathscr{G}_i$ is the result of compressing $\vec{\mathscr{E}}_i$. Then $Z_{\mathit{top}(\sigma)}(\vec{\mathscr{E}}_1 \mathscr{F} \vec{\mathscr{E}}_2) = \langle \vec{\mathscr{E}}_1 \mathscr{F}, \mathscr{G}_2 \rangle$, and $Z_\sigma(\vec{\mathscr{E}}_1 \mathscr{F} \vec{\mathscr{E}}_2) = \mathscr{G}_1 \mathscr{F} \mathscr{G}_2$. The definitions are as follows.[5]

$$\begin{aligned}
Z(\cdot)(\mathscr{G}) &= \mathscr{G} \\
Z(\vec{\mathscr{E}}\mathscr{F})(\mathscr{G}) &= Z(\vec{\mathscr{E}})(\mathscr{F}[\mathscr{G}]) \\
Z_m(\mathscr{E}_1 \ldots \mathscr{E}_n) &= \langle (\mathscr{E}_1 \ldots \mathscr{E}_{n-m}), Z(\mathscr{E}_{n-m+1} \ldots \mathscr{E}_{n-1})(\mathscr{E}_n) \rangle
\end{aligned}$$

$$Z_\sigma(\vec{\mathscr{E}}) = \begin{cases} \vec{\mathscr{E}} & \text{if } \mathit{empty}(\sigma) \\ \vec{\mathscr{F}}'\mathscr{G} & \text{if } Z_{\mathit{top}(\sigma)}(\vec{\mathscr{E}}) = \langle \vec{\mathscr{F}}, \mathscr{G} \rangle \text{ and } \vec{\mathscr{F}}' = Z_{\mathit{pop}(\sigma)}(\vec{\mathscr{F}}) \end{cases}$$

Definition 37. σ is valid for $\Gamma;\Delta \vdash \mathbf{M}$ if:

- (sorting) If α is an advice variable, $\vec{U}_{\sigma(\alpha)}$ is of the form $\lambda z.U$;
- (acyclicity) there is a total ordering of Γ, say $\phi_1, \ldots, \phi_n$ satisfying: for any $1 \leq k \leq n$; ϕ_k is not free in $\sigma(\phi_j)$ for $j \geq k$,
- the domain of σ is a subset of Γ; and
- $\mathit{sum}(\sigma)$ is less than the length of $\vec{\mathscr{E}}$, where $\mathbf{M} = \_/\vec{\mathscr{E}}/\_/\_$.

If σ is valid for $\Gamma;\Delta \vdash \vec{A}/\vec{\mathscr{E}}/M/\vec{U}$, we define $[\Gamma;\Delta \vdash \vec{A}/\vec{\mathscr{E}}/M/\vec{U}]_\sigma = \Gamma';\Delta' \vdash \vec{A}'/\vec{\mathscr{E}}''/M'/\vec{U}'$ where (a) for every metavariable χ, χ' is derived by substituting ϕ by $\sigma(\phi)$ in the configuration — the substitution in carried out following the total order of the variables testifying to the validity of σ — and (b) $\vec{\mathscr{E}}'' = Z_\sigma(\mathscr{E}')$. □

Definition 38. Write $\Gamma;\Delta \vdash \mathbf{M} \lessapprox_\sigma \mathbf{N}$ if there exists $\Gamma;\Delta \vdash \mathbf{M}' \lesssim \mathbf{N}'$ such that σ is valid for $\Gamma;\Delta \vdash \mathbf{M}'$ and for $\Gamma;\Delta \vdash \mathbf{N}'$ and $\Gamma;\Delta \vdash \mathbf{M} = [\Gamma;\Delta \vdash \mathbf{M}']_\sigma$ and $\Gamma;\Delta \vdash \mathbf{N} = [\Gamma;\Delta \vdash \mathbf{N}']_\sigma$. □

Two configurations are related by $\lessapprox_\sigma$ if they are in the σ-image of configurations that are related by $\lesssim$.

Proposition 39. *The relation* $\lessapprox = \bigcup_\sigma \lessapprox_\sigma$ *is a simulation.*

PROOF SKETCH. See Appendix C. □

Identity inclusion lemmas. The notion of *compatibility* captures some useful properties of the initial configurations of Definition 17 and those reachable from them.

A pair of LTS configurations $\Gamma;\Delta \vdash \vec{A}/\vec{\mathscr{E}}/M/\vec{U}$ and $\Gamma;\Delta \vdash \vec{B}/\vec{\mathscr{F}}/N/\vec{V}$ are *compatible* if: (a) All advice in Δ is symbolic advice of the form $\mathsf{adv}\ q = \alpha$. (b) If $\mathsf{pcd}\ q \in \Delta$, then there exists $\mathsf{adv}\ q = \alpha \in \Delta$. (c) If $\mathsf{fun}\ f\mathtt{@}q = \phi \in \Delta$ then there exists $1 \leq i \leq \min(|\vec{U}|, |\vec{V}|)$ such that $\vec{U}_i = \vec{V}_i = f$

The next two lemmas provide the infrastructure required to reason separately about the active term and the remaining pieces of a configuration. Lemma 30 permits the substitution of identical terms for values in the active term spot of bisimilar configurations, while maintaining bisimilarity.

For proofs, see Appendices D and E.

Suppose $\Gamma;\Delta \vdash \vec{A}/\vec{\mathscr{E}}/U/\vec{U}$ and $\Gamma;\Delta \vdash \vec{B}/\vec{\mathscr{F}}/V/\vec{V}$ are compatible and $\mathit{fn}(L) \subseteq \Gamma \cup \mathit{dn}(\Delta)$. Then

$$\Gamma;\Delta \vdash \vec{A}/\vec{\mathscr{E}}/U/\vec{U} \sim \vec{B}/\vec{\mathscr{F}}/V/\vec{V}$$

implies

$$\Gamma;\Delta \vdash \vec{A}/\vec{\mathscr{E}}/L/\vec{U} \sim \vec{B}/\vec{\mathscr{F}}/L/\vec{V}. \qquad \square$$

Lemma 31 is dual.

Suppose $\Gamma;\Delta \vdash \cdot/\cdot/M/\vec{U}$ and $\Gamma;\Delta \vdash \cdot/\cdot/N/\vec{V}$ are compatible and $\Gamma;\Delta \vdash \vec{A}/\vec{\mathscr{E}}/()/\vec{W}$ is well-formed. Then

$$\Gamma;\Delta \vdash \cdot/\cdot/M/\vec{U} \sim \cdot/\cdot/N/\vec{V}$$

implies

$$\Gamma;\Delta \vdash \vec{A}/\vec{\mathscr{E}}/M/\vec{U},\vec{W} \sim \vec{A}/\vec{\mathscr{E}}/N/\vec{V},\vec{W}. \qquad \square$$

Given this machinery, the proof that bisimulation is a congruence (and is therefore sound for contextual equivalence) is quite routine. Appendix F contains a sketch of the proof.

A.2 Completeness

We show that $M \equiv N$ implies $M \sim N$ by demonstrating the contrapositive. Let s, t range over *traces* of visible labels $s,t ::= \kappa_1, \ldots, \kappa_n$, with empty trace ε.

Definition 40. A *complete normal trace* is a trace that is generable by the following grammar over labels:

$$\begin{aligned}
\text{START} &::= \text{TERM}^*, \mathsf{put}, \text{CTXT}^* \\
\text{TERM} &::= \mathsf{fcall}\ \phi, \mathsf{put}, \text{CTXT}^*, \mathsf{ret}\ \psi \mid \mathsf{acall}\ \alpha, \mathsf{put}, \text{CTXT}^*, \mathsf{ret}\ \psi \\
\text{CTXT} &::= \mathsf{get}\ i, \mathsf{app}\ \phi, \text{TERM}^*, \mathsf{put} \mid \mathsf{fun}\ f\mathtt{@}q = \phi \mid \mathsf{adv}\ q = \alpha
\end{aligned}$$

A *normal trace* is a prefix of a complete normal trace.

[5] To avoid confusion, we elide sequence element separators here.

Proposition 41. *(a) If* $\Gamma;\Delta \vdash \mathbf{M} \lesssim \mathbf{N}$ *and* $\Gamma;\Delta \vdash \mathbf{M} \overset{s}{\twoheadrightarrow}$ *then* $\Gamma;\Delta \vdash \mathbf{N} \overset{s}{\twoheadrightarrow}$. *(b) If* $\Gamma;\Delta \vdash \mathbf{M} \not\lesssim \mathbf{N}$ *then for some normal trace s and label* $\kappa \in \{\mathsf{fcall}, \mathsf{acall}\}$*:* $\Gamma;\Delta \vdash \mathbf{M} \overset{s}{\twoheadrightarrow}\overset{\kappa}{\twoheadrightarrow}$ *and* $\Gamma;\Delta \vdash \mathbf{N} \overset{s}{\twoheadrightarrow}\overset{\kappa}{\not\twoheadrightarrow}$. □

In Appendix G we show how to construct a term $\mathbb{C}^s_t[\Gamma;\Delta \vdash \mathbf{M}]$ to satisfy the following lemmas (upto a structural equivalence that allows reordering of unrelated declarations). Intuitively, the configuration $\mathbf{M}$ is in the process of performing actions s,t with its (supplied) context; actions s are completed, whereas t have yet to be performed.

