Stack Inspection: Theory and Variants

Cédric Fournet Andrew D. Gordon

December 2001

Technical Report
MSR–TR–2001–103

Microsoft Research
Microsoft Corporation
One Microsoft Way
Redmond, WA 98052

A shortened version of this paper appears in the proceedings of the 29th ACM Symposium on Principles of Programming Languages, Portland, Oregon, January 2002.

Stack Inspection: Theory and Variants

Cédric Fournet   Andrew D. Gordon

December 2001

Abstract

Stack inspection is a security mechanism implemented in runtimes such as the JVM and the CLR to accommodate components with diverse levels of trust. Although stack inspection enables the fine-grained expression of access control policies, it has rather a complex and subtle semantics. We present a formal semantics and an equational theory to explain how stack inspection affects program behaviour and code optimisations. We discuss the security properties enforced by stack inspection, and also consider variants with stronger, simpler properties.

Contents

1 Security By Stack Inspection?

Stack inspection is a software-based access control mechanism. Its purpose is to allow components with diverse origins to share the same runtime and access its resources in a controlled manner, according to their respective levels of trust. It is a key security mechanism in typed runtime environments such as the JVM [19, 10] and the CLR [8] that support distributed computation based on mobile code. It enables the fine-grained expression of access control policies, and hence is more liberal and flexible than a strict sandboxing mechanism. It has received much attention in the literature [5, 7, 9, 16, 17, 28, 30, 31].

Now, stack inspection is often marketed as a feature that:

(1) allows security-conscious developers, such as the authors of trusted libraries, to express their security requirements easily and precisely, and

(2) can safely be ignored by everyone else.

We began this work with the realisation that these two claims are problematic and need careful qualification:

(1) The first problem is that stack inspection, as its name suggests, is usually thought of in specific, low level terms. It seems to be remarkably hard to give a general account of what actually is guaranteed by stack inspection. Hence, it can be difficult to assess whether it is correctly implementing a higher level security policy. Besides, certain higher-order features, such as threads and method delegation, need careful treatment.

(2) The second problem is that stack inspection profoundly affects the semantics of all programs. In particular, it invalidates a wide variety of program transformations, such as inlining and tail call optimisations.

We address these two problems in the setting of a λ-calculus model [28, 30] of stack inspection. We formally state some of the guarantees given by stack inspection and suggest variations of stack inspection with stronger, simpler properties. We develop an equational theory of stack inspection that helps to highlight its subtle effects and also justifies certain transformations.

Having outlined our motivations, we next review the ideas of stack inspection. Then we elaborate on the difficulties it raises. We close this introductory section by describing our contributions in more detail.

An Outline of Stack Inspection The situation addressed by stack inspection mechanisms is as follows. Applications are collections of components, possibly compiled from different languages, that share the same runtime. Components have a variety of origins, more or less trusted. Some mechanism—such as scoping or typing rules—prevents direct access from untrusted components to resources protected by trusted components. Still, untrusted code may call trusted code, and the other way round.

We express access to different kinds of protected resources in terms of permissions, such as "may perform screen I/O" or "may perform file I/O". A configurable policy determines the access rights available to each component given evidence of its origin, that is, where it came from and who wrote it. The access rights are simply a set of allowed permissions. Here we abstract from the details of policy and evidence, and simply refer to this set of permissions itself as the *principal* that owns the code.

For example, a *System* principal might consist of all permissions, whereas an *Applet* principal might consist of a very limited set of rights, including "may perform screen I/O," but not including "may perform file I/O."

During compilation and loading, but before execution, each function or method body securely receives an annotation (here called a *frame*) specifying the principal owning it.

During execution, when trusted code is about to access some protected resource, it invokes the stack inspection primitive (here called *test*) to determine whether the appropriate permission is present. A first requirement is that its immediate caller be statically annotated with the permission. In fact, the basic algorithm is to inspect the whole call stack to ensure that indirect callers as well as the immediate caller are all statically annotated with the permission. The purpose of inspecting the whole stack is to prevent the possibility that untrusted code lacking the permission could somehow cause an indirect call to a trusted function that itself accesses the resource—an instance of the Confused Deputy attack [14, 31]. Abstractly, this basic algorithm computes a compound principal whose access rights are the intersection of the access rights of all the principals on the stack, and then checks whether this compound principal has the appropriate permission.

The full algorithm allows trusted code to invoke a primitive (here called *grant*) to override the inspection of its callers for some permissions and hence to assert responsibility for use of those permissions in every context.

For example, suppose some *System*-owned function implementing screen I/O needs to write into a log-file, for performance debugging purposes, and hence needs the "may perform file I/O" permission. If this function is called by an *Applet*-owned function, access to the log-file is denied because *Applet* does not have the "may perform file I/O" permission. The function would override inspection of its callers for the "may perform file I/O" permission so that the file write is allowed even if its caller is *Applet*-owned.

Limitations of Stack Inspection The point of stack inspection is to allow a component to protect its resources in spite of interactions with other components of diverse origin. The permissions authorised by stack inspection (the *test* primitive) are determined by a clever algorithm, outlined above, that scans control stacks on demand. The authorisation decision depends solely on the current series of nested calls. Therefore, it does not depend on other kinds of interaction between software components. Such interactions include, for instance, the use of results returned by untrusted code, mutable state, inheritance, side effects, concurrency, and dynamic loading. These interactions are commonplace, and their impact on security must be addressed independently. In the formalism of this paper, the problem appears when trusted and untrusted code exchange functions as values.

As a result, a careful analysis of any code that explicitly manipulates permissions may not in fact yield any strong guarantee (although it may reveal security problems). This significantly restricts the scope of stack inspection, in isolation. On the other hand, stricter mechanisms, based for instance on systematic flow analyses, yield stronger guarantees, but may be harder and more costly to implement and to use.

Living in Harmony with Stack Inspection Assuming the target platform features stack inspection, the programmer faces two conflicting problems. Some untrusted component may take advantage of the programmer's code to breach security—this is potentially quite bad, but it is hard to characterize. A more immediate concern is that some permission may be missing in the middle of a computation involving this code (even if the code statically has those permissions); typically, an unexpected security exception is raised—this complies with the policy, but remains undesirable.

In addition, the compiler writer must deal with a specific problem: stack inspection makes the control stack observable, hence the actual runtime stack must agree with the stack as it appears to the source program. This correctness issue hinders any program transformation that changes the structure of the stack. (A prerequisite to using stack inspection is to make the control stack apparent in the source language. This may be troublesome in declarative languages like, for instance, Haskell or Mercury.)

There are two further problems. Programmers and compiler writers may be concerned about the runtime costs incurred by stack inspection and by other operations on permissions. Besides, they have little control of the security policy that will be applied to their code, and must program without knowing exactly which static permissions their code will receive.

Contributions of the Paper We discuss stack inspection in the precise and abstract setting of λ_{sec}, a call-by-value λ-calculus [27] with notions of permissions, principals, and stack inspection, introduced by Pottier, Skalka, and Smith [28, 30]. Previous studies of λ_{sec} focus on type systems for checking information about permissions. Here, we use the untyped λ_{sec}-calculus as a minimal formalism for investigating the runtime behaviour of stack inspection.

- We present the first equational theory for a calculus of stack inspection. We prove soundness of a primitive set of equations with respect to Morris-style contextual equivalence (Theorem 1), and completeness with respect to the reduction semantics (Theorem 2).

- To obtain a co-inductive proof technique to justify our equational theory, we recast Abramsky's applicative bisimilarity for the λ_{sec}-calculus. We show that bisimilarity is a congruence by Howe's method (Theorem 3). Hence, we prove that bisimilarity in fact equals contextual equivalence (Theorem 4), admitting bisimulation-style proofs of program equivalence.

- Applications of the equational theory include justification of compiler transformations—such as elimination of redundant frames and tests—and programming techniques—such as performing security tests eagerly to speed up stack inspection. Moreover, we use the equational theory to discuss the effect of stack inspection on inlining and tail call optimisations.

- We explain how stack inspection can be understood as a form of data dependency analysis and—relying in part on our equational theory—discuss somewhat limited properties guaranteed by stack inspection (Theorems 5 and 6). We describe how stack inspection only partly fulfils its intent with respect to the higher-order features of λ_{sec}. (Similar limitations arise in practice with side-effects, exception handling, and method delegates). We give precise rules for how stack inspection could be amended to overcome these limitations, and formalize guarantees provided by the amended semantics (Theorems 7 and 8).

Although the technical contributions of this paper are phrased in terms of a formalism, the formalism is not an end in itself: the development is inspired by a study of stack inspection in the CLR, in relation to the compilation of functional languages. It also suggests potential improvements and validates optimisations performed by its JIT compiler.

Contents In Section 2, we recall (a variant of) λ_{sec} and give its operational semantics, as a security-indexed reduction semantics. In Section 3, we illustrate stack inspection and its limitations in a series of examples. In Section 4, we define contextual equivalence in the presence of stack inspection, we present our equational theory, and we use applicative bisimilarity to prove the soundness of the theory with respect to contextual equivalence. In Section 5, we study simple program transformations. In Section 6, we discuss the security guarantees provided by stack inspection, and compare it to simpler, stricter mechanisms that keep track of dependencies during evaluation. In Section 7, we discuss related works and conclude.

There are three appendixes. Appendix A presents a semantics for our variant of λ_{sec} in terms of explicit stack inspection, and shows it is equivalent to the semantics of Section 2. Appendix B presents a security-indexed big-step evaluation semantics for λ_{sec}, and shows it equivalent is to the small-step reduction semantics of Section 2. This alternative semantics is useful in Appendix C, which contains proofs of all the results stated without proof in the main part of the paper.

2 A Calculus of Stack Inspection

We describe the syntax and informal semantics of a version of the λ_{sec}-calculus [28], present an operational semantics, and explain how we use λ_{sec} to model loading components of diverse origins.

2.1 Syntax and Informal Semantics

We assume there is a set $\mathcal{P}$ of atomic *permissions*. Let a *principal* be a subset of $\mathcal{P}$.

Permissions and principals

$p, q \in \mathcal{P}$	permission
$R, S, T, D \subseteq \mathcal{P}$	principal: a set of permissions

This presentation is a little more abstract than the original λ_{sec}, where a principal is a name, and a function maps each principal to its set of permissions. For our purposes we may as well eliminate this indirection.

Expressions include variables, functions, and applications, as usual, plus constructs for stack inspection. A framed expression $R[e]$ is the expression e framed with the principal R; the principal represents permissions conferred on the code e given its origin. We have *grant* and *test* expressions as discussed in the introduction. Finally, *fail* is an exception, used, for example, to indicate a security failure.

Expressions

$e, f ::=$	expression
$\quad x$	variable
$\quad \lambda x.e$	function
$\quad e\ f$	application
$\quad R[e]$	framed expression
$\quad \mathit{grant}\ R\ \mathit{in}\ e$	permission grant
$\quad \mathit{test}\ R\ \mathit{then}\ e\ \mathit{else}\ f$	permission test
$\quad \mathit{fail}$	abnormal termination

Abstractly, the behaviour of an expression depends on two sets of permissions: the *static permissions*, S, and the *dynamic permissions*, D, with $D \subseteq S$. The static permissions are the principal in the nearest enclosing frame, an upper bound on the permissions available to the expression. The dynamic permissions are those effectively available at runtime; they represent what can be retrieved by a stack inspection. We consider a top-level expression to be fully trusted, so take the static and dynamic permission sets to be $\mathcal{P}$ initially.

The expression $R[e]$ behaves as e, but with static permissions set to R, and dynamic permissions intersected with R. The expression $\mathit{grant}\ R\ \mathit{in}\ e$ behaves as e, but with the dynamic permissions extended with all the static permissions that also appear in R. The expression $\mathit{test}\ R\ \mathit{then}\ e\ \mathit{else}\ f$ behaves as e if all permissions in R are dynamic permissions, but otherwise behaves as f. The other expressions do not inspect or modify the permission sets. They behave as in a standard call-by-value λ-calculus with a single uncatchable exception *fail* and left-to-right evaluation order.

We follow some standard syntactic conventions. In a function $\lambda x.e$, the variable x is bound, with scope e. We write $\mathit{fv}(e)$ for the set of variables occurring free in e, and write $e\{x \leftarrow e'\}$ for the outcome of a capture-avoiding substitution of the expression e' for each free occurrence of the variable x in e. An expression e is *closed* when $\mathit{fv}(e) = \varnothing$. We identify expressions up to capture-avoiding renamings of bound variables, that is, $\lambda x.e = \lambda x'.(e\{x \leftarrow x'\})$ if $x' \notin \mathit{fv}(e)$.

We introduce notions of *values* and *outcomes*. A value is a function or a variable; values represent the formal and actual arguments passed to a function. An outcome is a value or the exception *fail*; outcomes are fully-reduced expressions.

Values and outcomes

$u, v ::= x \mid \lambda x.e$	value
$o ::= v \mid \mathit{fail}$	outcome

The first four of the following abbreviations are fairly standard. The fifth defines an arbitrary value *ok* to indicate normal termination in our examples. The last, *check p for e*, represents a common idiom, a primitive in earlier formulations of λ_{sec} [28, 30]: test whether a single permission p is effectively available; if so, run e; otherwise, raise a security exception.