Proposition 42. *Let* s,κ,t *be a normal trace. If* $\Gamma;\Delta \vdash \mathbf{M} \overset{\kappa}{\twoheadrightarrow} \Gamma';\Delta' \vdash \mathbf{M}'$ *then* $\mathbb{C}^s_{\kappa,t}[\Gamma;\Delta \vdash \mathbf{M}] \twoheadrightarrow \mathbb{C}^{s,\kappa}_t[\Gamma';\Delta' \vdash \mathbf{M}']$. □

Proposition 43. *Let* s,κ *be a normal trace, where* $\kappa \in \{\mathsf{fcall}, \mathsf{acall}\}$*, and let* $\Gamma;\Delta \vdash \mathbf{M}$ *be an LTS state in which* signal *does not occur. (a) If* $\Gamma;\Delta \vdash \mathbf{M} \overset{\kappa}{\twoheadrightarrow}$ *then* $\mathbb{C}^s_\kappa[\Gamma;\Delta \vdash \mathbf{M}]\lightning$. *(b) If* $\Gamma;\Delta \vdash \mathbf{M} \overset{\kappa}{\not\twoheadrightarrow}$ *then* $\neg(\mathbb{C}^s_\kappa[\Gamma;\Delta \vdash \mathbf{M}]\lightning)$. □

Starting from Definition 17, completeness follows by induction on the length of trace s from Proposition 41b, using Propositions 42 and 43.

B. η equality is a congruence

In this section, we sketch the proof that η equality is a congruence.

Definition 44. We define a relation R^η_1 on configurations as follows: $\Gamma;\Delta \vdash \mathbf{M}\ R^\eta_1\ \mathbf{N}$ if $\mathbf{N}$ is got from $\mathbf{M}$ by replacing a value subterm U (with x not free) by $\lambda x.Ux$.

Let $R^\eta_\star$ be the reflexive and transitive closure of R^η_1. □

We overload R^η_1 (rep. $R^\eta_\star$) and also use them as a relation between terms (resp. evaluation contexts, etc).

Lemma 45. $R^\eta_\star$ *is a bisimulation.*

PROOF. Suffices to prove that R^η_1 is a bisimulation upto $R^\eta_\star$.

We first consider the case for τ-transitions.
Let $\vec{A}\ R^\eta_1\ \vec{A}'$, $\mathscr{E}[\cdot]\ R^\eta_1\ \mathscr{E}'[\cdot]$, $U\ R^\eta_1\ U'$, $V\ R^\eta_1\ V'$, $N\ R^\eta_1\ N'$. Then:

- $\vec{A}/\mathscr{E}\big[N[x := U]\big] R^\eta_\star \vec{A}'/\mathscr{E}'[N'[x := U']]$
- $\vec{A}(U) R^\eta_\star \vec{A}'(U')$, if U is an abstraction or $U = U'$.

So, if $N R^\eta_1 N'$, and $\vec{A}/N \twoheadrightarrow \vec{A}_1/N_1$ then $\vec{A}'/N' \twoheadrightarrow \vec{A}'_1/N'_1$ such that: $\vec{A}_1/N_1\ R^\eta_\star\ \vec{A}'_1/N'_1$.

Thus, if

$$\Gamma;\Delta \vdash \vec{A}/\vec{\mathscr{E}}/M/\vec{U}\ R^\eta_1\ \Gamma;\Delta \vdash \vec{A}'/\vec{\mathscr{E}}'/M'/\vec{U}'$$

and

$$\Gamma;\Delta \vdash \vec{A}/\vec{\mathscr{E}}/M/\vec{U} \xrightarrow{\tau} \Gamma;\Delta \vdash \vec{A}_1/\vec{\mathscr{E}}/M_1/\vec{U}$$

then:

$$\Gamma;\Delta \vdash \vec{A}'/\vec{\mathscr{E}}'/M'/\vec{U}' \overset{\tau}{\twoheadrightarrow} \Gamma;\Delta \vdash \vec{A}'_1/\vec{\mathscr{E}}/M'_1/\vec{U}'$$

such that

$$\Gamma;\Delta \vdash \vec{A}_1/\vec{\mathscr{E}}/M_1/\vec{U} R^\eta_\star \Gamma;\Delta \vdash \vec{A}'_1/\vec{\mathscr{E}}'/M'_1/\vec{U}'$$

We next consider the case for $\mathsf{fcall}\ x$ transition:

$$\Gamma;\Delta \vdash \vec{A}/\vec{\mathscr{E}}/\mathscr{F}[\phi\ V]/\vec{U} \xrightarrow{\mathsf{fcall}\ \phi} \Gamma;\Delta \vdash \vec{A}/\vec{\mathscr{E}},\mathscr{F}/V/\vec{U}$$

The key case to consider for this transition is the one that replaces ϕ by its one-step eta expansion $\lambda y.\phi y$ to yield $\Gamma;\Delta \vdash \vec{A}'/\vec{\mathscr{E}}'/M'/\vec{U}'$ with $\vec{A} = \vec{A}'; \vec{\mathscr{E}} = \vec{\mathscr{E}}'; M' = \mathscr{F}[\lambda y.\phi y\ V]; \vec{U} = \vec{U}'$. The required matching $\mathsf{fcall}\ x$ transition is validated after one β_v transition.

All the other cases of transitions with non-τ labels are straightforward and are omitted. □

C. Proof of substitution lemma

Our aim is to show that $\lessapprox$ is a simulation.

Since the LTS transitions never reduce Γ or the list of carried values, the critical point of interaction is the stack of the substitution. The following lemma addresses the cases when the stack of evaluation contexts is altered: when a new frame is added due to the call transitions, an extra frame is added to the substitution stack; when an evaluation frame is taken off due to the return transition, σ's stack determines the appropriate number of frames that actually need to be removed. The extra conditions for enabling non-τ transitions cases account for the priority given by the LTS to τ transitions. (The proof of Lemma 48 demonstrates how the top of the stack in σ is incremented.)

Lemma 46. *Let* σ *be valid for* $\Gamma;\Delta \vdash \mathbf{M}$ *and suppose* $\Gamma;\Delta \vdash \mathbf{M} \xrightarrow{\varkappa} \Gamma';\Delta' \vdash \mathbf{M}'$.

- *If* $\varkappa = \tau$ *then* $[\Gamma;\Delta \vdash \mathbf{M}]_\sigma \xrightarrow{\tau} [\Gamma';\Delta' \vdash \mathbf{M}']_\sigma$.
- *If* $\varkappa \in \{\mathsf{app}, \mathsf{put}, \mathsf{get}, \mathsf{fun}, \mathsf{adv}\}$ *and* $[\Gamma;\Delta \vdash \mathbf{M}]_\sigma \overset{\tau}{\not\rightarrow}$ *then* $[\Gamma;\Delta \vdash \mathbf{M}]_\sigma \xrightarrow{\varkappa} [\Gamma';\Delta' \vdash \mathbf{M}']_\sigma$.
- *If* $\varkappa \in \{\mathsf{fcall}\ \phi, \mathsf{acall}\ \alpha\}$ *and* $\phi, \alpha \notin dom(\sigma)$ *then* $[\Gamma;\Delta \vdash \mathbf{M}]_\sigma \xrightarrow{\varkappa} [\Gamma';\Delta' \vdash \mathbf{M}']_{pushone(\sigma)}$.
- *If* $\varkappa = \mathsf{ret}\ \phi$ *and* $\phi \notin dom(\sigma)$ *then let* $\mathbf{M} = \vec{A}/\vec{\mathscr{E}}/M/\vec{U}$ *and* $\mathbf{M}' = \vec{B}/\vec{\mathscr{F}}/N/\vec{V}$. *Let* $Z_{top(\sigma)}(\vec{\mathscr{E}}) = \langle \vec{\mathscr{F}}, \mathscr{G}\rangle$. *Then* $[\Gamma;\Delta \vdash \vec{A}/\vec{\mathscr{F}},\mathscr{G}/M/\vec{U}]_\sigma \xrightarrow{\varkappa} [\Gamma,\phi;\Delta \vdash \vec{B}/\vec{\mathscr{F}}/\mathscr{G}[\phi]/\vec{V}]_{pop(\sigma)}$.

The next two lemmas together show that for any σ, $\lessapprox_\sigma$ is a simulation-upto $\lessapprox$. The first lemma addresses the case when the left-hand configuration is in a stuck state. The second lemma addresses the remaining cases.

Lemma 47. *If*

- $\Gamma;\Delta \vdash \vec{A}_1/\vec{\mathscr{E}}_1/M_1/\vec{U}_1 \lessapprox_\sigma \vec{A}'_1/\vec{\mathscr{E}}'_1/M'_1/\vec{U}'_1$, *and*
- $\Gamma;\Delta \vdash \vec{A}_1/\vec{\mathscr{E}}_1/M_1/\vec{U}_1 \xrightarrow{\kappa} \Gamma_2;\Delta_2 \vdash \vec{A}_2/\vec{\mathscr{E}}_2/M_2/\vec{U}_2$

then there exists σ' *such that:*

- $\Gamma;\Delta \vdash \vec{A}'_1/\vec{\mathscr{E}}'_1/M'_1/\vec{U}'_1 \overset{\kappa}{\twoheadrightarrow} \Gamma_2;\Delta_2 \vdash \vec{A}'_2/\vec{\mathscr{E}}'_2/M'_2/\vec{U}'_2$
- $\Gamma_2;\Delta_2 \vdash \vec{A}_2/\vec{\mathscr{E}}_2/M_2/\vec{U}_2 \lessapprox_{\sigma'} \vec{A}'_2/\vec{\mathscr{E}}'_2/M'_2/\vec{U}'_2$

PROOF. Since $\Gamma;\Delta \vdash \vec{A}_1/\vec{\mathscr{E}}_1/M_1/\vec{U}_1 \lessapprox_\sigma \vec{A}'_1/\vec{\mathscr{E}}'_1/M'_1/\vec{U}'_1$ there exists: $\Gamma;\Delta \vdash \vec{A}/\vec{\mathscr{E}}/M/\vec{U} \lesssim \vec{A}'/\vec{\mathscr{E}}'/M'/\vec{U}'$ such that:

- σ is valid for $\Gamma;\Delta \vdash \vec{A}/\vec{\mathscr{E}}/M/\vec{U}$ and $\Gamma;\Delta \vdash \vec{A}'/\vec{\mathscr{E}}'/M'/\vec{U}'$
- $[\vec{A}/\vec{\mathscr{E}}/M/\vec{U}]_\sigma = \vec{A}_1/\vec{\mathscr{E}}_1/M_1/\vec{U}_1$ and
- $[\vec{A}'/\vec{\mathscr{E}}'/M'/\vec{U}']_\sigma = \vec{A}'_1/\vec{\mathscr{E}}'_1/M'_1/\vec{U}'_1$

There is a non-τ labeled transition from $\Gamma;\Delta \vdash \vec{A}_1/\vec{\mathscr{E}}_1/M_1/\vec{U}_1$. So, there are only three possible forms for M_1:

1. $M_1 = [U]_\sigma$
2. $M_1 = [\mathscr{F}[x\ V]]_\sigma$, and $\sigma(x)\uparrow$
3. $M_1 = [\mathscr{F}[\alpha\texttt{<}V\texttt{>}\ W]]_\sigma$, and $\sigma(\alpha)\uparrow$

We consider the cases in turn below:

2. Using $\Gamma;\Delta \vdash \vec{A}/\vec{\mathscr{E}}/M/\vec{U} \lesssim \vec{A}'/\vec{\mathscr{E}}'/M'/\vec{U}'$:

   $$\Gamma;\Delta \vdash \vec{A}'/\vec{\mathscr{E}}'/M'/\vec{U}' \overset{\tau}{\twoheadrightarrow} \Gamma;\Delta \vdash \vec{B}'/\vec{\mathscr{E}}'/\mathscr{F}'[x\ V']/\vec{U}'$$

   such that

   $$\Gamma;\Delta \vdash \vec{A}/\vec{\mathscr{E}}/\mathscr{F}[x\ V]/\vec{U} \lesssim \vec{A}'/\vec{\mathscr{E}}'/\mathscr{F}'[x\ V']/\vec{U}'$$

   Using $\mathsf{fcall}\ x$ transition on both:

   $$\Gamma;\Delta \vdash \vec{A}/\vec{\mathscr{E}},\mathscr{F}/V/\vec{U} \lesssim \vec{A}'/\vec{\mathscr{E}}',\mathscr{F}'/V'/\vec{U}'$$

   From Lemma 46, σ is valid for $\Gamma;\Delta \vdash \vec{B}'/\vec{\mathscr{E}}'/\mathscr{F}'[x\ V']/\vec{U}'$ and

   $$[\Gamma;\Delta \vdash \vec{A}'/\vec{\mathscr{E}}'/M'/\vec{U}']_\sigma \overset{\tau}{\twoheadrightarrow} [\Gamma;\Delta \vdash \vec{B}'/\vec{\mathscr{E}}'/\mathscr{F}'[x\ V']/\vec{U}']_\sigma$$

   Since $\sigma(x)\uparrow$, $\mathsf{fcall}\ x$ transition is enabled after σ substitution, and result follows from Lemma 46.

3. Similar to (2.)
1. Using $\Gamma;\Delta \vdash \vec{A}/\vec{\mathscr{E}}/U/\vec{U} \lesssim \vec{A}'/\vec{\mathscr{E}}'/M'/\vec{U}'$:
$$\Gamma;\Delta \vdash \vec{A}'/\vec{\mathscr{E}}'/M'/\vec{U}' \overset{\tau}{\twoheadrightarrow} \Gamma;\Delta \vdash \vec{B}'/\vec{\mathscr{E}}'/U'/\vec{U}'$$
such that
$$\Gamma;\Delta \vdash \vec{A}/\vec{\mathscr{E}}/U/\vec{U} \lesssim \vec{A}'/\vec{\mathscr{E}}'/U'/\vec{U}'$$
The applicable LTS transitions are $\mathsf{ret}\cdot, \mathsf{app}\cdot, \mathsf{get}\cdot, \mathsf{put}$ and the advise transition.
We illustrate with put: the cases for $\mathsf{app}\cdot, \mathsf{get}\cdot$ are essentially identical.
Using put on both sides, we get:
$$\Gamma;\Delta \vdash \vec{A}/\vec{\mathscr{E}}/U/\vec{U},U \lesssim \vec{A}'/\vec{\mathscr{E}}'/U'/\vec{U}',U'$$
Substitution of values into values preserves values. So, put transition is enabled after σ substitution on both sides. Result follows using Lemma 46 on both sides of the above.
The case for $\mathsf{ret}\cdot$ differs only in the stack management of extended substitutions as handled by Lemma 46. □

The following lemma relies on the mimicking of internal τ-reductions using LTS transitions. One way to view the case of the proof for lambda-application is as our version of the delayed substitutions of the SECD machine [37].

Lemma 48. *Let $\kappa \neq \tau$. If*

- $\Gamma;\Delta \vdash [\vec{A}/\vec{\mathscr{E}}/M/\vec{U}]_\sigma \lessapprox_\sigma [\vec{A}'/\vec{\mathscr{E}}'/M'/\vec{U}']_\sigma$, *and*
- $\Gamma;\Delta \vdash \vec{A}_1/\vec{\mathscr{E}}_1/M_1/\vec{U}_1 \overset{\kappa}{\twoheadrightarrow} \Gamma_2;\Delta_2 \vdash \vec{A}_2/\vec{\mathscr{E}}_2/M_2/\vec{U}_2$,

then there exists σ' such that

- $\Gamma;\Delta \vdash \vec{A}'_1/\vec{\mathscr{E}}'_1/M'_1/\vec{U}'_1 \overset{\kappa}{\twoheadrightarrow} \Gamma_2;\Delta_2 \vdash \vec{A}'_2/\vec{\mathscr{E}}'_2/M'_2/\vec{U}'_2$, *and*
- $\Gamma_2;\Delta_2 \vdash \vec{A}_2/\vec{\mathscr{E}}_2/M_2/\vec{U}_2 \lessapprox_{\sigma'} \vec{A}'_2/\vec{\mathscr{E}}'_2/M'_2/\vec{U}'_2$

Proof. Let the weak transition $\overset{\kappa}{\twoheadrightarrow}$ have n τ transitions before κ. Proof is by induction on n. Base case $n = 0$ has been addressed in Lemma 47. Assume proof for $n <= k$. Consider $n = k+1$.

Since $\Gamma;\Delta \vdash \vec{A}_1/\vec{\mathscr{E}}_1/M_1/\vec{U}_1 \lessapprox_\sigma \vec{A}'_1/\vec{\mathscr{E}}'_1/M'_1/\vec{U}'_1$ there exists: $\Gamma;\Delta \vdash \vec{A}/\vec{\mathscr{E}}/M/\vec{U} \lesssim \vec{A}'/\vec{\mathscr{E}}'/M'/\vec{U}'$ such that:

- σ is valid for $\Gamma;\Delta \vdash \vec{A}/\vec{\mathscr{E}}/M/\vec{U}$ and $\Gamma;\Delta \vdash \vec{A}'/\vec{\mathscr{E}}'/M'/\vec{U}'$
- $[\vec{A}/\vec{\mathscr{E}}/M/\vec{U}]_\sigma = \vec{A}_1/\vec{\mathscr{E}}_1/M_1/\vec{U}_1$ and
- $[\vec{A}'/\vec{\mathscr{E}}'/M'/\vec{U}']_\sigma = \vec{A}'_1/\vec{\mathscr{E}}'_1/M'_1/\vec{U}'_1$

There are two cases corresponding to whether the τ transition is enabled before substitution or not.
Transition enabled before substitution. The first case is when
$$\Gamma;\Delta \vdash \vec{A}/\vec{\mathscr{E}}/M/\vec{U} \xrightarrow{\tau} \Gamma;\Delta \vdash \vec{B}/\vec{\mathscr{E}}/N/\vec{U}$$
In this case, by Lemma 46
$$[\Gamma;\Delta \vdash \vec{A}/\vec{\mathscr{E}}/M/\vec{U}]_\sigma \xrightarrow{\tau} [\Gamma;\Delta \vdash \vec{B}/\vec{\mathscr{E}}/N/\vec{U}]_\sigma$$
Since $\Gamma;\Delta \vdash \vec{A}/\vec{\mathscr{E}}/M/\vec{U} \lesssim \vec{A}'/\vec{\mathscr{E}}'/M'/\vec{U}'$, there exists:
$$\Gamma;\Delta \vdash \vec{A}'/\vec{\mathscr{E}}'/M'/\vec{U}' \overset{\tau}{\twoheadrightarrow} \Gamma;\Delta \vdash \vec{A}'/\vec{\mathscr{E}}'/N'/\vec{U}'$$
such that
$$\Gamma;\Delta \vdash \vec{B}/\vec{\mathscr{E}}/N/\vec{U} \lesssim \vec{B}'/\vec{\mathscr{E}}'/N'/\vec{U}'$$
and by Lemma 46, σ is valid for $\Gamma;\Delta \vdash \vec{A}'/\vec{\mathscr{E}}'/N'/\vec{U}'$ and:
$$[\Gamma;\Delta \vdash \vec{A}'/\vec{\mathscr{E}}'/M'/\vec{U}']_\sigma \overset{\tau}{\twoheadrightarrow} [\Gamma;\Delta \vdash \vec{A}'/\vec{\mathscr{E}}'/N'/\vec{U}']_\sigma$$
In this case, result follows from the induction hypothesis, since the transition $\overset{\kappa}{\twoheadrightarrow}$: $[\Gamma;\Delta \vdash \vec{B}/\vec{\mathscr{E}}/N/\vec{U}]_\sigma \overset{\kappa}{\twoheadrightarrow} \Gamma_2;\Delta_2 \vdash \vec{A}_2/\vec{\mathscr{E}}_2/M_2/\vec{U}_2$ has only k τ transitions.
Transition enabled by substitution. There are two possible forms for M_1:

1. $M_1 = [\mathscr{F}[x\,V]]_\sigma$, and $\sigma(x)$ is defined.
2. $M_1 = [\mathscr{F}[\alpha\texttt{<}V\texttt{>}\,W]]_\sigma$, and $\sigma(\alpha)$ is defined.