Abbreviations

$\lambda x_1 \cdots x_n.e \triangleq \lambda x_1.\ldots\lambda x_n.e$
$\mathit{let}\ x = e_1\ \mathit{in}\ e_2 \triangleq (\lambda x.e_2)\ e_1$
$\lambda\_.e \triangleq \lambda x.e \quad$ for any $x \notin \mathit{fv}(e)$
$e_1; e_2 \triangleq \mathit{let}\ \_ = e_1\ \mathit{in}\ e_2$
$\mathit{ok} \triangleq \lambda x.x$
$\mathit{check}\ p\ \mathit{for}\ e \triangleq \mathit{test}\ \{p\}\ \mathit{then}\ e\ \mathit{else}\ \mathit{fail}$

We adopt the standard syntactic conventions that applications associate to the left, and the scope of a bound variable extends as far to the right as possible.

Syntactic conventions

$e_1\ e_2\ e_3$ is read $(e_1\ e_2)\ e_3$
$\lambda x.e_1\ e_2$ is read $\lambda x.(e_1\ e_2)$
$\mathit{let}\ x = e_1\ \mathit{in}\ e_2\ e_3$ is read $\mathit{let}\ x = e_1\ \mathit{in}\ (e_2\ e_3)$

2.2 Operational Semantics

We formalize the behaviour of expressions as a small-step reduction relation, indexed by the security context: the relation $e \rightarrow^S_D e'$ means that, in a context with static permissions S and dynamic permissions D, the expression e may evolve to e'. We allow $e \rightarrow^S_D e'$ only when $D \subseteq S$.

Reduction relation

$e \rightarrow^S_D e'$	security-indexed reduction ($D \subseteq S$)

Security-indexed reduction rules

(Ctx Rator)
$$\frac{e_1 \rightarrow^S_D e_1'}{e_1\, e_2 \rightarrow^S_D e_1'\, e_2}$$

(Ctx Rand)
$$\frac{e_2 \rightarrow^S_D e_2'}{v_1\, e_2 \rightarrow^S_D v_1\, e_2'}$$

(Red Appl)
$$(\lambda x.e)\, v \rightarrow^S_D e\{x \leftarrow v\}$$

(Fail Rator)
$$\mathit{fail}\ e \rightarrow^S_D \mathit{fail}$$

(Fail Rand)
$$v\ \mathit{fail} \rightarrow^S_D \mathit{fail}$$

(Ctx Frame)
$$\frac{e \rightarrow^R_{D\cap R} e'}{R[e] \rightarrow^S_D R[e']}$$

(Ctx Grant)
$$\frac{e \rightarrow^S_{D\cup(R\cap S)} e'}{\mathit{grant}\ R\ \mathit{in}\ e \rightarrow^S_D \mathit{grant}\ R\ \mathit{in}\ e'}$$

(Red Frame)
$$R[o] \rightarrow^S_D o$$

(Red Grant)
$$\mathit{grant}\ R\ \mathit{in}\ o \rightarrow^S_D o$$

(Red Test)
$$\mathit{test}\ R\ \mathit{then}\ e_{\text{true}}\ \mathit{else}\ e_{\text{false}} \rightarrow^S_D e_{R\subseteq D}$$

Rules (Ctx Rator), (Ctx Rand), and (Red Appl) implement call-by-value function evaluation; as usual, we do not reduce within function bodies. Rules (Fail Rator) and (Fail Rand) propagate exceptions through applications. The context rules (Ctx Frame) and (Ctx Grant) specify how a frame and a grant, respectively, manipulate permission sets, as described above. Rules (Red Frame) and (Red Grant) discard a frame and a grant, respectively, once its body has reduced to an outcome—this reflects the deletion of the actual stack frame for that body. Finally, (Red Test) specifies how a test inspects the dynamic permission set.

As usual, contexts $\mathcal{C}$ are expressions with a placeholder $(\cdot)$ and evaluation contexts $\mathcal{E}$ are the contexts derived from the (Ctx-) rules:

$$\mathcal{E}(\cdot) ::= (\cdot) \mid \mathcal{E}(\cdot)\, e \mid v\, \mathcal{E}(\cdot) \mid R[\mathcal{E}(\cdot)] \mid \mathit{grant}\ R\ \mathit{in}\ \mathcal{E}(\cdot)$$

The top-level reduction relation, $e \rightarrow e'$, describes the single-step evolution of a fully trusted expression e (which may of course contain partially trusted subexpressions). It is defined from the security-indexed relation by setting the static and dynamic permissions to be the full set, $\mathcal{P}$. The top-level evaluation relation, $e \Downarrow o$, computes the outcome o of evaluating an expression e.

Our semantic rules (in particular, (Ctx Frame) and (Ctx Grant)) specify how to update the dynamic permission set upon change of security context.

This strategy is known as the security-passing style [31] or the eager semantics [3, 10]. The alternative strategy—the lazy semantics used by most implementations—is to compute the dynamic permissions indirectly by inspecting the stack. We show in Appendix A that our eager semantics corresponds exactly to a lazy semantics given by Pottier, Skalka, and Smith [28]. The eager semantics is more convenient for the theory of this paper. Still, the lazy semantics appears to lead to more efficient implementations [10, 31].

Top-level reduction and evaluation

$e \to e' \triangleq e \to_{\mathcal{P}}^{\mathcal{P}} e'$	top-level reduction
$e \Downarrow o \triangleq e \to^* o$	top-level evaluation

2.3 Framing

The syntax of λ_{sec} enables framed subexpressions anywhere in an expression. In practice, framed subexpressions would appear only as the result of applying a security policy, for example, when code is first loaded. (Without a similar restriction, untrusted code could grant itself any right.)

We can describe the application of a uniform security policy as a function from the frameless fragment of λ_{sec} to the full calculus, that inserts the same, given frame under every abstraction: $R[\![\lambda x.e]\!] = \lambda x.R[R[\![e]\!]]$ and $R[\![\cdot]\!]$ commutes with all other constructs.

Framing an expression with principal R

$$R[\![x]\!] \triangleq x$$
$$R[\![\lambda x.e]\!] \triangleq \lambda x.R[R[\![e]\!]]$$
$$R[\![e_1\ e_2]\!] \triangleq R[\![e_1]\!]\ R[\![e_2]\!]$$
$$R[\![\mathit{grant}\ S\ \mathit{in}\ e]\!] \triangleq \mathit{grant}\ S\ \mathit{in}\ R[\![e]\!]$$
$$R[\![\mathit{test}\ S\ \mathit{then}\ e_1\ \mathit{else}\ e_2]\!] \triangleq \mathit{test}\ S\ \mathit{then}\ R[\![e_1]\!]\ \mathit{else}\ R[\![e_2]\!]$$
$$R[\![\mathit{fail}]\!] \triangleq \mathit{fail}$$

By construction, every abstraction in the image of the translation is of the form $\lambda x.R[e]$ for some R. Moreover, this property is preserved by reduction and substitution of values in the image of the translation for free variables. Similarly, if a frame does not contain grants in evaluation context, this property is preserved by reduction and substitution of values in the image of the translation for free variables. These structural properties are often useful as we try to perform program transformations.

Initially, we model a runtime configuration by an expression of the form

$$e\ R_1[\![v_1]\!]\ \dots\ R_n[\![v_n]\!]$$

where e accounts for the runtime, linker, and low-level resources, while $v_1, \dots, v_n$ are miscellaneous additional components, with respective static permissions $R_1, \dots R_n$ attributed by the secure loader.

3 Programming Examples

Our series of examples models interaction between I/O library functions and applets. The intent is to prevent applets from accessing the content of arbitrary files. We consider permissions $\mathcal{P} = \{\mathit{screenIO}, \mathit{fileIO}\}$ and the principals:

- $\mathit{Applet} \triangleq \{\mathit{screenIO}\}$, a mostly untrusted principal
- $\mathit{System} \triangleq \{\mathit{screenIO}, \mathit{fileIO}\}$, a fully trusted principal

Direct Access. First, consider an I/O library function that protects read access to the file system by requiring the *fileIO* permission. We assume some encoding for strings, and let *primRF* be a primitive for returning the contents of a file as a string.

$$\mathit{readFile} \triangleq \lambda n.\mathit{System}[\mathit{check\ fileIO\ for\ primRF\ n}]$$

For instance, we have

$$\mathit{Applet}[\mathit{readFile}\ \text{"secrets"}] \Downarrow \mathit{fail} \tag{1}$$
$$\mathit{System}[\mathit{readFile}\ \text{"version"}] \Downarrow \text{"Build 2601"} \tag{2}$$

In this setting, the applet code (here, *readFile* "secrets") may refer to *readFile* but not to *primRF*, and must be framed with principal *Applet*. Such expressions can be obtained by framing and linking; for instance, the expression in (1) is obtained from the initial configuration

$$\begin{array}{l}(\lambda sa.\, a\ s)\\ \mathit{System}[\![\lambda n.\mathit{check\ fileIO\ for\ primRF\ n}]\!]\\ \mathit{Applet}[\![\lambda \mathit{readFile}.\mathit{readFile}\ \text{"secrets"}]\!]\end{array}$$

One may check that no (frameless, closed) applet code substituted for $\lambda\mathit{readFile}.\mathit{readFile}$ "secrets" can cause any file to be read. We state a more general result in Section 6.

Indirect Access. Consider now a *System*-routine that calls another *System*-routine. We assume that *primDS* is the primitive that displays a string and returns *ok*.

$$\begin{aligned} \mathit{displayString} &\triangleq \lambda s.\mathit{System}[\mathit{check}\ \mathit{screenIO}\ \mathit{for}\ \mathit{primDS}\, s] \\ \mathit{displayFile} &\triangleq \lambda n.\mathit{System}[\mathit{displayString}\,(\mathit{readFile}\, n)] \end{aligned}$$

For example:

$$\mathit{Applet}[\mathit{displayString}\ \text{“hi”}] \Downarrow \mathit{ok} \tag{3}$$

$$\mathit{Applet}[\mathit{displayFile}\ \text{“secrets”}] \Downarrow \mathit{fail} \tag{4}$$

$$\mathit{System}[\mathit{displayFile}\ \text{“version”}] \Downarrow \mathit{ok} \tag{5}$$

If stack inspection did not compound principals, the call in example (4) would succeed.

Overriding Policy. Sometimes it is acceptable for trusted code to make exceptions to a standard policy. For instance, we may wish to allow any code read access to a file containing the operating system version.

$$\begin{aligned} &\mathit{readVersion} \\ &\quad \triangleq \lambda\_.\mathit{System}[\mathit{grant}\ \{\mathit{fileIO}\}\ \mathit{in}\ \mathit{readFile}\ \text{“version”}] \end{aligned}$$

For example:

$$\mathit{Applet}[\mathit{readVersion}\ \mathit{ok}] \Downarrow \text{“Build 2601”} \tag{6}$$

The above are examples of calls from less trusted to more trusted code. A symmetric situation is where more trusted code calls less trusted, such as when trusted libraries call methods such as *ToString* or *Equals* on untrusted objects. Attempts by such methods to exploit the greater privileges of their callers are also thwarted by stack inspection.

Untrusted Results. The following example describes some trusted code depending on data supplied by untrusted code. We have a *System*-function *foolishDisplayFile* that calls a function parameter h to compute a filename s, and then calls *displayFile* s to display it.

$$\mathit{foolishDisplayFile} \triangleq \lambda h.\mathit{System}[\mathit{displayFile}\,(h\ \mathit{ok})]$$

Now, since the call to h completes before the call to *displayFile* begins, the principal associated with h has disappeared from the stack before the access tests in *displayFile* occur. So the following call, which allows an untrusted function to determine which file is displayed, succeeds.

$$\mathit{foolishDisplayFile}\,(\lambda\_.\mathit{Applet}[\text{“secrets”}]) \;\Downarrow\; \mathit{ok} \tag{7}$$

This example illustrates that stack inspection does not track data dependencies. Stack inspection does prevent the function parameter from making privileged calls while it is running, but it does not prevent it influencing computation, perhaps against policy, once it has terminated and returned a result.

Higher Order. Our last example is more involved. Trusted code (*main*) calls an applet; the applet calls trusted code (*fileHandler*) to build a *System*-closure for its choice of parameters (“secrets” and *leak*) and returns that closure; later, a trusted call triggers the closure:

$$\begin{array}{rcl} \mathit{main} & \triangleq & \mathit{System}[\![\lambda h.(h\ \mathit{ok}\ \mathit{ok})]\!] \\ \mathit{fileHandler} & \triangleq & \mathit{System}[\![\lambda\, s\, c\, \_.\, c\,(\mathit{readFile}\, s)]\!] \\ \mathit{leak} & \triangleq & \mathit{Applet}[\![\lambda s.\mathit{displayString}\, s]\!] \end{array}$$

$$\mathit{main}\,\big(\lambda\_.\mathit{Applet}[\mathit{fileHandler}\ \text{“secrets”}\ \mathit{leak}]\big) \;\Downarrow\; \mathit{ok} \tag{8}$$

Since the security context used to create the closure is discarded as $\mathit{Applet}[\mathit{fileHandler}\ \text{“secrets”}\ \mathit{leak}]$ returns, the closure gets access to “secrets”. In more detail, we have the following, where ok_S is short for $\mathit{System}[\![\mathit{ok}]\!]$.

$$\begin{array}{rl} & \mathit{main}\,\big(\lambda\_.\mathit{Applet}[\mathit{fileHandler}\ \text{“secrets”}\ \mathit{leak}]\big) \\ \to^2 & \mathit{System}[\mathit{Applet}[\mathit{fileHandler}\ \text{“secrets”}\ \mathit{leak}]\ \mathit{ok}_S] \\ \to^2 & \mathit{System}[\mathit{Applet}[\mathit{System}[\mathit{System}[\\ & \qquad \lambda\_.\mathit{System}[\mathit{leak}\,(\mathit{readFile}\ \text{“secrets”})]]]]\ \mathit{ok}_S] \\ \to^3 & \mathit{System}[\lambda\_.\mathit{System}[\mathit{leak}\,(\mathit{readFile}\ \text{“secrets”})]\ \mathit{ok}_S] \\ \to^5 & \mathit{System}[\mathit{System}[\mathit{leak}\,\langle\text{content of “secrets”}\rangle]] \\ \to^6 & \mathit{System}[\mathit{System}[\mathit{Applet}[\mathit{ok}]]] \;\to^3\; \mathit{ok} \end{array}$$

The first four steps substitute values for h, $\_$, s, and c; the next three steps discard the frames after evaluating the closure; then, we have two reductions with all *System* permissions followed by the reductions in the applet code.

In this situation, it is quite hard to modify the code so that a suitably framed closure is returned. A safe approach may be to request the permissions that will be used within the closure before returning the closure.

However, this requires specific knowledge of those permissions. Instead of *fileHandler*, one may write, for instance:

$$
\begin{aligned}
\mathit{safeFileHandler} \ \triangleq\ & \lambda s.\mathit{test}\ \{\mathit{fileIO}\} \\
& \quad \mathit{then}\ \mathit{System}[\![\lambda c\,\_.\,c\,(\mathit{readFile}\,s)]\!] \\
& \quad \mathit{else}\ \mathit{System}[\![\lambda c\,\_.\mathit{fail}]\!]
\end{aligned}
$$

Another, more uniform approach is to provide a general mechanism to capture the current dynamic permissions (D) and restore them as the closure is triggered. In the JVM and in the CLR, such a mechanism is used internally for special cases of closures, for instance to start a new thread. As the corresponding closure is created, the stack is scanned to compute D, then D is used to build the first frame of the new stack. This design issue is discussed in [10, section 3.11].

The example above may seem a little contrived, but in fact is very common in an object-oriented setting: whenever a call returns an object from untrusted code, further calls to its methods will be performed using virtual calls, and there is no simple, uniform way to test whether that object encapsulates low-trust parameters (or even code).

4 Equational Reasoning

In order to transform programs while preserving their semantics, we rely on Morris-style contextual equivalence [24]. Since it is preserved by all contexts, local transformations based on contextual equivalence may be used anywhere in a program.

Contextual equivalence

Let $e\Downarrow$ if and only if there is an outcome o with $e \Downarrow o$.
Let $e \simeq e'$ if and only if, for all contexts $\mathcal{C}$,
if both $\mathcal{C}(e)$ and $\mathcal{C}(e')$ are closed, then $\mathcal{C}(e)\Downarrow \Longleftrightarrow \mathcal{C}(e')\Downarrow$.

Contextual equivalence is strictly more discriminating than in the call-by-value λ-calculus (CBV), even for pure λ-terms. For instance, the terms

$$
\begin{aligned}
& \lambda x.\mathit{let}\ z = x\ \mathit{ok}\ \mathit{in}\ \lambda\_.z \\
\text{and}\quad & \lambda x.\mathit{let}\ z = x\ \mathit{ok}\ \mathit{in}\ \lambda\_.(x\ \mathit{ok})
\end{aligned}
$$

are equivalent in CBV but can be separated in λ_{sec} using the context

$$\varnothing[(\cdot)\,(\lambda\_.\mathit{test}\ \mathcal{P}\ \mathit{then}\ \Omega\ \mathit{else}\ \mathit{ok})]\ \mathit{ok}$$

where Ω is an expression that diverges. This suggests that usual optimizations may break, and motivates our study of contextual equivalence.

To see why these two expressions are not contextually equivalent, we calculate as follows. Let $v = \lambda\_.\mathit{test}\ \mathcal{P}\ \mathit{then}\ \Omega\ \mathit{else}\ \mathit{ok}$. Placing the first expression in context, we obtain

$$\begin{array}{rl} & \varnothing[(\lambda x.\mathit{let}\ z = x\ \mathit{ok}\ \mathit{in}\ \lambda\_.z)\ v]\ \mathit{ok} \\ \rightarrow & \varnothing[\mathit{let}\ z = v\ \mathit{ok}\ \mathit{in}\ \lambda\_.z]\ \mathit{ok}\ \rightarrow^3\ \varnothing[\lambda\_.\mathit{ok}]\ \mathit{ok}\ \rightarrow^2\ \mathit{ok} \end{array}$$

In contrast, placing the second expression in context, we obtain

$$\begin{array}{rl} & \varnothing[(\lambda x.\mathit{let}\ z = x\ \mathit{ok}\ \mathit{in}\ \lambda\_.(x\ \mathit{ok}))\ v]\ \mathit{ok} \\ \rightarrow & \varnothing[\mathit{let}\ z = v\ \mathit{ok}\ \mathit{in}\ \lambda\_.(v\ \mathit{ok})]\ \mathit{ok}\ \rightarrow^3\ \varnothing[\lambda\_.(v\ \mathit{ok})]\ \mathit{ok}\ \rightarrow^4\ \Omega \end{array}$$

4.1 Equational Properties of λ_{sec}

We present a new equational theory for λ_{sec} that is sound for contextual equivalence and complete with respect to the reduction semantics. We first state the theory and briefly comment on its equations.

Let $e \equiv e'$ be the smallest relation on expressions to satisfy the congruence equations and primitive equations listed below.

Judgment

$e \equiv e'$	equational theory of λ_{sec}

Congruence equations

(Eq Symm) $e' \equiv e \Longrightarrow e \equiv e'$
(Eq Trans) $e \equiv e', e' \equiv e'' \Longrightarrow e \equiv e''$
(Eq x) $x \equiv x$
(Eq Fun) $e \equiv e' \Longrightarrow \lambda x.e \equiv \lambda x.e'$
(Eq Appl) $e_1 \equiv e_1', e_2 \equiv e_2' \Longrightarrow e_1\ e_2 \equiv e_1'\ e_2'$
(Eq Frame) $e \equiv e' \Longrightarrow R[e] \equiv R[e']$
(Eq Grant) $e \equiv e' \Longrightarrow \mathit{grant}\ R\ \mathit{in}\ e \equiv \mathit{grant}\ R\ \mathit{in}\ e'$
(Eq Test) $e_1 \equiv e_1', e_2 \equiv e_2' \Longrightarrow \mathit{test}\ R\ \mathit{then}\ e_1\ \mathit{else}\ e_2 \equiv \mathit{test}\ R\ \mathit{then}\ e_1'\ \mathit{else}\ e_2'$
(Eq Fail) $\mathit{fail} \equiv \mathit{fail}$

Primitive equations

(Fun Beta) $(\lambda x.e)\ v \equiv e\{x{\leftarrow}v\}$
(Fun Eta) $x \notin \mathit{fv}(v) \Longrightarrow \lambda x.v\ x \equiv v$

(Let Eta) $let\ x = e\ in\ x \equiv e$
(Let Let) $x_1 \notin fv(e_3) \Longrightarrow$
$let\ x_1 = e_1\ in\ (let\ x_2 = e_2\ in\ e_3) \equiv$
$let\ x_2 = (let\ x_1 = e_1\ in\ e_2)\ in\ e_3$

(Frame o) $R[o] \equiv o$
(Frame Frame Appl)
$R_1[R_2[e_1\ e_2]] \equiv R_1[R_2[(R_1[R_2[e_1]])\ (R_1[R_2[e_2]])]]$
(Frame Let) $R[let\ x = e_1\ in\ e_2] \equiv let\ x = R[e_1]\ in\ R[e_2]$
(Frame Frame) $R_1 \supseteq R_2 \Longrightarrow R_1[R_2[e]] \equiv R_2[e]$
(Frame Frame Frame)
$R_1[R_2[R_3[e]]] \equiv (R_1 \cap R_2)[R_3[e]]$
(Frame Frame Grant)
$R_1[R_2[grant\ R_3\ in\ e]] \equiv (R_1 \cup R_3)[R_2[grant\ R_3\ in\ e]]$
(Frame Grant)
$R_1[grant\ R_2\ in\ e] \equiv R_1[grant\ R_1 \cap R_2\ in\ e]$
(Frame Grant Frame) $R_1 \supseteq R_2 \Longrightarrow$
$R_1[grant\ R_2\ in\ R_3[e]] \equiv R_1[R_3[grant\ R_2\ in\ e]]$
(Frame Grant Test) $R_1 \supseteq R_2 \supseteq R_3 \Longrightarrow$
$R_1[grant\ R_2\ in\ test\ R_3\ then\ e_1\ else\ e_2] \equiv$
$R_1[grant\ R_2\ in\ e_1]$
(Frame Test Then) $R_1 \supseteq R_2 \Longrightarrow$
$R_1[test\ R_2\ then\ e_1\ else\ e_2] \equiv$
$test\ R_2\ then\ R_1[e_1]\ else\ R_1[e_2]$
(Frame Test Else) $\neg(R_1 \supseteq R_2) \Longrightarrow$
$R_1[test\ R_2\ then\ e_1\ else\ e_2] \equiv R_1[e_2]$

(Grant $\varnothing$) $grant\ \varnothing\ in\ e \equiv e$
(Grant o) $grant\ R\ in\ o \equiv o$
(Grant Appl) $grant\ R\ in\ (e_1\ e_2) \equiv$
$grant\ R\ in\ ((grant\ R\ in\ e_1)\ grant\ R\ in\ e_2)$
(Grant Let) $grant\ R\ in\ (let\ x = e_1\ in\ e_2) \equiv$
$let\ x = (grant\ R\ in\ e_1)\ in\ (grant\ R\ in\ e_2)$
(Grant Grant)
$grant\ R_1\ in\ grant\ R_2\ in\ e \equiv grant\ R_1 \cup R_2\ in\ e$
(Grant Frame) $grant\ R_1\ in\ R_2[e] \equiv grant\ R_1 \cap R_2\ in\ R_2[e]$
(Grant Frame Grant)
$grant\ R_2\ in\ R_1[grant\ R_2\ in\ e] \equiv R_1[grant\ R_2\ in\ e]$

(Test $\varnothing$) $test\ \varnothing\ then\ e_1\ else\ e_2 \equiv e_1$
(Test Refl) $test\ R\ then\ e\ else\ e \equiv e$

(Test ∪) $\textit{test } R_1{\cup}R_2 \textit{ then } e_1 \textit{ else } e_2 \equiv$
$\quad \textit{test } R_1 \textit{ then } (\textit{test } R_2 \textit{ then } e_1 \textit{ else } e_2) \textit{ else } e_2$
(Test Grant) $\textit{test } R \textit{ then } e_1 \textit{ else } e_2 \equiv$
$\quad \textit{test } R \textit{ then } (\textit{grant } R \textit{ in } e_1) \textit{ else } e_2$

(Eq Fail Rator) $\textit{fail}\ e \equiv \textit{fail}$
(Eq Fail Rand) $v\ \textit{fail} \equiv \textit{fail}$

Derived equations

(Eq Refl) $e \equiv e$
(Let Beta) $\textit{let } x = v \textit{ in } e \equiv e\{x{\leftarrow}v\}$

(Frame Dup) $R[R[e]] \equiv R[e]$
(Frame Appl) $R[e_1\ e_2] \equiv R[R[e_1]\ R[e_2]]$
(Frame Frame ∩) $R_1[R_2[e]] \equiv (R_1{\cap}R_2)[R_2[e]]$
(Frame Grant ∩) $R_1[\textit{grant } R_2 \textit{ in } e] \equiv R_1[\textit{grant } R_1{\cap}R_2 \textit{ in } R_1[e]]$
(Frame Grant ∩ ∅) $R_1 \cap R_2 = \varnothing \implies R_1[\textit{grant } R_2 \textit{ in } e] \equiv R_1[e]$
(Frame Grant Frame ∩) $R_1[\textit{grant } R_2 \textit{ in } R_3[e]] \equiv R_1[R_3[\textit{grant } R_1{\cap}R_2 \textit{ in } e]]$
(Frame Frame Test Else) $\neg(R_1 \supseteq R_3) \implies$
$\quad R_1[R_2[\textit{test } R_3 \textit{ then } e_1 \textit{ else } e_2]] \equiv R_1[R_2[e_2]]$

Proposition 1 *The equations in the preceding table are derivable within the equational theory.*

The λ_{sec}-calculus extends Plotkin's call-by-value λ_v; accordingly, we retain β_v and η_v equations, here named (Fun Beta) and (Fun Eta). As in Plotkin's calculus, the following more general laws are unsound: $(\lambda x.e)\, e' \equiv e\{x{\leftarrow}e'\}$ and $x \notin \textit{fv}(e) \implies \lambda x.e\, x \equiv e$. We also have the standard monad laws for *let* from Moggi's computational λ-calculus [23], here named (Let Beta), (Let Eta), and (Let Let).