We consider the first case below. Let $\sigma(x) = i$, and the value at $i'th$ position in $\vec{U}$ is $\lambda y.N$.

This proof first evaluates the function body: we address the evaluation of N with V substituted for the formal y.

Consider the following sequence of transitions:
$$\begin{aligned}
&\Gamma;\Delta \vdash \vec{A}/\vec{\mathscr{E}}/\mathscr{F}[x\,V]/\vec{U}\\
\xrightarrow{\mathsf{fcall}\,x}\ &\Gamma;\Delta \vdash \vec{A}/\vec{\mathscr{E}},\mathscr{F}/V/\vec{U}\\
\xrightarrow{\mathsf{put}}\ &\Gamma;\Delta \vdash \vec{A}/\vec{\mathscr{E}},\mathscr{F}/V/\vec{U},V\\
\xrightarrow{\mathsf{get}\,i}\ &\Gamma;\Delta \vdash \vec{A}/\vec{\mathscr{E}},\mathscr{F}/\lambda y.N/\vec{U},V\\
\xrightarrow{\mathsf{app}\,\phi}\ &\Gamma,\phi;\Delta \vdash \vec{A}/\vec{\mathscr{E}},\mathscr{F}/\lambda y.N\ \phi/\vec{U},V\\
\xrightarrow{\tau}\ &\Gamma,\phi;\Delta \vdash \vec{A}/\vec{\mathscr{E}},\mathscr{F}/N[y := \phi]/\vec{U},V\\
\overset{\tau}{\twoheadrightarrow}\ &\Gamma,\phi;\Delta \vdash \vec{B}/\vec{\mathscr{E}},\mathscr{F}/L/\vec{U},V
\end{aligned}$$
Since $\Gamma;\Delta \vdash \vec{A}/\vec{\mathscr{E}}/M/\vec{U} \lesssim \vec{A}'/\vec{\mathscr{E}}'/M'/\vec{U}'$:

- $M' = \mathscr{F}'[x\,V']$
- There is a similar sequence resulting in
$$\Gamma,\phi;\Delta \vdash \vec{B}'/\vec{\mathscr{E}}',\mathscr{F}'/L'/\vec{U}',V'$$

Let $\sigma_3 = inctop((\sigma \uplus (\phi \mapsto |(\vec{U},\vec{V})|)))$. Since ϕ is new, and one internal extra evaluation context has been exposed:

- σ_3 is valid for $\Gamma,\phi;\Delta \vdash \vec{B}/\vec{\mathscr{E}},\mathscr{F}/L/\vec{U},V$ and
- σ_3 is valid for $\Gamma,\phi;\Delta \vdash \vec{B}'/\vec{\mathscr{E}}',\mathscr{F}'/L'/\vec{U}',V'$.

The number of τ reductions from $[\Gamma,\phi;\Delta \vdash \vec{B}/\vec{\mathscr{E}},\mathscr{F}/L/\vec{U}',V']_{\sigma_3}$ is $<= k$. So, induction hypothesis applies and we deduce the existence of a σ_4 such that:

- $[\Gamma,\phi;\Delta \vdash \vec{B}/\vec{\mathscr{E}},\mathscr{F}/L/\vec{U},V]_{\sigma_3} \overset{\tau}{\twoheadrightarrow} [\Gamma,\phi;\Delta \vdash \vec{C}/\vec{\mathscr{E}},\mathscr{F}/K/\vec{U},V]_{\sigma_4}$
- $[\Gamma,\phi;\Delta \vdash \vec{B}'/\vec{\mathscr{E}}',\mathscr{F}'/L'/\vec{U},V]_{\sigma_3} \overset{\tau}{\twoheadrightarrow} [\Gamma,\phi;\Delta \vdash \vec{C}'/\vec{\mathscr{E}}',\mathscr{F}'/K'/\vec{U}',V']_{\sigma_4}$
- σ_4 is valid for $\Gamma,\phi;\Delta \vdash \vec{C}/\vec{\mathscr{E}},\mathscr{F}/K/\vec{U},V$
- σ_4 is valid for $\Gamma,\phi;\Delta \vdash \vec{C}'/\vec{\mathscr{E}}',\mathscr{F}'/K'/\vec{U}',V'$.
- $\Gamma,\phi;\Delta \vdash \vec{C}/\vec{\mathscr{E}},\mathscr{F}/K/\vec{U},V \lesssim \vec{C}'/\vec{\mathscr{E}}',\mathscr{F}'/K'/\vec{U}',V'$

There are no τ transitions in the resulting configurations using σ_4. There are now two cases, depending on whether $[K]_{\sigma_4}$ is a value or not. If it is not a value, we are in a form that can appeal to Lemma 47.

If $[K]_{\sigma_4}$ is a value, say W, consider:
$$\begin{aligned}
&\Gamma,\phi;\Delta \vdash \vec{C}/\vec{\mathscr{E}},\mathscr{F}/W/\vec{U},V\\
\xrightarrow{\mathsf{put}}\ &\Gamma,\phi;\Delta \vdash \vec{C}/\vec{\mathscr{E}},\mathscr{F}/W/\vec{U},V,W\\
\xrightarrow{\mathsf{ret}\,\psi}\ &\Gamma,\phi,\psi;\Delta \vdash \vec{C}/\vec{\mathscr{E}}/\mathscr{F}[\psi]/\vec{U},V,W
\end{aligned}$$
This sequence can be mimicked from $\Gamma,\phi;\Delta \vdash \vec{C}'/\vec{\mathscr{E}}',\mathscr{F}'/W'/\vec{U}',V'$ to yield $\Gamma,\phi,\psi;\Delta \vdash \vec{C}'/\vec{\mathscr{E}}'/\mathscr{F}'[\psi]/\vec{U}',V',W'$ and the induction hypothesis applies to the following data:

- $\sigma_5 = pop(\sigma_4) \uplus (\psi \mapsto |\vec{U}',V',W'|)$
- $[\Gamma,\phi,\psi;\Delta \vdash \vec{C}/\vec{\mathscr{E}},\mathscr{F}/\mathscr{F}[\psi]/\vec{U},V,W]_{\sigma_5}$
- $[\Gamma,\phi,\psi;\Delta \vdash \vec{C}'/\vec{\mathscr{E}}',\mathscr{F}'/\mathscr{F}'[\psi]/\vec{U}',V',W']_{\sigma_5}$

yielding the overall required result for the case when $M_1 = [\mathscr{F}[x\,V]]_\sigma$.

We now address the case for advice application, i.e., $M_1 = [\mathscr{F}[\alpha\texttt{<}V\texttt{>}\,W]]_\sigma$. Let $\sigma(\alpha) = i$. Let the value at $i'th$ position in $\vec{U}$ be $\lambda z.\lambda x.X$. Consider the following sequence of transitions:
$$\begin{aligned}
&\Gamma;\Delta \vdash \vec{A}/\vec{\mathscr{E}}/\mathscr{F}[\alpha\texttt{<}V\texttt{>}\,W]/\vec{U}\\
\xrightarrow{\mathsf{acall}\,\alpha}\ &\Gamma;\Delta \vdash \vec{A}/\vec{\mathscr{E}},\mathscr{F}/W/\vec{U},V\\
\xrightarrow{\mathsf{put}}\ &\Gamma;\Delta \vdash \vec{A}/\vec{\mathscr{E}},\mathscr{F}/V/\vec{U},V,W\\
\xrightarrow{\mathsf{get}\,i}\ &\Gamma;\Delta \vdash \vec{A}/\vec{\mathscr{E}},\mathscr{F}/\lambda z.\lambda x.X/\vec{U},V,W\\
\xrightarrow{\mathsf{app}\,\phi}\ &\Gamma,\phi;\Delta \vdash \vec{A}/\vec{\mathscr{E}},\mathscr{F}/\lambda z.\lambda x.X\ \phi/\vec{U},V,W\\
\xrightarrow{\tau}\ &\Gamma,\phi;\Delta \vdash \vec{A}/\vec{\mathscr{E}},\mathscr{F}/\lambda x.X[z := \phi]/\vec{U},V,W\\
\overset{\mathsf{app}\,\psi}{\twoheadrightarrow}\ &\Gamma,\phi,\psi;\Delta \vdash \vec{A}/\vec{\mathscr{E}},\mathscr{F}/X[x := \psi][z := \phi]/\vec{U},V,W
\end{aligned}$$
The rest of the proof, mimics the case for application described above and is omitted. □

D. Identity extension for terms

This section sketches the proof of Lemma 30. We first sketch an auxiliary lemma concerning the addition of a fresh public PCD and initial advice to bisimilar configurations.

Lemma 49. *If* $\Gamma;\Delta \vdash \vec{A}/\vec{\mathscr{E}}/M/\vec{U} \sim \vec{B}/\vec{\mathscr{F}}/N/\vec{V}$ *and* q, α *are fresh, then:* $\Gamma,\alpha;\Delta,\mathsf{pcd}\ q,\mathsf{adv}\ q=\alpha \vdash \vec{A}/\vec{\mathscr{E}}/M/\vec{U} \sim \vec{B}/\vec{\mathscr{F}}/N/\vec{V}$.

PROOF. A straightforward bisimulation proof using lemma 26. The bisimulation contains not only the configuration with the addition of α and $\mathsf{pcd}\ q,\mathsf{adv}\ q=\alpha$, but also functions and advice (at q) that can be added by the environment. The values resulting from looking up those functions are identical on both sides and state-free, and thus lemma 26 allows them to be safely ignored. □

For the proof sketch of Lemma 30 in the rest of this section, we assume that:

- $\Gamma;\Delta \vdash \vec{A}/\vec{\mathscr{E}}/M/\vec{U} \sim \vec{B}/\vec{\mathscr{F}}/N/\vec{V}$
- M,N are values.
- All names in $fn(L)$ are bound in $\Gamma;\Delta$.
- $\Gamma;\Delta \vdash \vec{A}/\vec{\mathscr{E}}/M/\vec{U}$ and $\Gamma;\Delta \vdash \vec{B}/\vec{\mathscr{F}}/N/\vec{V}$ are compatible.

The proof proceeds by structural induction on L.

Case x***,*** x ***not bound in*** Δ. Let $\mathscr{R}$ be a relation witnessing $\Gamma;\Delta \vdash \vec{A}/\vec{\mathscr{E}}/M/\vec{U} \sim \vec{B}/\vec{\mathscr{F}}/N/\vec{V}$. Consider the set O consisting of terms that are variables or applications of variables, ie. of the form $x_1,x_2,\ldots,x_1\ x_1,x_1\ x_2,\ldots.x_2\ x_1,x_2\ x_2,\ldots.$

Consider a relation $\mathscr{S}$ as follows: $\Gamma';\Delta' \vdash \vec{A}'/\vec{\mathscr{E}}'/M'/\vec{U}'$ and $\Gamma';\Delta' \vdash \vec{B}'/\vec{\mathscr{F}}'/N'/\vec{V}'$ are related by $\mathscr{S}$ if:

- $\Gamma';\Delta' \vdash \vec{A}'/\vec{\mathscr{E}}'/M'/\vec{U}'$ and $\Gamma'';\Delta' \vdash \vec{B}'/\vec{\mathscr{F}}'/N'/\vec{V}'$ are compatible.
- There exists configurations $\Gamma';\Delta' \vdash \vec{A}_1/\vec{\mathscr{E}}_1/M_1/\vec{U}_1$ and $\Gamma';\Delta' \vdash \vec{B}_1/\vec{\mathscr{F}}_1/N_1/\vec{V}_1$ related by $\mathscr{R}$, such that
    - None of the elements of O are bound in Δ'.
    - $\vec{U}'$ (resp, $\vec{V}'$) are obtained by interleaving $\vec{U}_1$ (resp. $\vec{V}_1$) with a sequence of terms from set O.
    - Either both $M' = M_1$ and $N' = N_1$ hold; or $M' = M = L$ where $L \in O$.

Proof follows since $\mathscr{S}$ is easily seen to be a bisimulation.

Case f***,*** f ***bound in*** Δ. $\Gamma;\Delta \vdash \vec{A}/\vec{\mathscr{E}}/M/\vec{U}$ and $\Gamma;\Delta \vdash \vec{B}/\vec{\mathscr{F}}/N/\vec{V}$ have no τ-reductions.

Using assumption "Public functions as values", let i be the index of the value list that has f in $\vec{U}$ and $\vec{V}$. Using transition $\mathsf{get}\ i$, we get:

$$\Gamma;\Delta \vdash \vec{A}/\vec{\mathscr{E}}/f/\vec{U} \sim \vec{B}/\vec{\mathscr{F}}/f/\vec{V}$$

Case $U\ V$. The induction hypothesis on U yields:

$$\Gamma;\Delta \vdash \vec{A}/\vec{\mathscr{E}}/U/\vec{U} \sim \vec{B}/\vec{\mathscr{F}}/U/\vec{V}$$

and hence using put:

$$\Gamma;\Delta \vdash \vec{A}/\vec{\mathscr{E}}/U/\vec{U},U \sim \vec{B}/\vec{\mathscr{F}}/U/\vec{V},U$$

Using induction hypothesis on V yields:

$$\Gamma;\Delta \vdash \vec{A}/\vec{\mathscr{E}}/V/\vec{U},U \sim \vec{B}/\vec{\mathscr{F}}/V/\vec{V},U$$

and hence using put

$$\Gamma;\Delta \vdash \vec{A}/\vec{\mathscr{E}}/V/\vec{U},U,V \sim \vec{B}/\vec{\mathscr{F}}/V/\vec{V},U,V$$

A hand-crafted bisimulation proof as used in the first base case ($L = x$, x not bound in Δ) shows that:

$$\Gamma,x_1,x_2;\Delta \vdash \vec{A}/\vec{\mathscr{E}}/x_1\ x_2/\vec{U},U,V \sim \vec{B}/\vec{\mathscr{F}}/x_1\ x_2/\vec{V},U,V$$

Consider substitution σ with empty stack and partial function given by $\{x_i \mapsto i+|\vec{U}| \mid i=1,2\}$. Using Proposition 39 yields the required result.

Case $\mathsf{pcd}\ q\,;L$. Applying lemma 49 to:

$$\Gamma;\Delta \vdash \vec{A}/\vec{\mathscr{E}}/U/\vec{U} \sim \vec{B}/\vec{\mathscr{F}}/V/\vec{V}$$

gives:

$$\Gamma,\alpha;\Delta,\mathsf{pcd}\ q,\mathsf{adv}\ q=\alpha \vdash \vec{A}/\vec{\mathscr{E}}/U/\vec{U} \sim \vec{B}/\vec{\mathscr{F}}/V/\vec{V}$$

By the induction hypothesis on L:

$$\Gamma,\alpha;\Delta,\mathsf{pcd}\ q,\mathsf{adv}\ q=\alpha \vdash \vec{A}/\vec{\mathscr{E}}/L/\vec{U} \sim \vec{B}/\vec{\mathscr{F}}/L/\vec{V}$$

A bisimulation proof establishes:

$$\Gamma,\alpha;\Delta \vdash \vec{A},\mathsf{pcd}\ q,\mathsf{adv}\ q=\alpha/\vec{\mathscr{E}}/L/\vec{U} \sim \vec{B},\mathsf{pcd}\ q,\mathsf{adv}\ q=\alpha/\vec{\mathscr{F}}/L/\vec{V}$$

And, with $W = \lambda z.\lambda x.z\,x$, a second bisimulation proof using lemma 26 yields:

$$\Gamma,\alpha;\Delta \vdash \vec{A},\mathsf{pcd}\ q,\mathsf{adv}\ q=\alpha/\vec{\mathscr{E}}/L/\vec{U},W \sim \vec{B},\mathsf{pcd}\ q,\mathsf{adv}\ q=\alpha/\vec{\mathscr{F}}/L/\vec{V},W$$

Substitution of W for α using Proposition 39 gives:

$$\Gamma;\Delta \vdash \vec{A},\mathsf{pcd}\ q,\mathsf{adv}\ q=W/\vec{\mathscr{E}}/L/\vec{U},W \sim \vec{B},\mathsf{pcd}\ q,\mathsf{adv}\ q=W/\vec{\mathscr{F}}/L/\vec{V},W$$

A final bisimulation proof shows:

$$\Gamma;\Delta \vdash \vec{A}/\vec{\mathscr{E}}/\mathsf{pcd}\ q\,;L/\vec{U} \sim \vec{B}/\vec{\mathscr{F}}/\mathsf{pcd}\ q\,;L/\vec{V}$$

Case $\mathsf{fun}\ f@q=U\,;L$. Using induction on U we deduce that:

$$\Gamma;\Delta \vdash \vec{A}/\vec{\mathscr{E}}/U/\vec{U} \sim \vec{B}/\vec{\mathscr{F}}/V/\vec{V}$$

and hence:

$$\Gamma;\Delta \vdash \vec{A}/\vec{\mathscr{E}}/U/\vec{U},U \sim \vec{B}/\vec{\mathscr{F}}/U/\vec{V},U$$

Using $\mathsf{fun}\ f@q=\phi$ on both sides, we get:

$$\Gamma,\phi;\Delta,\mathsf{fun}\ f@q=\phi \vdash \vec{A}/\vec{\mathscr{E}}/L/\vec{U},U \sim \vec{B}/\vec{\mathscr{F}}/L/\vec{V},U$$

Consider substitution σ with empty stack and partial function given by $\{\phi \mapsto 1+|\vec{U}|\}$. Using Proposition 39 yields the required result.

Case $\mathsf{adv}\ q=U\,;L$. Similar to above, but using $\mathsf{adv}\ q=\alpha$ transitions instead of $\mathsf{fun}\ f@q=\phi$.