Specific rules manipulate nested security constructors. In $R_1[R_2[e]]$, the effect of a *grant* in e is determined by R_2 but not by R_1. Therefore, the equation $R_1[R_2[e]] \equiv (R_1{\cap}R_2)[e]$ is not sound in general. Still, (Frame Frame) coalesces two frames into one when the outer principal dominates the inner, and (Frame Frame Frame) unconditionally coalesces three frames into two. Rules (Frame Let) and (Grant Let) are limited forms of the more general equations $R[e_1\, e_2] \equiv R[e_1]\, R[e_2]$ and $\textit{grant } R \textit{ in } (e_1\, e_2) \equiv (\textit{grant } R \textit{ in } e_1)(\textit{grant } R \textit{ in } e_2)$, which are not sound. Rule (Frame Frame Appl) pushes doubly nested frames into applications.

When the enclosing permission modifiers are available, the outcome of a grant may be determined, independently of the enclosing context. We obtain partial commutativity laws (Frame Grant), (Grant Frame), (Frame Grant Frame), (Grant Frame Grant). Similarly, the outcome of a test may be determined. Regarding (Frame Test Else), if the principal R_1 cannot access the resource R_2, testing for that resource must fail. On the other hand, $R_1 \supseteq R_2$ does not imply $R_1[\mathit{test}\ R_2\ \mathit{then}\ e_1\ \mathit{else}\ e_2] \equiv R_1[e_1]$, because the calling context may not have been granted R_2. A corollary of (Frame Grant) and (Grant $\varnothing$) is the rule $R_1 \cap R_2 = \varnothing \implies R_1[\mathit{grant}\ R_2\ \mathit{in}\ e] \equiv R_1[e]$. If the principal R_1 cannot access the resources R_2, it is futile for code framed by R_1 to try to grant R_2.

Using bisimulation proof techniques discussed in the next section, we can show the equational theory to be sound with respect to contextual equivalence.

Theorem 1 *If $e \equiv e'$ then $e \simeq e'$.*

We cannot expect the converse, completeness with respect to contextual equivalence. The set of provable equations $e \equiv e'$ is recursively enumerable whereas the set of contextual equivalences $e \simeq e'$ is not.

Still, we do obtain a limited completeness result with respect to the security-indexed reduction semantics. To state the theorem, we introduce *security-setters*, $\mathcal{C}^S_D(\cdot)$, evaluation contexts that set the static and dynamic permissions within the context to S and D, respectively. More precisely, when running $\mathcal{C}^S_D(e)$ with arbitrary permission sets S' and D', the expression e runs with permission sets S and D.

Security-setters

$\mathcal{C}^S_D(\cdot) \triangleq D[\mathit{grant}\ D\ \mathit{in}\ S[\cdot]]$	where $D \subseteq S$

Theorem 2 *If $e \rightarrow^S_D e'$ then $\mathcal{C}^S_D(e) \equiv \mathcal{C}^S_D(e')$.*

The proof shows there are sufficient equations to distribute information about the security context to where it is needed to justify reduction steps; indeed, the proof prompted the discovery of various equations. If we view the equational theory as a new axiomatic semantics of λ_{sec}, the theorem shows that the reduction relation is a correct algorithm for computing certain equations.

Corollary 1 *If $e \Downarrow o$ then $\mathcal{P}[\mathit{grant}\ \mathcal{P}\ \mathit{in}\ e] \equiv o$.*

Proof Assume $e \Downarrow o$, that is, $e \rightarrow^{\mathcal{P}}_{\mathcal{P}}{}^* o$. We apply Theorem 2 to each step and, by transitivity of $\equiv$, we obtain $\mathcal{C}^{\mathcal{P}}_{\mathcal{P}}(e) \equiv \mathcal{C}^{\mathcal{P}}_{\mathcal{P}}(o)$. Finally, $\mathcal{C}^{\mathcal{P}}_{\mathcal{P}}(o) \equiv o$ by rules (Grant o) and (Frame o). $\square$

Simplified Theory in λ_{sec} without Grants In the subcalculus without *grant* expressions, the equations above remain sound (when applicable), and they can be simplified further. The single rule $R_1[R_2[e]] \equiv (R_1 \cap R_2)[e]$, which becomes sound in the absence of grants, subsumes (Frame Frame Appl), (Frame Frame), and (Frame Frame Frame). Besides, we need (Frame Frame Test Else) no more—we use (Frame Test Else) instead.

In combination, we can normalize expressions by pushing frames under all other constructors (except applications) until all frames appear in subterms of the form $R[x\ e]$.

4.2 Basic Applications

In addition to justifying contextual equivalences mentioned in Section 5, we can apply the theory as follows.

Framing versus Currying. As illustrated in example (8), the framing translation of Section 2.3 yields multiple nested frames when applied to functions with multiple arguments. Using (Frame o), we can discard these duplicate frames:

$$R[\![\lambda xy.e]\!] \triangleq \lambda x.R[\lambda y.R[R[\![e]\!]]] \equiv \lambda xy.R[R[\![e]\!]]$$

Hence, we can choose the latter form as a more efficient translation when dealing with multiple arguments (or more generally with functions that have multiple entry points).

Shortening Stack Inspections. In typical implementations of stack inspection, permissions are tested on demand, with a runtime cost that grows linearly with the depth of the stack. When the same permissions are frequently tested, it may be worth testing those permissions in advance, then granting them, so that all further tests succeed faster. Indeed, this is a recommended idiom for optimizing programs that perform frequent checks in the CLR [20].

In the theory, we can use (Test Grant) to justify this kind of program transformations by deriving the equation:

$$e \equiv \mathit{test}\ R\ \mathit{then}\ (\mathit{grant}\ R\ \mathit{in}\ e)\ \mathit{else}\ e$$

Normal Forms for Security-Modifiers. We say that an evaluation context is a security-modifier when it is built using one or more frames and any number of grants. Using the equational theory, we can systematically simplify such contexts. (We lift the relation $\equiv$ pointwise from expressions to contexts seen as functions: $\mathcal{C} \equiv \mathcal{C}'$ when for all e we have $\mathcal{C}(e) \equiv \mathcal{C}'(e)$.) More generally, one can first apply distributive laws to push every modifier below other constructors, then simplify the resulting security modifiers, as summarized below.

Proposition 2 *For every security-modifier $\mathcal{C}$, there exist unique permission sets $D \subseteq A \subseteq R \subseteq S$ such that*

$$\mathcal{C} \equiv \mathit{grant}\ A\ \mathit{in}\ R[S[\mathit{grant}\ D\ \mathit{in}\ (\cdot)]]$$

Proof Let $\mathcal{C}$ be a security modifier. We obtain an equivalent security modifier of the form given in Proposition 2 by applying the following series of rewritings:

(1) Introduce an empty grant between nested frames, on the outside, and on the inside (Grant $\varnothing$), then merge all nested grants (Grant Grant). For some $n \geq 1$, this yields a context of the form

$$\mathit{grant}\ A'\ \mathit{in}\ S_1[\mathit{grant}\ D_1\ \mathit{in}\ \ldots S_n[\mathit{grant}\ D_n\ \mathit{in}\ (\cdot)]\ldots]$$

(2) For $i = 1, \ldots, (n-1)$, apply (Frame Grant Frame $\cap$) to S_i, D_i, and S_{i+1}. This yields a context of the form $\mathit{grant}\ A'\ \mathit{in}\ S_1[\ldots S_n[\mathit{grant}\ D'\ \mathit{in}\ (\cdot)]]$.

(3) Apply (Frame Frame Frame) $n-2$ times then (Frame Frame)—or (Frame Dup) once if $n = 1$—to substitute $R'[S[(\cdot)]]$ for $S_1[\ldots S_n[(\cdot)]]$, for some permissions R' and S with $R' \subseteq S$.

(4) Let $D \triangleq D' \cap S$. We calculate

$$\begin{array}{rll}
\mathcal{C} & \equiv & \mathit{grant}\ A'\ \mathit{in}\ R'[S[\mathit{grant}\ D'\ \mathit{in}\ (\cdot)]] \quad \text{as described above} \\
& \equiv & \mathit{grant}\ A'\ \mathit{in}\ R'[S[\mathit{grant}\ D\ \mathit{in}\ (\cdot)]] \quad \text{by (Frame Grant)} \\
& \equiv & \mathit{grant}\ A'\ \mathit{in}\ (R' \cup D)[S[\mathit{grant}\ D\ \mathit{in}\ (\cdot)]] \\
& & \text{by (Frame Frame Grant)} \\
& \equiv & \mathit{grant}\ A'\ \mathit{in}\ (R' \cup D)[\mathit{grant}\ D\ \mathit{in}\ S[(\cdot)]] \\
& & \text{by (Frame Grant Frame) since } R' \cup D \supseteq D \\
& \equiv & \mathit{grant}\ A'\ \mathit{in}\ \mathit{grant}\ D\ \mathit{in}\ (R' \cup D)[\mathit{grant}\ D\ \mathit{in}\ S[(\cdot)]] \\
& & \text{by (Grant Frame Grant)} \\
& \equiv & \mathit{grant}\ A' \cup D\ \mathit{in}\ (R' \cup D)[\mathit{grant}\ D\ \mathit{in}\ S[(\cdot)]] \\
& & \text{by (Grant Grant)} \\
& \equiv & \mathit{grant}\ A' \cup D\ \mathit{in}\ (R' \cup D)[S[\mathit{grant}\ D\ \mathit{in}\ (\cdot)]] \\
& & \text{by (Frame Grant Frame) since } R' \cup D \supseteq D \\
& \equiv & \mathit{grant}\ (A' \cup D) \cap (R' \cup D)\ \mathit{in}\ (R' \cup D)[S[\mathit{grant}\ D\ \mathit{in}\ (\cdot)]] \\
& & \text{by (Grant Frame)}
\end{array}$$

(5) Let $R \triangleq R' \cup D$ and $A \triangleq (A' \cup D) \cap R$. We already have $R \subseteq S$. We immediately obtain $A \subseteq R$. We also have $A = (A' \cup D) \cap (R' \cup D) = (A' \cap R') \cup D$ hence $D \subseteq A$.

Uniqueness follows from the soundness of the equational theory and a characterization of A, R, S, and D.

Let $t \triangleq \mathit{test}\ X\ \mathit{then}\ \mathit{ok}\ \mathit{else}\ \Omega$. For a given security-modifier $\mathcal{E}$, we let $P(\mathcal{E})$ be the largest set of permissions X such that $\mathcal{E}(t)\Downarrow$. Such a set exists, and is preserved by contextual equivalence for security-modifiers: $\mathcal{E} \simeq \mathcal{E}'$ implies, for all X, $\mathcal{E}(t)\Downarrow \iff \mathcal{E}'(t)\Downarrow$; choosing $X = P(\mathcal{E})$ and $X = P(\mathcal{E}')$, we obtain $P(\mathcal{E}) \supseteq P(\mathcal{E}')$ and $P(\mathcal{E}) \subseteq P(\mathcal{E}')$, and finally $P(\mathcal{E}) = P(\mathcal{E}')$.

For a given $\mathcal{C} = \mathit{grant}\ A\ \mathit{in}\ R[S[\mathit{grant}\ D\ \mathit{in}\ (\cdot)]]$ with $D \subseteq A \subseteq R \subseteq S$, it suffices to establish that these permission sets can be computed using $P(\cdot)$:

$$\begin{aligned} D &= P(\varnothing[\mathcal{C}]) \\ A &= P(\varnothing[\mathcal{P}[\mathcal{C}]]) \\ R &= P(\mathcal{C}) \\ S &= P(\mathcal{C}(\mathit{grant}\ \mathcal{P}\ \mathit{in}\ (\cdot))) \end{aligned}$$

Then, for any $\mathcal{C}' = \mathit{grant}\ A'\ \mathit{in}\ R'[S'[\mathit{grant}\ D'\ \mathit{in}\ (\cdot)]]$ with $D' \subseteq A' \subseteq R' \subseteq S'$ and $\mathcal{C} \simeq \mathcal{C}'$, we have $\varnothing[\mathcal{C}] \simeq \varnothing[\mathcal{C}']$, $P(\varnothing[\mathcal{C}]) = P(\varnothing[\mathcal{C}'])$, and thus $D = D'$. Similarly, $A = A'$, $R = R'$, and $S = S'$.