Case $\mathsf{let}\ x=L_1\,;L_2$. Using induction on L_1 we deduce:

$$\Gamma;\Delta \vdash \vec{A}/\vec{\mathscr{E}}/L_1/\vec{U} \sim \vec{B}/\vec{\mathscr{F}}/L_1/\vec{V}$$

We need to show that:

$$\Gamma;\Delta \vdash \vec{A}/\vec{\mathscr{E}}/\mathsf{let}\ x=L1\,;L_2/\vec{U} \sim \vec{B}/\vec{\mathscr{F}}/\mathsf{let}\ x=L1\,;L_2/\vec{V}$$

L_1 is evaluated first. So, we use $\Gamma;\Delta \vdash \vec{A}/\vec{\mathscr{E}}/L_1/\vec{U} \sim \vec{B}/\vec{\mathscr{F}}/L_1/\vec{V}$ to mimic transitions between the configurations till we end up with L_1 evaluated to a value on both sides: i.e., the configurations that we have to show to be bisimilar are of the form: $\Gamma';\Delta' \vdash \vec{A}'/\vec{\mathscr{E}}/\mathsf{let}\ x=U\,;L_2/\vec{U}'$ and $\Gamma';\Delta' \vdash \vec{B}'/\vec{\mathscr{F}}/\mathsf{let}\ x=U\,;L_2/\vec{V}'$ where we know that

$$\Gamma;\Delta \vdash \vec{A}'/\vec{\mathscr{E}}/U/\vec{U}' \sim \vec{B}'/\vec{\mathscr{F}}/V/\vec{V}'$$

and hence

$$\Gamma;\Delta \vdash \vec{A}'/\vec{\mathscr{E}}/U/\vec{U}',U \sim \vec{B}'/\vec{\mathscr{F}}/V/\vec{V}',V$$

By induction hypothesis on L_2,

$$\Gamma,x;\Delta \vdash \vec{A}/\vec{\mathscr{E}}/L_2/\vec{U}',U \sim \vec{B}/\vec{\mathscr{F}}/L_2/\vec{V}',V$$

Consider substitution σ with with empty stack and partial function given by $\{x \mapsto 1+|\vec{U}|\}$. Using Proposition 39 yields the required result.

***Case* $\lambda x.L$.** Consider the relation that consists of all compatible pairs of configurations $(\Gamma;\Delta \vdash \vec{A}/\vec{\mathscr{E}}/M'/\vec{U}, \Gamma;\Delta \vdash \vec{B}/\vec{\mathscr{F}}/N'/\vec{V})$ such that there exists:

$$\Gamma,x;\Delta \vdash \vec{A}/\vec{\mathscr{E}}/L_1/\vec{U}' \sim \vec{B}/\vec{\mathscr{F}}/L_2/\vec{V}'$$

such that:

- $\vec{U}'$ (resp. $\vec{V}'$) is got from $\vec{U}$ (resp. $\vec{V}$) by deleting all occurrences of $\lambda x.L$.
- The possibilities for L_1, L_2 are as follows
  - $L_1 = M'$ and $L_2 = N'$
  - Both M', N' are values, and one of the following hold:
    - $L_1 = L_2 = \lambda x.L$
    - $L_1 = L_2 = L[x := \phi]$

The required result follows from showing that this relation is a bisimulation. The straightforward proof to show this uses inductive hypothesis on L at all configurations having $L_1 = L_2 = L[x := \phi]$ in the active term position.

Inclusion of identical values and identical evaluation contexts. Since the first time when values from the value list (or contexts from the context list) can be moved into active position is when the term in the active position has become a value, and hence in the realm of applicability of Lemma 30, the addition of identical contexts and values can be done in slightly more general situations.

Corollary 50 (to Lemma 30). *If:*

- $\Gamma;\Delta \vdash \vec{A}/\vec{\mathscr{E}}/M/\vec{U} \sim \vec{B}/\vec{\mathscr{F}}/N/\vec{V}$
- *All names in $fn(U), \mathscr{E}, \mathscr{E}'$ are bound in* Γ, Δ
- $\Gamma;\Delta \vdash \vec{A}/\vec{\mathscr{E}}/M/\vec{U}$ *and* $\Gamma;\Delta \vdash \vec{B}/\vec{\mathscr{F}}/N/\vec{V}$ *are compatible.*

then:

$$\Gamma;\Delta \vdash \vec{A}/\mathscr{E}',\vec{\mathscr{E}},\mathscr{E}/M/\vec{U},U \sim \vec{B}/\mathscr{E}',\vec{\mathscr{F}},\mathscr{E}/N/\vec{V},U \qquad \square$$

Corollary 50 is used in the proof of Lemma 31 given in Section E.

E. Inclusion of identical contexts

In this section we sketch the proof of Lemma 31. The proof relies on the following auxiliary lemma that uses initial advice in Δ of the form $\mathsf{adv}\ q = \beta$ to add further advice of the form $\mathsf{adv}\ q = \alpha$ or $\mathsf{adv}\ q = W$. The new advice appears after $\mathsf{adv}\ q = \beta$ but before all other advice on q.

Lemma 51. *If:*

- $\Gamma;\Delta \vdash \vec{A}/\vec{\mathscr{E}}/M/\vec{U} \sim \vec{B}/\vec{\mathscr{F}}/N/\vec{V}$
- $\Gamma;\Delta \vdash \vec{A}/\vec{\mathscr{E}}/M/\vec{U}$ *and* $\Gamma;\Delta \vdash \vec{B}/\vec{\mathscr{F}}/N/\vec{V}$ *are compatible.*
- $\alpha \in \Gamma$ *and* $\mathsf{pcd}\ q \in \Delta$.

Then:

$$\Gamma;\Delta \vdash \mathsf{adv}\ q = \alpha\,;\vec{A}/\vec{\mathscr{E}}/M/\vec{U} \sim \mathsf{adv}\ q = \alpha\,;\vec{B}/\vec{\mathscr{F}}/N/\vec{V}$$

Moreover, if $\mathsf{lam\ p.\ U}$ *is well-formed over* $\Gamma;\Delta$, *then:*

$$\Gamma;\Delta \vdash \mathsf{adv}\ q = \lambda z.U\,;\vec{A}/\vec{\mathscr{E}}/M/\vec{U} \sim \mathsf{adv}\ q = \lambda z.U\,;\vec{B}/\vec{\mathscr{F}}/N/\vec{V}$$

PROOF SKETCH. The proof for the first part proceeds by moving the rightmost advice in Δ of the form $\mathsf{adv}\ q = \beta$ into the private declarations $\vec{A}$, $\vec{B}$ (a bisimulation proof), then using inclusion of identical values (Corollary 50) and substitution (Proposition 39) to replace $\mathsf{adv}\ q = \beta$ with $\mathsf{adv}\ q = \lambda z.\lambda x.\alpha\texttt{<}\gamma\texttt{<}z\texttt{>>}\,x$, where γ is fresh. A second bisimulation proof shows that the above advice is equivalent to $\mathsf{adv}\ q = \gamma$; $\mathsf{adv}\ q = \alpha$. A final bisimulation proof shows that γ can be renamed to β (which was substituted away) and then moved back into the public declaration list to recover the original Δ. The second part follows from the first by inclusion of identical values (Corollary 50) $\lambda z.U$ and then substituting (Proposition 39) $\lambda z.U$ for α. □

For the sketch of Lemma 31, we assume without loss of generality that in $\Gamma;\Delta \vdash \vec{A}/\vec{\mathscr{E}}/()/\vec{W}$, that is to be added to $\Gamma;\Delta \vdash \cdot/\cdot/M/\vec{U} \sim \cdot/\cdot/N/\vec{V}$, the declarations $\vec{A}$ have the form $\vec{A}_1,\vec{A}_2,\vec{A}_3$ where ($\vec{p}$ and $\vec{r}$ may be bound in $\vec{A}_1$ or Δ):

$$\begin{aligned}\vec{A}_1 &= \mathsf{pcd}\ \vec{q}\\ \vec{A}_2 &= \mathsf{fun}\ \vec{f}\,\texttt{@}\,\vec{p} = \vec{W}'\\ \vec{A}_3 &= \mathsf{adv}\ \vec{r} = \vec{W}''\end{aligned}$$

Using Lemma 49, $\vec{A}_1$ can be added along with initial advice $\vec{A}_4 = \mathsf{adv}\ \vec{q} = \vec{\alpha}$:

$$\Gamma,\vec{\alpha};\Delta,\vec{A}_1,\vec{A}_4 \vdash \cdot/\cdot/M/\vec{U} \sim \cdot/\cdot/N/\vec{V}$$

Function definitions with symbolic bodies can be added using $\mathsf{fun}\ f\texttt{@}q = x$ transitions since the PCDs $\vec{q}$ are public, so with $\vec{A}_5 = \mathsf{fun}\ \vec{f}\,\texttt{@}\,\vec{p} = x$:

$$\Gamma,\vec{\alpha},\vec{x};\Delta,\vec{A}_1,\vec{A}_4,\vec{A}_5 \vdash \cdot/\cdot/M/\vec{U},\vec{f} \sim \cdot/\cdot/N/\vec{V},\vec{f}$$

Using Corollary 50 to add values and evaluation contexts, we have:

$$\Gamma,\vec{\alpha},\vec{x};\Delta,\vec{A}_1,\vec{A}_4,\vec{A}_5 \vdash \cdot/\vec{\mathscr{E}}/M/\vec{U},\vec{f},\vec{W},\vec{W}' \sim \cdot/\vec{\mathscr{E}}/N/\vec{V},\vec{f},\vec{W},\vec{W}'$$

Using Lemma 51 in an induction working from the right to the left of $\vec{A}_3$, we can add the advice $\vec{A}_3$ to both sides:

$$\Gamma,\vec{\alpha},\vec{x};\Delta,\vec{A}_1,\vec{A}_4,\vec{A}_5 \vdash \vec{A}_3/\vec{\mathscr{E}}/M/\vec{U},\vec{f},\vec{W},\vec{W}' \sim \vec{A}_3/\vec{\mathscr{E}}/N/\vec{V},\vec{f},\vec{W},\vec{W}'$$

The function declarations $\vec{A}_5$ can be moved to the private declaration lists and the $\vec{f}$ value list removed by a bisimulation proof. Then the real function bodies $\vec{W}'$ can be substituted for $\vec{x}$ to recover $\vec{A}_2$:

$$\Gamma,\vec{\alpha};\Delta,\vec{A}_1,\vec{A}_4 \vdash \vec{A}_2,\vec{A}_3/\vec{\mathscr{E}}/M/\vec{U},\vec{W} \sim \vec{A}_2,\vec{A}_3/\vec{\mathscr{E}}/N/\vec{V},\vec{W}$$

Finally, the $\vec{A}_1$ and $\vec{A}_4$ can be moved to the private declaration lists by a bisimulation proof, then the advice $\vec{A}_4$ eliminated by substitution:

$$\Gamma;\Delta \vdash \vec{A}_1,\vec{A}_2,\vec{A}_3/\vec{\mathscr{E}}/M/\vec{U},\vec{W} \sim \vec{A}_1,\vec{A}_2,\vec{A}_3/\vec{\mathscr{E}}/N/\vec{V},\vec{W}$$

This completes the proof.

F. Bisimulation is a congruence (Proof of lemma 32)

This section contains the proof that bisimulation is a congruence.

Application. Let $U_1 \sim U_1'$ and $U_2 \sim U_2'$. We need to show that $U_1\ U_2 \sim U_1'\ U_2'$. From:

$$\Gamma;\Delta \vdash \cdot/\cdot/U_2/\vec{g} \sim \cdot/\cdot/U_2'/\vec{g}$$

we deduce:

$$\Gamma;\Delta \vdash \cdot/\cdot/U_2/\vec{g},U_2 \sim \cdot/\cdot/U_2'/\vec{g},U_2'$$

From this, we deduce:

$$\Gamma,x_1,x_2;\Delta \vdash \cdot/\cdot/()/\vec{g},U_2 \sim \cdot/\cdot/()/\vec{g},U_2'$$

and using Lemma 30:

$$\Gamma,x_1,x_2;\Delta \vdash \cdot/\cdot/x_1\ x_2/\vec{g},U_2 \sim \cdot/\cdot/x_1\ x_2/\vec{g},U_2'$$

Now, using Corollary 50 yields

$$\Gamma,x_1,x_2;\Delta \vdash \cdot/\cdot/x_1\ x_2/\vec{g},U_2 \sim \cdot/\cdot/x_1\ x_2/\vec{g},U_2'$$

Similarly, from $\Gamma;\Delta \vdash \cdot/\cdot/U_1/ \sim \cdot/\cdot/U_1'/\cdot$ we get

$$\Gamma,x_1,x_2;\Delta \vdash \cdot/\cdot/x_1\ x_2/\vec{g},U_1 \sim \cdot/\cdot/x_1\ x_2/\vec{g},U_1'$$

Combining:

$$\Gamma, x_1, x_2; \Delta \vdash \cdot/\cdot/x_1\, x_2/\vec{g}, U_1, U_2 \sim \cdot/\cdot/x_1\, x_2/\vec{g}, U_1', U_2'$$

Consider the substitution σ with empty stack and partial function given by $\{x_i \mapsto |\vec{g}| + i\}$. Proposition 39 yields

$$\Gamma; \Delta \vdash [\cdot/\cdot/x_1\, x_2/\vec{g}, U_1, U_2]_\sigma \sim [\cdot/\cdot/x_1\, x_2/\vec{g}, U_1', U_2']_\sigma$$

and finishes the proof.

Function declaration. Let $U \sim U'$ and $M \sim M'$. We need to show that $\mathsf{fun}\ f@q = U\,;\, M \sim \mathsf{fun}\ f@q = U'\,;\, M'$.

Using Lemma 31, and $\Gamma, \phi; \Delta \vdash \cdot/\cdot/M/\vec{g} \sim \cdot/\cdot/M'/\vec{g}$ we deduce that:

$$\Gamma, \phi; \Delta \vdash \mathsf{fun}\ f@q = \phi/\cdot/()/\vec{g} \sim \mathsf{fun}\ f@q = \phi/\cdot/M/\vec{g}$$

Using Corollary 50:

$$\Gamma, \phi; \Delta \vdash \mathsf{fun}\ f@q = \phi/\cdot/()/\vec{g}, U \sim \mathsf{fun}\ f@q = \phi/\cdot/M'/\vec{g}, U'$$

Consider the substitution σ with empty stack and partial function given by $\{\phi \mapsto |\vec{g}| + 1\}$. Proposition 39 finishes the proof.

Advice declaration. As above.

Lambda abstraction. Given $L \sim L'$, consider $\lambda x.L$ and $\lambda x.L'$. The proof that these terms are bisimilar proceeds by a direct bisimulation argument.

Define a bisimulation candidate $\mathscr{R}$ as follows: $\Gamma; \Delta \vdash \vec{A}/\vec{\mathscr{E}}/M/\vec{U}$ and $\Gamma; \Delta \vdash \vec{B}/\vec{\mathscr{F}}/N/\vec{V}$ are related by $\mathscr{R}$ iff the following holds:

There exists $\vec{U}', \vec{V}'$ such that:

- $\vec{U}$ (resp. $\vec{V}$) is got by deleting all occurrences of $\lambda x.L$ (resp. $\lambda x.L'$) from $\vec{U}'$ (resp. $\vec{U}$) that are at identical indices in $\vec{U}, \vec{U}'$.
- One of the following holds:
  - Either $\Gamma; \Delta \vdash \vec{A}/\mathscr{E}/M/\vec{U}' \sim \vec{B}/\mathscr{F}/N/\vec{V}'$,
  - Or, $M = \lambda x.L$, $N = \lambda x.L'$ and for some U, V: $\Gamma; \Delta \vdash \vec{A}/\mathscr{E}/U/\vec{U}' \sim \vec{B}/\mathscr{F}/V/\vec{V}'$

The key case to consider is an $\mathsf{app}\ \phi$ transition in the second case above leading to configurations: $\Gamma, \phi; \Delta \vdash \vec{A}/\vec{\mathscr{E}}/L[x := \phi]/\vec{U}$ and $\Gamma, \phi; \Delta \vdash \vec{A}/\vec{\mathscr{E}}/L'[x := \phi]/\vec{U}$ These configurations are proved bisimilar by Lemma 31 on: $\Gamma, \phi; \Delta \vdash \vec{A}/\mathscr{E}/U/\vec{U}' \sim \vec{B}/\mathscr{F}/V/\vec{V}'$. So, $\Gamma, \phi; \Delta \vdash \vec{A}/\vec{\mathscr{E}}/L[x := \phi]/\vec{U}$ and $\Gamma, \phi; \Delta \vdash \vec{A}/\vec{\mathscr{E}}/L'[x := \phi]/\vec{U}$ are related by the candidate bisimulation relation.

Let. Given $L_1 \sim L_1'$ and $L_2 \sim L_2'$ we must show that $\mathsf{let}\ x = L_1\,;\, L_2 \sim \mathsf{let}\ x = L_1'\,;\, L_2'$. We do this in two stages and then use the transitivity of $\sim$:

$$\begin{aligned}\mathsf{let}\ x = L_1\,;\, L_2 &\sim \mathsf{let}\ x = L_1'\,;\, L_2\\ \mathsf{let}\ x = L_1'\,;\, L_2 &\sim \mathsf{let}\ x = L_1'\,;\, L_2'\end{aligned}$$

For the first stage, we use the bisimulation for $L_1 \sim L_1'$ to construct a bisimulation for $\mathsf{let}\ x = L_1\,;\, L_2 \sim \mathsf{let}\ x = L_1'\,;\, L_2$. The interesting case is when L_1 and L_1' are both values, in which case we can save those two values into the value lists, use Lemma 30 to add L_2 in the term position in both configurations, and then apply substitution (Proposition 39) to establish the relationship between configurations with $L_2[x := L_1]$ and $L_2[x := L_1']$ in the term positions.