We now establish the characterizations of D, A, R, S given above. For any given security-modifier $\mathcal{E}$, we have either $\mathcal{E}(t) \to \mathcal{E}(\mathit{ok}) \Downarrow \mathit{ok}$ or $\mathcal{E}(t) \to \mathcal{E}(\Omega) \to^* \mathcal{E}(\Omega)$, according to the test in the first step. We make the dependence of $\mathcal{E}(t)\Downarrow$ upon X explicit by unfolding (Red Test) for the test and (Ctx Frame) or (Ctx Grant) for every security constructor that appears in $\mathcal{E}$. In the following, we let e and f range over expressions that are not outcomes. We will use the equivalences:

$$\mathcal{C}(e) \to^{U}_{V} \mathcal{C}(e') \iff e \to^{S}_{D \cup (U \cap A) \cup (V \cap R)} e' \tag{9}$$

$$t \to^{U'}_{V'} \mathit{ok} \iff X \subseteq V' \tag{10}$$

- Let $\mathcal{E} = \varnothing[\mathcal{C}]$. We have $\varnothing[f] \to \varnothing[f'] \iff f \to^{\varnothing}_{\varnothing} f'$. For $f = \mathcal{C}(t)$ and $f' = \mathcal{C}(\mathit{ok})$, we compose this equivalence with (9) for $U = V = \varnothing$ and (10) for $V' = D$ and obtain $\mathcal{E}(t)\Downarrow \iff X \subseteq D$, hence $D = P(\mathcal{E})$.

- Let $\mathcal{E} = \varnothing[\mathcal{P}[\mathcal{C}]]$. We have $\varnothing[\mathcal{P}[f]] \to \varnothing[\mathcal{P}[f']] \iff f \to^{\mathcal{P}}_{\varnothing} f'$. We compose this equivalence with (9) for $U = \mathcal{P}$ and $V = \varnothing$, then (10) for $V' = A$, and obtain $\mathcal{E}(t)\Downarrow \iff X \subseteq A$, hence $A = P(\mathcal{E})$.

- We compose (9) for $U = V = \mathcal{P}$ and (10) for $V' = R$ and obtain $\mathcal{C}(t)\Downarrow \iff X \subseteq R$, hence $R = P(\mathcal{C})$.

- Let $\mathcal{E} = \mathcal{C}(\mathit{grant}\ \mathcal{P}\ \mathit{in}\ (\cdot))$. We have $\mathit{grant}\ \mathcal{P}\ \mathit{in}\ f \rightarrow^S_R \mathit{grant}\ \mathcal{P}\ \mathit{in}\ f' \iff f \rightarrow^S_S f'$. We compose (9) for $U = V = \mathcal{P}$ with this equivalence, then (10) for $V' = S$, and obtain $\mathcal{E}(t)\Downarrow \iff X \subseteq S$, hence $S = P(\mathcal{E})$. $\square$

Informally, D collects the dynamic permissions always present in $(\cdot)$, A collects the permissions present when statically available in the enclosing context, R collects the permissions present when dynamically available in the enclosing context, and S collects the permissions present when self-granted.

These contexts summarize the security content of arbitrary slices of the stack; they may be used to rearrange stacks at runtime. The security-setter contexts $\mathcal{C}^S_D(\cdot)$ used in Theorem 2 are a special case. The two forms are equivalent only when $A = R = D$, that is, when $\mathcal{C}$ does not depend on its environment. We have:

$$\begin{array}{rcl} \mathcal{C}^S_D(\cdot) & \triangleq & D[\mathit{grant}\ D\ \mathit{in}\ S[\cdot]] \\ & \equiv & \mathit{grant}\ D\ \mathit{in}\ D[\mathit{grant}\ D\ \mathit{in}\ S[\cdot]] \text{ by (Grant Frame Grant)} \\ & \equiv & \mathit{grant}\ D\ \mathit{in}\ D[S[\mathit{grant}\ D\ \mathit{in}\ \cdot]] \text{ by (Frame Grant Frame)} \end{array}$$

As another illustration for our normal form, consider a test for permissions in an arbitrary security modifier $\mathcal{C}$, and let A, R, S, D be the permissions of its normal form. If $T \subseteq R$, we have

$$\begin{array}{rl} & \mathcal{C}(\mathit{test}\ T\ \mathit{then}\ e_1\ \mathit{else}\ e_2) \\ \equiv & \mathit{test}\ (R \backslash A) \cap T\ \mathit{then}\ \mathcal{C}(\mathit{test}\ (A \backslash D) \cap T\ \mathit{then}\ e_1\ \mathit{else}\ e_2)\ \mathit{else}\ \mathcal{C}(e_2) \end{array}$$

The latter expression emphasizes the only dynamic parts of the test $(R \backslash A) \cap T$ and $(A \setminus D) \cap T$; the form can be simplified further when they are empty, for instance within a security-setter.

If $T \not\subseteq R$, the test fails independently of the context, and we have $\mathcal{C}(\mathit{test}\ T\ \mathit{then}\ e_1\ \mathit{else}\ e_2) \equiv \mathcal{C}(e_2)$.

4.3 Proof Technique: Applicative Bisimilarity

As usual, the quantification over arbitrary contexts in the definition of contextual equivalence makes it cumbersome to apply the definition directly when proving equivalences. In this section, we present a secondary equivalence, a form of Abramsky's applicative bisimilarity [2], that avoids any quantification over contexts, and hence is easier to establish. We can show that bisimilarity is a congruence relation using Howe's method [15], and hence that it coincides with contextual equivalence. Therefore, we can use bisimulation arguments to establish contextual equivalences. We use this technique to prove Theorem 1.

Two closed expressions are applicatively bisimilar if, given any static and dynamic permissions, S and D, whenever one expression reduces to an outcome, so does the other, and moreover, the two outcomes match in the sense that either (1) both are failures, or (2) both are abstractions such that when they receive identical values they are themselves applicatively bisimilar.

We formally define applicative bisimilarity by the following fairly standard series of definitions. The novelty relative to previous versions of applicative bisimilarity is the quantification over static and dynamic permissions; without this quantification, we would lose congruence with respect to frames and grants. For several papers discussing applicative bisimilarity, and related techniques, see the book edited by Gordon and Pitts [11].

- Let $e \Downarrow_D^S o$ if and only if both $D \subseteq S$ and $e(\rightarrow_D^S)^* o$.
- An *applicative simulation* is a relation $\mathcal{S}$ on closed expressions such that $e_1 \mathcal{S} e_2$ implies:
  (1) if $e_1 \Downarrow_D^S \mathit{fail}$ then $e_2 \Downarrow_D^S \mathit{fail}$;
  (2) if $e_1 \Downarrow_D^S \lambda x.f_1$ then there is $\lambda x.f_2$ such that $e_2 \Downarrow_D^S \lambda x.f_2$ and for every closed value v, $f_1\{x \leftarrow v\} \mathcal{S} f_2\{x \leftarrow v\}$.
- An *applicative bisimulation* is a relation $\mathcal{S}$ such that both $\mathcal{S}$ and $\mathcal{S}^{-1}$ are applicative simulations.
- Let *ground applicative bisimilarity*, $\sim$, be the greatest applicative bisimulation, that is, the union of all applicative bisimulations.
- Let *(applicative) bisimilarity*, $\sim^\circ$, be such that $e \sim^\circ e'$ if and only if $e\sigma \sim e'\sigma$ for all substitutions σ such that $\sigma = \{x_1 \leftarrow v_1\} \cdots \{x_n \leftarrow v_n\}$ for some closed $v_1, \ldots, v_n$ where $\{x_1, \ldots, x_n\} = \mathit{fv}(e\, e')$.

We prove congruence by Howe's method. The idea is to construct an auxiliary relation, the congruence candidate, that clearly includes bisimilarity and is a congruence. By showing that the congruence candidate is a bisimulation, it follows that it is included in bisimilarity, and therefore the two are one. Hence, we obtain:

Theorem 3 *Bisimilarity is a congruence.*

Given congruence, the identity of contextual equivalence and applicative bisimilarity follows easily. The interesting step in the proof is to show that contextual equivalence is an applicative bisimulation.

Theorem 4 *Bisimilarity equals contextual equivalence.*

Some (though not all) of the equations of Section 4.1 are justified by Theorem 4 in combination with the following simple proof principle. It is justified by a bisimulation argument. Using this proposition is considerably simpler than attempting direct proofs of contextual equivalence.

Proposition 3 *For any expressions e_1 and e_2, $e_1 \sim^\circ e_2$ if for all D and S such that $D \subseteq S$, and for all substitutions σ sending variables to closed values with $dom(\sigma) = fv(e_1\ e_2)$ and for all o, we have $e_1\sigma \Downarrow_D^S o \Longleftrightarrow e_2\sigma \Downarrow_D^S o$.*

We can show that security-setting contexts $\mathcal{C}_D^S(\cdot)$ relate top-level and security-indexed evaluation in the sense that in general $\mathcal{C}_D^S(e) \Downarrow o \Longleftrightarrow e \Downarrow_D^S o$. Therefore, this proof principle can be read as a simple context lemma [21] reducing proofs of contextual equivalence to the consideration of a limited set of contexts.

An alternative strategy for proving soundness of the equational theory would be to rely on a denotational semantics of λ_{sec} in (a variant of) the plain λ-calculus, such as the security passing transformation [28], and use an equivalence in the target calculus. However, contextual equivalence after the translation is complicated and in general finer. For instance, the environment may provide a representation of the dynamic permissions that diverges when a test is performed.

5 Program Transformations

We consider two categories of program transformations. One may try to optimize the use of permissions and stack inspections to reduce their runtime costs; such optimizations are studied in the literature, and illustrated in Section 4.1. Alternatively, one may try to carry over standard optimizations to a setting with stack inspection. The examples given below suggest that this requires some care, even for simple optimizations. As can be expected, it is important (and hard) to effectively combine both kinds of optimizations. We largely ignore this issue, and instead establish the correctness of individual transformations.

Runtime behaviour is complicated by the application of a security policy. We may consider program transformations in different situations:

(1) Seen from the front-end compiler (usually in charge of performing global optimizations), optimizations operate before the framing translation, so their correctness must be assessed in every context after framing $R[\![(\cdot)]\!]$, for every principal R. One may also consider cross-module optimizations such that R varies.

(2) From the JIT compiler viewpoint, optimizations operate on expressions obtained by framing; this gives structural guarantees, such as the presence of a frame in every function.

(3) For later optimizations, such as runtime optimizations, one can no longer assume all expressions are obtained by framing.

In case (1), we are considering equations before framing, so we have to lift contextual equivalence, assuming a single, uniform but unknown frame. Accordingly, we introduce *front-end equivalence*, $e \llbracket\simeq\rrbracket e'$, defined as follows.

Front-end equivalence

$e \llbracket\simeq\rrbracket e'$ if and only if for all R, $R\llbracket e\rrbracket \simeq R\llbracket e'\rrbracket$.

5.1 Function Inlining

Code inlining is a fundamental program transformation, used by most global program optimizations.

Informally, inlining is problematic when it merges several frames that may have different permissions at runtime. For instance, when the caller and the inlined code have different static permissions, the inlined code is run with its caller's permissions. This effectively rules out cross-module inlining prior to setting the security policy.

In the following, we inline a function with principal R; we let $\mathcal{D}(\cdot)$ abbreviate the context $let\ h = R\llbracket\lambda x.e\rrbracket\ in\ \mathcal{C}(\cdot)$ and assume a preliminary renaming to prevent variable captures. Inlining of framed code may be described by the equation

$$\mathcal{D}(h\,v) \quad\longmapsto\quad \mathcal{D}(R\llbracket e\rrbracket\{x{\leftarrow}v\}) \tag{11}$$

that transforms a function call $h\,v$ into an inlined copy of the body e of h with v taking place of the formal parameter x—and thereby discards the inner frame. This differs from the literal inlining justified by equation (Fun Beta):

$$\begin{aligned} \mathcal{D}(h\,v) &\equiv \mathcal{D}(R\llbracket\lambda x.e\rrbracket\,v) \\ &\triangleq \mathcal{D}((\lambda x.R[R\llbracket e\rrbracket])\,v) \\ &\equiv \mathcal{D}(R[\,R\llbracket e\rrbracket\{x{\leftarrow}v\}]\,) \end{aligned} \tag{12}$$

This is correct in λ_{sec}, but leaves the frame $R[\cdot]$ around inlined code. Conversely, (11) may or may not be a contextual equivalence, depending on the context $\mathcal{D}$.