For the second stage, a bisimulation proof establishes:

$$\begin{aligned}\mathsf{let}\ x = L_1'\,;\, L_2 &\sim \mathsf{let}\ x = L_1'\,;\, (\lambda x.L_2)\ x\\ \mathsf{let}\ x = L_1'\,;\, L_2' &\sim \mathsf{let}\ x = L_1'\,;\, (\lambda x.L_2')\ x\end{aligned}$$

Now we know from the previous case that $L_2 \sim L_2'$ implies $\lambda x.L_2 \sim \lambda x.L_2'$. After saving those values into the value lists, use Lemma 30 to add $\mathsf{let}\ x = L_1'\,;\, y\ x$ in the term position in both configurations. Substitution (Proposition 39) establishes the relationship between configurations with $(\mathsf{let}\ x = L_1'\,;\, y\ x)[y := \lambda x.L_2]$ and $(\mathsf{let}\ x = L_1'\,;\, y\ x)[y := \lambda x.L_2']$ in the term positions.

primitive pointcut declaration. Given $L \sim L'$ we must show that $\mathsf{pcd}\ q\,;\, L \sim \mathsf{pcd}\ q\,;\, L'$. From $L \sim L'$ we know:

$$\Gamma, \alpha; \Delta, \mathsf{pcd}\ q, \mathsf{adv}\ q = \alpha \vdash \cdot/\cdot/L/\cdot \sim \cdot/\cdot/L'/\cdot$$

A straightforward bisimulation shows that the declarations $\mathsf{pcd}\ q$ and $\mathsf{adv}\ q = \alpha$ can be moved to the front of the terms L and L'. Then the advice $\mathsf{adv}\ q = \alpha$ can be eliminated by substitution of $\lambda z.\lambda x.z\,x$ for α and a bisimulation proof, to leave:

$$\Gamma; \Delta \vdash \cdot/\cdot/\mathsf{pcd}\ q\,;\, L/\cdot \sim \cdot/\cdot/\mathsf{pcd}\ q\,;\, L'/\cdot$$

G. Completeness

The proof of Proposition 41a is a straightforward induction on the length of the trace. The proof of Proposition 41b is a rather tedious analysis of the commutativity of various labels and the resulting configurations. The essential observation is the following.

Lemma 52. *The following categories of LTS states are disjoint:*

- *Write* $\Gamma; \Delta \vdash \mathbf{M} \Uparrow$ *if* $\Gamma; \Delta \vdash \mathbf{M} \rightarrow^{\omega}$.
- *Write* $\Gamma; \Delta \vdash \mathbf{M} \Downarrow \textsc{term}$ *if* $\Gamma; \Delta \vdash \mathbf{M} \xrightarrow{\kappa}\!\!\!\!\!\rightarrow$ *for* $\kappa \in \{\mathsf{fcall}, \mathsf{acall}\}$.
- *Write* $\Gamma; \Delta \vdash \mathbf{M} \Downarrow \textsc{ctxt}$ *if* $\Gamma; \Delta \vdash \mathbf{M} \xrightarrow{\kappa}\!\!\!\!\!\rightarrow$ *for* $\kappa \in \{\mathsf{put}, \mathsf{get}, \mathsf{ret}, \mathsf{app}, \mathsf{fun}, \mathsf{adv}\}$. □

In rest of this appendix we describe the strategy for building contexts to satisfy the requirements of Propositions 42 and 43. Since we are concerned only with normal traces, we adopt the following abbreviations,

$$\begin{aligned}\mathsf{getapp}\ i\ \phi &\triangleq \mathsf{get}\ i, \mathsf{app}\ \phi\\ \mathsf{fcallput}\ \phi &\triangleq \mathsf{fcall}\ \phi, \mathsf{put}\\ \mathsf{acallput}\ \alpha &\triangleq \mathsf{acall}\ \alpha, \mathsf{put}\end{aligned}$$

with completed normal traces formed by the following grammar.

$$\begin{array}{rcl}\textsc{start} & ::= & \textsc{term}^*, \mathsf{put}, \textsc{ctxt}^*\\ \textsc{term} & ::= & \mathsf{fcallput}\ \phi, \textsc{ctxt}^*, \mathsf{ret}\ \psi \mid \mathsf{acallput}\ \alpha, \textsc{ctxt}^*, \mathsf{ret}\ \psi\\ \textsc{ctxt} & ::= & \mathsf{getapp}\ i\ \phi, \textsc{term}^*, \mathsf{put} \mid \mathsf{fun}\, f@q = \phi \mid \mathsf{adv}\ q = \alpha\end{array}$$

We use the phrase *call label* for labels fcallput and acallput. We use the phrase *context label* for labels getapp, put, fun and adv. Recalling the definition on page 15, let $Z(\vec{\mathscr{E}})(M) = \big(Z(\vec{\mathscr{E}})([-])\big)[M]$.

Fix s, t, Γ, Δ and $\mathbf{M} = \vec{A}/\vec{\mathscr{E}}/M/\vec{U}$. We show how to build term

$$\mathbb{C}_t^s[\Gamma; \Delta \vdash \mathbf{M}] = \mathscr{C}[Z(\vec{\mathscr{G}})(M)].$$

We refer to as $\mathscr{C}$ as *the context*. We refer to $\vec{\mathscr{G}}$ as *the stack*; this is an interleaving of the given $\vec{\mathscr{E}}$, the *term stack*, and a defined $\vec{\mathscr{F}}$, the *context stack*.

The context includes function declarations for each name in $\Gamma \cup dn(s,t)$, as well as the declarations Δ and $\vec{A}$. The context also includes the following mutable structures.

- The vector values keeps track of the stored values in a configuration. $put(\mathsf{values}, V)$ pushes V onto the end of the vector; $get(\mathsf{values}, i)$ returns the i^{th} value from the vector; these functions have standard encodings in the lambda calculus with references. In $\mathbb{C}_{\_}^{\_}[\_/\_/\_/U_1, \ldots, U_n]$, $get(\mathsf{values}, i)$ returns U_i.
- The reference callcount holds the number of call labels occurring in t; thus $!\mathsf{callcount}$ is 0 in $\mathbb{C}_\varepsilon^s[-]$.

The functions for $\Gamma \cup dn(s,t)$ are unadvisable, i.e., declared at a fresh primitive pointcut. (Symbolic advice and functions are treated similarly; to simplify the presentation, we abuse notation to allow function declarations at symbolic advice names.) The basic structure of a function body is a case structure on callcount.

$$\begin{aligned}&\mathsf{fun}\ \phi = \lambda \mathsf{x}.\mathsf{callcount{-}{-}}\,;\, put(\mathsf{values}, \mathsf{x})\,;\\ &\qquad\qquad \mathsf{case}\ !\mathsf{callcount}\ \mathsf{of}\ \cdots\ \mathsf{default} \Rightarrow \Omega\\ &\mathsf{fun}\ \alpha = \lambda \mathsf{z}.\lambda \mathsf{x}.\mathsf{callcount{-}{-}}\,;\, put(\mathsf{values}, \mathsf{z})\,;\, put(\mathsf{values}, \mathsf{x})\,;\\ &\qquad\qquad \mathsf{case}\ !\mathsf{callcount}\ \mathsf{of}\ \cdots\ \mathsf{default} \Rightarrow \Omega\end{aligned}$$

We generate additional cases for these function bodies by working through the trace s,t. The context stacks $\vec{\mathscr{F}}$, mentioned above, are "suffixes" of these function bodies that have been called (in s) but have not yet returned (reading s forward); the context stack includes the actions yet to be performed by these functions (generated by analyzing t). The term stack $\vec{\mathscr{E}}$ includes the suffixes of functions interrupted by a call label; the context stack $\vec{\mathscr{F}}$ includes the suffixes of functions interrupted by a getapp. Call labels that do not have matching returns in s,t will end in Ω, both in the function declaration and in the context stack.

The last element of the trace is treated specially, as initialization. From Proposition 41, we can assume that the last element is a call label. Suppose it is $\mathsf{fcallput}\ \phi$ ($\mathsf{acallput}$ is similar). Then we add case "$0 \Rightarrow \mathsf{signal}\ \mathsf{()}$" to the definition of ϕ, and the definition becomes

$$\begin{aligned}&\mathsf{fun}\ \phi = \lambda x.\mathsf{callcount{-}{-}}\,;\, \mathit{put}(\mathsf{values}, x)\,;\\ &\qquad\qquad \mathsf{case}\ \mathsf{!callcount}\ \mathsf{of}\ \cdots\ 0 \Rightarrow \mathsf{signal}\ \mathsf{()}\,;\ \mathsf{default} \Rightarrow \Omega.\end{aligned}$$

Now that we have initialized the function declaration, we can begin to generate new cases by working *backwards* through s,t. We generate these using a stack of contexts, called the *generating stack*. When we reach the beginning of t (before getting to the end of s), we record the generating stack, which becomes the context stack $\vec{\mathscr{F}}$. We continue the backwards processing of s to generate the remaining function body cases (so that function bodies do not change from $\mathbb{C}^{s}_{\kappa,t}[-]$ to $\mathbb{C}^{s,\kappa}_{t}[-]$).

Initially the generating stack contains a context "$[-]\,;\,\Omega$" for every un-returned call label in s,t. Labels are processed as follows:

- We push a new context "$[-]\,;\,\psi$" onto the generating stack for every label $\mathsf{ret}\ \psi$ that we process.
- We pop a context $\mathscr{G}$ and add a case "$\Rightarrow \mathscr{G}[\mathsf{()}]$" to ϕ for every label $\mathsf{fcallput}\ \phi$, and similarly for $\mathsf{acallput}\ \alpha$. The guard on the case is derived by counting the number of call labels that have been processed.
- As we process context labels, we replace the top context of the generating stack $\mathscr{G}$ with a new one, as dictated by the following table. (We use the name y for the variable holding the return value from all calls to term functions; using a single variable name simplifies code generation.)

$\mathsf{getapp}\ i\ \phi$	$\mathsf{let}\ \mathsf{y} = (\mathit{get}(\mathsf{values}, i))\ \phi\,;\mathscr{G}$
put	$\mathit{put}(\mathsf{values}, \mathsf{y})\,;\mathscr{G}$
$\mathsf{fun}\, f\mathsf{@}q = \phi$	$\mathsf{fun}\ f\mathsf{@}q = \phi\,;\mathscr{G}$
$\mathsf{adv}\ q = \alpha$	$\mathsf{adv}\ q = \alpha\,;\mathscr{G}$

This strategy generates contexts that satisfy the requirements. We elide further details.