As a consequence, literal inlining before framing (as performed by a source compiler) is also problematic, even if $\lambda x.e$ and v have the same principal. In the case $v = R[\![w]\!]$, an instance of (11) is

$$\begin{array}{rl} e_0 \triangleq & \mathit{let}\ h = R[\![\lambda x.e]\!]\ \mathit{in}\ R[\![h\, w]\!] \\ \longmapsto\ e_1 \triangleq & \mathit{let}\ h = R[\![\lambda x.e]\!]\ \mathit{in}\ R[\![e\{x\leftarrow w\}]\!] \end{array}$$

Again, this transformation is not generally correct. Consider the inlined code $e = \mathit{grant}\ R\ \mathit{in}\ \mathit{test}\ R\ \mathit{then}\ \mathit{ok}\ \mathit{else}\ \mathit{fail}$. Assuming $R \neq \varnothing$, we have $\varnothing[e_0] \Downarrow \mathit{ok}$ versus $\varnothing[e_1] \Downarrow \mathit{fail}$. In contrast, we do have

$$R[\![\mathit{let}\ h = \lambda x.e\ \mathit{in}\ h\, w]\!] \quad \simeq \quad R[\![\mathit{let}\ h = \lambda x.e\ \mathit{in}\ e\{x\leftarrow w\}]\!]$$

because our encoding of let, followed by framing, introduces an extra frame $R[\cdot]$ on both sides of the equation, which enable us to apply equation (Frame Frame). In the following, we extend the framing translation $R[\![\cdot]\!]$ from frameless expressions to frameless contexts, with $R[\![(\cdot)]\!] \triangleq (\cdot)$. Unfolding our definitions, we obtain

$$R[\![\mathit{let}\ h = \lambda x.e\ \mathit{in}\ \cdot]\!] \quad \triangleq \quad \mathit{let}\ h = R[\![\lambda x.e]\!]\ \mathit{in}\ R[\cdot]$$

We have a more general correctness result for inlining before framing, which justifies a limited form of (11):

Lemma 1 (Local Inlining) *For all expressions e, values w, and contexts $\mathcal{B}$ in the frameless λ_{sec}, we have*

$$\mathit{let}\ h = \lambda x.e\ \mathit{in}\ \mathcal{B}(h\, w) \quad [\![\simeq]\!] \quad \mathit{let}\ h = \lambda x.e\ \mathit{in}\ \mathcal{B}(e\{x\leftarrow w\})$$

Proof We let $\mathcal{C}(\cdot) = R[R[\![\mathcal{B}(\cdot)]\!]]$ and keep the same definitions for v and $\mathcal{D}$ as above: $v = R[\![w]\!]$ and $\mathcal{D}(\cdot) \triangleq \mathit{let}\ h = R[\![\lambda x.e]\!]\ \mathit{in}\ \mathcal{C}(\cdot)$.

By definition of framing, we have $R[\![\mathit{let}\ h = \lambda x.e\ \mathit{in}\ \mathcal{B}(\cdot)]\!] \triangleq \mathcal{D}(\cdot)$. By definition of $[\![\simeq]\!]$, we can thus rewrite the statement of the lemma as an instance of the problematic inlining (11): for all R, $\mathcal{D}(R[\![h\, w]\!]) \ \simeq\ \mathcal{D}(R[\![e\{x\leftarrow w\}]\!])$, that is, $\mathcal{D}(h\, v) \ \simeq\ \mathcal{D}(R[\![e]\!]\{x\leftarrow v\})$.

Using the literal inlining equation (12), we already have the equivalence $\mathcal{D}(h\, v) \simeq \mathcal{D}(R[\, R[\![e]\!]\{x\leftarrow v\}])$, so it suffices to show

$$\mathcal{D}(R[\![e]\!]\{x\leftarrow v\}) \simeq \mathcal{D}(R[\, R[\![e]\!]\{x\leftarrow v\}])$$

Since $\simeq$ is a congruence, we can discard the binding for h and it suffices to show $\mathcal{C}(R[u]) \simeq \mathcal{C}(u)$, that is

$$R[R[\![\mathcal{B}]\!](R[u])] \quad \simeq \quad R[R[\![\mathcal{B}]\!](u)] \tag{13}$$

for all principals R, expressions u, and frameless context $\mathcal{B}$. The proof of equation (13) is by structural induction on $\mathcal{B}$:

$\mathcal{B}(\cdot) = (\cdot)$: equation $R[R[u]] \simeq R[u]$ is rule (Frame Dup).

$\mathcal{B}(\cdot) = x$, $\mathcal{B}(\cdot) = \mathit{fail}$: equations $R[x] \simeq R[x]$ and $R[\mathit{fail}] \simeq R[\mathit{fail}]$ are instances of rule (Eq Refl).

$\mathcal{B}(\cdot) = \lambda x.\mathcal{B}_1(\cdot)$:

$$\begin{array}{ll} & R[R[\![\lambda x.\mathcal{B}_1]\!](R[u])] \\ \triangleq & R[\lambda x.R[R[\![\mathcal{B}_1]\!]](R[u])] \\ \simeq & R[\lambda x.R[R[\![\mathcal{B}_1]\!]](u)] \text{ by induction hypothesis in context } R[\lambda x.(\cdot)] \\ \triangleq & R[R[\![\lambda x.\mathcal{B}_1]\!](u)] \end{array}$$

$\mathcal{B}(\cdot) = \mathcal{B}_1(\cdot)\, \mathcal{B}_2(\cdot)$:

$$\begin{array}{ll} & R[R[\![\mathcal{B}_1\, \mathcal{B}_2]\!](R[u])] \\ \triangleq & R[\big(R[\![\mathcal{B}_1]\!](R[u])\big)\ \big(R[\![\mathcal{B}_2]\!](R[u])\big)] \\ \simeq & R[\big(R[R[\![\mathcal{B}_1]\!](R[u])]\big)\ \big(R[R[\![\mathcal{B}_2]\!](R[u])]\big)] \text{ by (Frame Appl)} \\ \simeq & R[\big(R[R[\![\mathcal{B}_1]\!](u)]\big)\ \big(R[R[\![\mathcal{B}_2]\!](u)]\big)] \text{ by induction hypothesis (twice)} \\ \simeq & R[\big(R[\![\mathcal{B}_1]\!](u)\big)\ \big(R[\![\mathcal{B}_2]\!](u)\big)] \text{ by (Frame Appl)} \\ \triangleq & R[R[\![\mathcal{B}_1\, \mathcal{B}_2]\!](u)] \end{array}$$

$\mathcal{B}(\cdot) = \mathit{test}\ S\ \mathit{then}\ \mathcal{B}_1(\cdot)\ \mathit{else}\ \mathcal{B}_2(\cdot)$: First assume $S \subseteq R$.

$$\begin{array}{ll} & R[R[\![\mathit{test}\ S\ \mathit{then}\ \mathcal{B}_1\ \mathit{else}\ \mathcal{B}_2]\!](R[u])] \\ \triangleq & R[\mathit{test}\ S\ \mathit{then}\ R[\![\mathcal{B}_1]\!](R[u])\ \mathit{else}\ R[\![\mathcal{B}_2]\!](R[u])] \\ \simeq & \mathit{test}\ S\ \mathit{then}\ R[R[\![\mathcal{B}_1]\!](R[u])]\ \mathit{else}\ R[R[\![\mathcal{B}_2]\!]](R[u]) \\ & \text{by (Frame Test Then)} \\ \simeq & \mathit{test}\ S\ \mathit{then}\ R[R[\![\mathcal{B}_1]\!](u)]\ \mathit{else}\ R[R[\![\mathcal{B}_2]\!](u)] \\ & \text{by induction hypothesis (twice)} \\ \simeq & R[\mathit{test}\ S\ \mathit{then}\ R[\![\mathcal{B}_1]\!](u)\ \mathit{else}\ R[\![\mathcal{B}_2]\!](u)] \text{ by (Frame Test Then)} \\ \triangleq & R[R[\![\mathit{test}\ S\ \mathit{then}\ \mathcal{B}_1\ \mathit{else}\ \mathcal{B}_2]\!](u)] \end{array}$$

The case $\neg(S \subseteq R)$ is as above, using rule (Frame Test Else) before and after the induction hypothesis.

$\mathcal{B}(\cdot) = grant\ S\ in\ \mathcal{B}_1(\cdot)$:

$$\begin{array}{ll} & R[R[\![grant\ S\ in\ \mathcal{B}_1]\!](R[u])] \\ \triangleq & R[grant\ S\ in\ R[\![\mathcal{B}_1]\!](R[u])] \\ \simeq & R[grant\ S \cap R\ in\ R[R[\![\mathcal{B}_1]\!]](R[u])] \text{ by (Frame Grant } \cap) \\ \simeq & R[grant\ S \cap R\ in\ R[R[\![\mathcal{B}_1]\!]](u)] \text{ by induction hypothesis} \\ \simeq & R[grant\ S \cap R\ in\ R[R[\![\mathcal{B}_1]\!]](u)] \text{ by (Frame Grant } \cap) \\ \triangleq & R[R[\![grant\ S\ in\ \mathcal{B}_1]\!](u)] \end{array}$$

□

5.2 Tail Call Elimination

Tail call elimination is a useful optimization which also affects the structure of the stack. Instead of building a new frame for the last call in a function, the optimization overwrites the current frame so that the callee directly returns to the caller's caller. In the CLR, for instance, this may occur when the call is annotated as "tail callable" in the code [8], and the decision is made by the JIT compiler according to the security policy.

Informally, optimizing a tail call may create two problems: an untrusted caller may thereby remove its tracks from the calling stack; less importantly, perhaps, a trusted caller may inadvertently cancel permissions it has just granted. For these reasons, most implementations of stack inspection disallow or restrict tail calls. Various workarounds have been proposed [6, 29].

In our model, we reflect tail call elimination as a runtime transformation just before the call, rather than a specific language construct:

$$R[v\,w] \quad \longmapsto \quad v\,w \tag{14}$$

in some evaluation context or, more generally for callers that grant permissions, $R[grant\ S\ in\ v\,w] \longmapsto v\,w$. As in Section 2, we interpret (Red Frame) reduction steps as popping a runtime frame from the evaluation stack. With an ordinary call, the frame R is kept until $v\,w$ completes, whereas it is immediately discarded with the tail call optimization. For instance, if the callee is of the form $v = \lambda x.S[e]$, compare:

$$\begin{array}{lcll} R[v\,w] & \rightarrow & R[S[e\{x\leftarrow w\}]] & \text{ordinary call} \\ R[v\,w] & \longmapsto\rightarrow & S[e\{x\leftarrow w\}] & \text{optimized call} \end{array}$$

As with inlining, a frame is erased, but one level deeper in the stack. Clearly, (14) may not preserve contextual equivalence: we can formulate the two problems above as inequations. First, with examples (4) and (5) of Section 3, we have:

$$\begin{array}{rl} & \mathit{System}[\mathit{Applet}[\mathit{displayFile}\ \text{“secrets”}]] \\ \longmapsto & \mathit{System}[\mathit{displayFile}\ \text{“secrets”}] \end{array}$$

and the permission check fails only in the first expression, leading to different outcomes. Second, with example (6) from Section 3, we have

$$\begin{array}{rl} & \mathit{Applet}[\mathit{readVersion}\ \mathit{ok}] \\ \rightarrow & \mathit{Applet}[\mathit{System}[\mathit{grant}\ \{\mathit{fileIO}\}\ \mathit{in}\ \mathit{readFile}\ \text{“version”}]] \\ \longmapsto & \mathit{Applet}[\mathit{readFile}\ \text{“version”}] \end{array}$$

and the latter expression fails instead of returning the string “Build 2601”.

Fortunately, tail call elimination is actually correct in most common cases. For instance:

- Assume the callee has at most the static permissions of the caller, that is, $v = \lambda x.S[e]$ with $S \subseteq R$. We have:

  $$\begin{array}{rl} \multicolumn{2}{l}{R[v\ w]} \\ = & R[(\lambda x.S[e])\ w] \\ \simeq & R[S[e\{x \leftarrow w\}]] \text{ by (Fun Beta)} \\ \simeq & S[e\{x \leftarrow w\}] \text{ by (Frame Frame)} \\ = & (S[e])\{x \leftarrow w\} \\ \simeq & v\ w \text{ by (Fun Beta)} \end{array}$$

  In particular, any tail call within the same component can be optimized as long as the caller does not grant permissions.

- Even if the caller grants permissions T, and as long as both the static permissions of the callee and the granted permissions are statically given to the caller ($T \cup S \subseteq R$), the runtime may still be able to copy the grant to the new frame. With the same notations, let v' be v with the same additional grant ($v' = \lambda x.S[\mathit{grant}\ T\ \mathit{in}\ e]$). We have:

  $$\begin{array}{rl} \multicolumn{2}{l}{R[\mathit{grant}\ T\ \mathit{in}\ v\ w]} \\ = & R[\mathit{grant}\ T\ \mathit{in}\ (\lambda x.S[e])\ w] \\ \simeq & R[\mathit{grant}\ T\ \mathit{in}\ S[e\{x \leftarrow w\}]] \text{ by (Fun Beta)} \\ \simeq & R[S[\mathit{grant}\ T\ \mathit{in}\ e\{x \leftarrow w\}]] \text{ by (Frame Grant Frame)} \\ \simeq & S[\mathit{grant}\ T\ \mathit{in}\ e\{x \leftarrow w\}] \text{ by (Frame Frame)} \\ = & (S[\mathit{grant}\ T\ \mathit{in}\ e])\{x \leftarrow w\} \\ \simeq & v'\ w \text{ by (Fun Beta)} \end{array}$$

6 Keeping Track of Dependencies

Informally, stack inspection is a mechanism that prevents untrusted code from causing harm. However, it is surprisingly hard to state a useful theorem that captures this intent for a general class of trusted and untrusted code. We give it a try, and also explore variants of the operational semantics that yield stronger, easier-to-explain theorems. Our results are meant to illustrate these semantics, rather than provide the most general statements.

6.1 What is Guaranteed by Stack Inspection?

A first problem is that there is no generic notion of “something bad happens”. To this end, we re-interpret failures (*fail*) as security failures, rather than security exceptions. That is, we define “e does dangerous things” as $e \Downarrow \mathit{fail}$.

In the following, $S \subseteq \mathcal{P}$ represents an upper bound on the permissions effectively given to untrusted code. We introduce syntactic restrictions required in the results below, for any code (both trusted and untrusted).

Syntactic requirements

An expression e is safe against S when
(1) $\mathit{grant}\ R\ \mathit{in}\ e'$ occurs only with $R \subseteq S$.
(2) fail occurs only as $\mathit{test}\ R\ \mathit{then}\ \mathit{fail}\ \mathit{else}\ e'$ with $R \not\subseteq S$.

Conservatively, *fail* in (2) stands for any potentially dangerous code protected by R, such as primRF in the examples, and (1) rules out any dangerous grant.

Lemma 2 *Assume e is safe against S and $e \rightarrow^{U}_{V} e'$.*

(1) *Either $e' \Downarrow \mathit{fail}$ or e' is safe against S.*

(2) *If $V \subseteq S$, then e' is safe against S.*

Proof We detail the proof of statement (2) of the lemma. Syntactic requirement (1) is clearly preserved by any series of reductions. Requirement (2) is established by induction on the depth of the derivation, for all $V \subseteq S$.

- (Red Test): if the second branch is taken, e' is clearly safe against S. Since $V \subseteq S$ and $R \not\subseteq S$, the test yields $R \not\subseteq V$ hence the *fail*-branch is never taken.

- (Fail Rator) and (Fail Rand) are excluded by hypothesis.

- (Red Appl), (Red Frame), and (Red Grant) preserve requirement (2).
- (Ctx Rator), (Ctx Rand): by induction hypothesis for the same set V.
- (Ctx Frame): $R[e] \to_V^U R[e']$ using (Ctx Frame) if and only if $e \to_{V \cap R}^R e'$, so we apply the induction hypothesis to e for $V \cap R \subseteq S$.
- (Ctx Grant): *grant* R *in* $e \to_V^U$ *grant* R *in* e' using (Ctx Grant) if and only if $e \to_{V \cup (U \cap R)}^U e'$. We have $V \subseteq S$ by hypothesis and $R \subseteq S$ by safety of e against S, so we apply the induction hypothesis to e for $V \cup (U \cap R) \subseteq S$.

The proof of statement (1) is similar. We use the hypothesis $\neg(e' \Downarrow \mathit{fail})$ instead of $V \subseteq S$. Since "*fail* occurs in evaluation context in e' " implies $e' \Downarrow \mathit{fail}$, the first branch is never taken in (Red Test). $\square$

As a direct corollary, we obtain:

Theorem 5 (Sandbox) *If e is safe against S, then $S[e]$ does not fail.*

Proof Assume e is safe against S and $S[e] \to f$. There are two cases:

- e is an outcome o and $S[e] \to o$ using (Red Frame). We have $e \Downarrow o$ and $o \neq \mathit{fail}$ (since o is safe against S), hence e does not fail.
- $e \to_S^S e'$ and $f = S[e']$ using (Ctx Frame). Then e' is also safe against S by Lemma 2 for $V = S$.

This excludes any series of steps $S[e] \to^* \mathit{fail}$. $\square$

This basic result states that applets do nothing dangerous on their own, but does not capture the behaviour of a system that runs $S[e]$ in a more trusted environment, as illustrated in Section 3. Rather, it describes a sandbox policy with maximal permissions S. Such a policy can be enforced without the complications of dynamic stack inspections, using the constant set of permissions S or relying on types [18].

Next, we focus on trusted code that discards any untrusted result. With this discipline, applet code framed with S should not affect any code protected by permissions beyond S. The next theorem formalizes this reasonable property. Its statement relies on a partial erasure operator:

Partial erasure of untrusted code

Let $S \subseteq \mathcal{P}$. The function on terms $(\cdot) \setminus S$ is defined by
- $(S[e]; e') \setminus S \;\triangleq\; ok; (e' \setminus S)$
- $(\cdot) \setminus S$ otherwise commutes with all constructors.

The intent of the erasure is to make independence from the untrusted subterms syntactically obvious. We erase code that is framed with the permission set S exactly. However, we can apply our theorems several times with different S parameters to erase more code, and conversely we can add an extra permission to S and to some S-frames for a more selective erasure.

In general, erasure and evaluation do not commute, because diverging or failing computations may be erased. In our setting, we have:

Theorem 6 (Protection from untrusted procedures) *Assume e is safe against S. If $e \Downarrow o$, then $e \setminus S \Downarrow o \setminus S$.*

Hence, if $e \Downarrow \mathit{fail}$, then also $e \setminus S \Downarrow \mathit{fail}$ on its own. Informally, security failures do not depend on any untrusted code that is erased.

Proof By induction on the length of the derivation before erasure $e_0 \rightarrow^* o$ (for all expressions e_0) and case analysis on the first step $e_0 \rightarrow e_1$.

- $e_0 = \mathcal{C}(S[e]; f) \rightarrow e_1 = \mathcal{C}(S[e']; f)$ using $e \rightarrow_V^S e'$. Informally, the step occurs within an erased frame and is discarded by the erasure.

  By Lemma 2, e' is still safe against S, hence e_1 is also safe against S. If $e_0 \Downarrow o$, then $e_1 \Downarrow o$ and, by induction hypothesis, $e_1 \setminus S \Downarrow o$. Finally, $e_0 \setminus S = e_1 \setminus S$ and thus $e_0 \setminus S \Downarrow o$.

- We are not in the case above and $e_0 = \mathcal{C}(S[o]; f) \rightarrow e_1 = \mathcal{C}(o; f)$. Informally, a step (Red Frame) removes the boundary of a frame erased by the translation.

  Since e_0 is safe against S, o is also safe, thus $o \neq \mathit{fail}$ and we have

$$\begin{array}{rcl} e_0 \rightarrow e_1 & \rightarrow & \mathcal{C}(f) \\ e_0 \setminus S = (\mathcal{C} \setminus S)(ok; f \setminus S) & \rightarrow & \mathcal{C}(f) \setminus S \end{array}$$

  If $e_0 \Downarrow o$, then $\mathcal{C}(f) \Downarrow o$. Since $\mathcal{C}(f)$ is also safe against S, we have $\mathcal{C}(f) \setminus S \Downarrow o$ by induction hypothesis, and thus $e_0 \setminus S \Downarrow o$.

- We are not in the cases above, i.e. $e_0 = \mathcal{C}(e) \rightarrow e_1 = \mathcal{C}(e')$ for some evaluation context $\mathcal{C}$ and some instance $e \rightarrow_V^U e'$ of a base reduction rule. Informally, such steps always commute with the erasure.

  We show $e_0 \setminus S \rightarrow e_1 \setminus S$ using an instance $e \setminus S \rightarrow_V^U e' \setminus S$ of the same base rule. Base cases (Red Appl), (Fail Rator), (Fail Rand), and (Red Grant) are immediate.

Base cases (Red Frame) and (Red Test) follow from the absence of erased frames $S[\cdot]$ in evaluation context in $\mathcal{C}$ (otherwise one of the two cases above apply). In particular, this guarantees that tests in context $\mathcal{C}$ and $(\mathcal{C} \setminus S)(\cdot)$ always agree. $\square$

As can be expected from our examples, the theorem would not hold for a more general erasure operator that may discard untrusted expressions whose results are actually used by trusted code.

The theorem does not distinguish between trusted and untrusted code. Indeed, an erased frame $S[e]$ may contain both trusted and untrusted parts; such frames naturally occur by reduction from the initial configurations obtained by framing, described in Section 2.3.

Due to its strict syntactic requirements, Theorem 6 may not immediately apply to these configurations, but we can use our equational theory to rearrange them. Specifically:

(1) As a prerequisite, both trusted and untrusted code must be safe against S. In the case untrusted code contains grants of permissions not in S, one can sometimes apply equations (Frame Grant) and (Grant Frame) to lower those grants and meet requirement (1).

(2) The theorem is useful inasmuch as untrusted frames are discarded. Hence, S frames should be moved into contexts such that $(\cdot) \setminus S$ erases them, when possible.

Typically, after framing untrusted code, S frames appear under abstractions rather than in contexts $(\cdot); e$. Consider, for instance, an expression that links trusted code $(z\, e); e'$ and untrusted code $S[\![v]\!] = \lambda x.S[e'']$ for some $x \notin fv(e\, e')$. We have:

$$\begin{array}{rcl} (\lambda z.(z\, e); e')\ \lambda x.S[e''] & \equiv & ((\lambda x.S[e''])\ e); e' \\ & \triangleq & (let\ x = e\ in\ S[e'']); e' \\ & \equiv & let\ x = e\ in\ (S[e'']; e') \\ & (\cdot) \setminus S & let\ x = e \setminus S\ in\ (ok; e' \setminus S) \\ & \equiv & (\lambda z.(z\, e); e') \setminus S\ \lambda x.ok \end{array}$$

applying first equations (Fun Beta) and (Let Let), then erasing the S frame, and finally applying those equations again. Thus, we can extend Theorem 6 to a stronger notion of erasure that embeds this pattern.

(3) After applying the theorem, if there is any residual untrusted code, such as functions whose results are not discarded, some more equational reasoning may be required to assess their effect on the computation.

An interesting approach to obtain similar guarantees (and to benefit further from stack inspection) is to modify the interface between trusted and untrusted code. For instance, one can perform a local continuation-passing style transform (CPS) on untrusted functions: whenever the results of untrusted applets are used in trusted code, one can instead pass the result to a trusted continuation. (While it is tempting to apply a global CPS, this has little practical interest, inasmuch as its effective implementation rules out the stack-based, on demand inspection algorithm.)

For example, if the expression $(e_1\ S[f]); e_2$ is modified by CPS-transform into $(\lambda\kappa.(S[\kappa\ f]; e_2))\ e_1$, and as long as the whole expression is safe against S, we can erase f and apply Theorem 6 to show that the outcome of the expression does not depend on f. However, this modification is not a contextual equivalence in λ_{sec}.

6.2 Tracking all Call-by-Value Dependencies

To get a better understanding of the limitations of stack inspection, we now consider alternative operational semantics that keep track of dependencies more systematically.

We let extended values be values within frames and grants:

Grammar for extended values

$w ::=$	extended value
v	value
$R[w]$	framed value
$\mathit{grant}\ R\ \mathit{in}\ w$	privileged value

For every w, we have $w\ (\to_D^S)^*\ v$ for a unique v that does not depend on S or D—we just repeatedly apply steps (Red Frame) and (Red Grant)—and moreover $v \simeq w$ by rules (Frame o) and (Grant o). Hence, we could substitute extended values for values for the semantics given in Section 2 without affecting contextual equivalence.

For simplicity, in the following we only consider λ_{sec} without permission grants. (We believe that our alternative semantics can be generalized to deal with grants in a reasonable way.)

Our first modified semantics keeps track of all dependencies, much like information-flow.

Reduction rules for CBV dependency tracking

(Red Frame), (Ctx Rand), and (Fail Rand) are replaced by:

(Red Frame Rand)
$v_1\ R[w_2] \rightarrow_D^S R[v_1\ w_2]$

(Ctx Rand W)
$$\frac{e_2 \rightarrow_D^S e_2'}{w_1\ e_2 \rightarrow_D^S w_1\ e_2'}$$

(Red Frame Rator)
$R[w_1]\ w_2 \rightarrow_D^S R[w_1\ w_2]$

(Fail Frame)
$R[\mathit{fail}] \rightarrow_D^S \mathit{fail}$

(Fail Rand W)
$w\ \mathit{fail} \rightarrow_D^S \mathit{fail}$

Other rules are unchanged from Section 2.2:

(Red Appl)
$(\lambda x.e)\ v \rightarrow_D^S e\{x \leftarrow v\}$

(Red Test)
$\mathit{test}\ R\ \mathit{then}\ e_{\mathrm{true}}\ \mathit{else}\ e_{\mathrm{false}} \rightarrow_D^S e_{R \subseteq D}$

(Ctx Rator)
$$\frac{e_1 \rightarrow_D^S e_1'}{e_1\ e_2 \rightarrow_D^S e_1'\ e_2}$$

(Ctx Frame)
$$\frac{e \rightarrow_{D \cap R}^R e'}{R[e] \rightarrow_D^S R[e']}$$

(Fail Rator)
$\mathit{fail}\ e \rightarrow_D^S \mathit{fail}$

Rules (Red Frame Rand), (Red Frame Rator), and (Fail Frame) refine rule (Red Frame) with three disjoint cases. The net effect of the refined semantics is to accumulate every frame that ever occurs in evaluation context, instead of discarding frames after local evaluation. (Arbitrarily, the frames for the operator are placed outside those for the operand, but this is not observable in the absence of grants.)

Pragmatically, this variant is much harder to implement lazily: stack inspection must be supplemented with a mechanism that captures the current security environment and attaches it to any value. Conversely, a security-passing style implementation of the λ-calculus, at least, could easily accommodate this variation.

Rules (Red Frame Rand) and (Red Frame Rator) for frames correspond to the two operational rules for labels in the call-by-value semantics given by Abadi, Lampson, and Lévy in [1, section 3.7]. Their semantics also strictly keep track of dependencies, although their intent is quite different.

With our modified semantics, we have a stronger, simpler variant of Theorem 6. We redefine the erasure operator as follows: $S[e] \setminus S = S[\mathit{ok}]$, and $(\cdot) \setminus S$ commutes with all other constructors. Hence, we uniformly erase untrusted code, independently of its usage.

Theorem 7 (Independence from untrusted code) *Assume e is safe against S. With the dependency tracking semantics above, we have:*

$e \Downarrow \mathit{fail} \iff e \setminus S \Downarrow \mathit{fail}$

$e \Downarrow w \iff e \setminus S \Downarrow w \setminus S$ *for any extended value w not framed by S.*

The first claim of the theorem asserts that failures in e do not depend on any S-framed code. Less importantly, perhaps, the second claim describes computations that do not use S-framed code.

Before proving the theorem, we establish some basic properties of the dependency-tracking semantics. With this semantics, evaluation contexts now have the grammar $\mathcal{E}(\cdot) ::= \cdot \mid \mathcal{E}(\cdot)\, e \mid w\, \mathcal{E}(\cdot) \mid R[\mathcal{E}(\cdot)]$.

We say that an expression is S-*framed* when it is of the form $\mathcal{E}(S[e])$ for some evaluation context $\mathcal{E}$. Similarly, a context is S-framed when it is of the form $\mathcal{E}_1(S[\mathcal{E}_2(\cdot)])$ for some evaluation contexts $\mathcal{E}_1$ and $\mathcal{E}_2$.

Lemma 3 (Erasure in Framed Expressions)

(1) *e is S-framed if and only if $e \setminus S$ is S-framed.*

(2) *If e is not S-framed, then $e \rightarrow^U_V e' \iff e \setminus S \rightarrow^U_V e' \setminus S$.*

(3) *If $e = \mathcal{E}(fail)$ for some evaluation context $\mathcal{E}$ that is not S-framed, then both $e \Downarrow fail$ and $e \setminus S \Downarrow fail$.*

Proof (1) follows from the definition of $\cdot \setminus S$.

We prove (2) by induction on the derivation of $e \rightarrow^U_V$; since e is not S-framed, we have $R \neq S$ for the rules (Ctx Frame), (Red Frame Rand), (Red Frame Rator), (Fail Frame). The other rules are immediate.

We prove (3) by induction on the depth of $\mathcal{E}$. For each constructor, we apply (Fail Rator), (Fail Rand W), or (Fail Frame) until we obtain *fail*. $\square$

Lemma 4 (Reduction for Safe Expressions) *Let e be safe against S and assume $e \rightarrow^U_V e'$. We have the following properties:*

(1) *Either e' is also safe against S, or $e' = \mathcal{E}(fail)$ for some evaluation context $\mathcal{E}$ that is not S-framed.*

(2) *If e is S-framed, then e' is S-framed and safe against S.*

Proof The two proofs are by induction on the derivation of $\rightarrow^U_V$.

(1) We easily check that any reduction yields an expression e' safe against S, except perhaps (Red Test) on *test R then fail else f*. Since $R \not\subseteq S$ by safety requirement, the else branch is always chosen when the test occurs in an S-framed context, so *fail* never appears in S-framed evaluation contexts.

(2) The only reduction rule that may discard an S-frame in evaluation context is (Fail Frame). This rule never applies in expressions safe against S. $\square$

Proof of Theorem 7 Let e be safe against S, and consider its (finite or infinite) series of derivatives $e = e_0 \to e_1 \to \cdots \to e_i \to \dots$. We distinguish three cases:

e **becomes S-framed** : for some $n \geq 0$, there exists $e = e_0 \to \cdots \to e_{n-1} \to e_n$ such that $e_0, \dots\dots e_{n-1}$ are all safe against S but not S-framed, while e_n is both safe against S and S-framed.

By Lemma 3(2) we obtain $e \setminus S \to^n e_n \setminus S$. The expression after erasure $e_n \setminus S$ is also safe against S and, by Lemma 3(1), it is also S-framed. Applying Lemma 4(2) to e_n and $e_n \setminus S$, these two expressions always remain S-framed and safe against S; they may independently diverge or converge to any S-framed extended values.

e **becomes unsafe** : for some $n \geq 0$, there exists $e = e_0 \to \cdots \to e_n \to f$ such that $e_0, \dots, e_n$ are all safe against S and not S-framed, whereas f is not safe against S.

By Lemma 3(2) we obtain $e \setminus S \to^{n+1} f \setminus S$.
By Lemma 4(1), we have $f = \mathcal{E}(\mathit{fail})$ for some evaluation context $\mathcal{E}$ that is not S-framed. By Lemma 3(3), we have $f \Downarrow \mathit{fail}$ and $f \setminus S \Downarrow \mathit{fail}$, and thus $e \Downarrow \mathit{fail}$ and $e \setminus S \Downarrow \mathit{fail}$.

e **remains safe and not S-framed** : for all i, e_i is safe against S but not S-framed. By Lemma 3(2) we obtain $e \setminus S \to^i e_i \setminus S$. Thus, either e and $e \setminus S$ diverge, or we have $e \Downarrow w$ for some w that is not S-framed and, since $e \setminus S$ is also an extended value, $e \setminus S \Downarrow w \setminus S$. □

6.3 Two Intermediate Tracking Semantics

Starting from the semantics for CBV dependency tracking, we can give up the preservation of convergence and get a coarser semantics by (1) discarding rule (Red Frame Rand), and (2) generalizing (Red Appl) to substitute framed values. This is similar in spirit to the first labelled semantics of [1], where labels are parts of values.

Reduction rules with framed values

(Red Appl) is replaced by (Red Appl W) $(\lambda x.e)\ w \to^S_D e\{x \leftarrow w\}$

Other rules are unchanged from Sections 2.2 and 6.2:

(Ctx Rator) $\dfrac{e_1 \to^S_D e_1'}{e_1\ e_2 \to^S_D e_1'\ e_2}$ (Ctx Rand W) $\dfrac{e_2 \to^S_D e_2'}{w_1\ e_2 \to^S_D w_1\ e_2'}$ (Ctx Frame) $\dfrac{e \to^R_{D \cap R} e'}{R[e] \to^S_D R[e']}$

(Red Test)
$\mathit{test}\ R\ \mathit{then}\ e_{\text{true}}\ \mathit{else}\ e_{\text{false}} \to^S_D e_{R\subseteq D}$

(Red Frame Rator)
$R[w_1]\ w_2 \to^S_D R[w_1\ w_2]$

(Fail Frame)
$R[\mathit{fail}] \to^S_D \mathit{fail}$

(Fail Rator)
$\mathit{fail}\ e \to^S_D \mathit{fail}$

(Fail Rand W)
$w\ \mathit{fail} \to^S_D \mathit{fail}$

Alternatively, we can obtain a similar semantics without modifying (Red Appl) by pushing the frame constructors under abstractions instead of discarding them.

Reduction rules with frame capture in functions

(Red Frame) is replaced by (Red Frame Fun)
$R[\lambda x.e] \to^S_D \lambda x.R[e]$

Other rules are unchanged from Sections 2.2 and 6.2:

(Ctx Rator)
$$\frac{e_1 \to^S_D e_1'}{e_1\ e_2 \to^S_D e_1'\ e_2}$$

(Ctx Rand)
$$\frac{e_2 \to^S_D e_2'}{v_1\ e_2 \to^S_D v_1\ e_2'}$$

(Ctx Frame)
$$\frac{e \to^R_{D\cap R} e'}{R[e] \to^S_D R[e']}$$

(Red Appl)
$(\lambda x.e)\ v \to^S_D e\{x{\leftarrow}v\}$

(Red Test)
$\mathit{test}\ R\ \mathit{then}\ e_{\text{true}}\ \mathit{else}\ e_{\text{false}} \to^S_D e_{R\subseteq D}$

(Fail Frame)
$R[\mathit{fail}] \to^S_D \mathit{fail}$

(Fail Rator)
$\mathit{fail}\ e \to^S_D \mathit{fail}$

(Fail Rand)
$v\ \mathit{fail} \to^S_D \mathit{fail}$

These two intermediate semantics model the capture of the dynamic security environment (here D) that sometimes occurs in runtimes, for example when preparing the first call to a new thread. They are weaker than CBV dependency tracking; for instance, the divergence properties of low-privileged, unused subterms are not taken into account. For both of these semantics, we have $R[\lambda x.e] \simeq \lambda x.R[e]$ and so they are roughly equivalent.

We summarise our semantics variants by considering reductions for the expression $e_0 = (\lambda x.e)\ R[\lambda y.f]$, from the coarsest to the most restrictive: standard stack inspection; stack inspection with frame capture; stack inspection with framed values; and CBV dependency tracking.

$$\begin{array}{ll}
e_0 \xrightarrow{\text{(Red Frame)}}\xrightarrow{\text{(Red Appl)}} & e\{x{\leftarrow}\lambda y.f\} \\
e_0 \xrightarrow{\text{(Red Frame Fun)}}\xrightarrow{\text{(Red Appl)}} & e\{x{\leftarrow}\lambda y.R[f]\} \\
e_0 \xrightarrow{\text{(Red Appl W)}} & e\{x{\leftarrow}R[\lambda y.f]\} \\
e_0 \xrightarrow{\text{(Red Frame Rand)}}\xrightarrow{\text{(Red Appl)}} & R[e\{x{\leftarrow}\lambda y.f\}]
\end{array}$$

In order to get an adequate theorem for the intermediate semantics, we adapt again the erasure operator, as follows. To preserve convergence, we use a closed value t instead of *ok* such that $\lambda\_.t \simeq t$. We let $S[e] \setminus S = t$, and let $(\cdot) \setminus S$ commute with all other constructors.

Theorem 8 (Protection from untrusted code)
Assume e is safe against S. With any of the two semantics above, we have $e \Downarrow \mathit{fail} \Longrightarrow e \setminus S \Downarrow \mathit{fail}$.

Proof of Theorem 8 for the first variation (frame capture) The structure of the proof is similar to those for the previous semantics. Evaluation contexts are given by the grammar $\mathcal{E}(\cdot) ::= \cdot \mid \mathcal{E}(\cdot)\, e \mid v\, \mathcal{E}(\cdot) \mid R[\mathcal{E}(\cdot)]$.

We successively show that:

(1) Let e be safe against S. If $e \rightarrow^U_V e'$, then either e' is safe against S, or e' is of the form $\mathcal{E}(\mathit{fail})$ for some evaluation context $\mathcal{E}$ that is not S-framed.

(2) If *fail* occurs in evaluation context in e and is not S-framed, then $e \Downarrow \mathit{fail}$ and $e \setminus S \Downarrow \mathit{fail}$.

(3) Assume e is safe against S and $e \rightarrow e'$.
If $e' \setminus S \Downarrow \mathit{fail}$, then also $e \setminus S \Downarrow \mathit{fail}$.

The proof is by case on the derivation of $\rightarrow$. One of the following holds:

(a) e is not S-framed. Then, $e \setminus S \rightarrow e' \setminus S$ and thus $e \setminus S \Downarrow \mathit{fail}$.

(b) The step is $\mathcal{E}(S[f]) \rightarrow \mathcal{E}(S[f'])$ for some non-S-framed evaluation context $\mathcal{E}$. Then we have $e \setminus S = e' \setminus S$.

(c) The step is $\mathcal{E}(S[\lambda x.f]) \rightarrow \mathcal{E}(\lambda x.S[f])$ for some non-S-framed evaluation context $\mathcal{E}$, using (Red Frame Fun) in context $\mathcal{E}$.
Then, we have $e \setminus S = (\mathcal{E} \setminus S)(t)$ and $e' \setminus S = (\mathcal{E} \setminus S)(\lambda x.t)$. From the equation $t \simeq \lambda x.t$ in context $\mathcal{E} \setminus S$, we obtain $e \setminus S \Downarrow \mathit{fail}$.

(4) Assume e is safe against S and $e \Downarrow \mathit{fail}$. We obtain $e \setminus S \Downarrow \mathit{fail}$ by induction on the length of the derivation $e \rightarrow^n \mathit{fail}$. $\square$

Proof of Theorem 8 for the second variation (framed values) The proof has the same structure as above. In the case analysis, the specific case (3c) now uses rule (Red Frame Rator) with $R = S$.

The step is $\mathcal{E}(S[w_1]\ w_2) \rightarrow \mathcal{E}(S[w_1\ w_2])$. After the erasure, we have $e \setminus S = (\mathcal{E} \setminus S)(t\, w_2)$ versus $e' \setminus S = (\mathcal{E} \setminus S)(t)$. From the equation $t \simeq \lambda\_.t$, we obtain $(\mathcal{E} \setminus S)(t\, w_2) \simeq (\mathcal{E} \setminus S)((\lambda\_.t)\, w_2) \rightarrow (\mathcal{E} \setminus S)(t) = e' \setminus S$. $\square$

7 Conclusions and Related Work

We began the paper by casting doubt on the claims that stack inspection (1) allows easy and precise statement of security requirements and (2) is transparent for most programmers. To be clear, we are not denying the entirety of these claims; after all, stack inspection has been an effective security tool in runtimes like the JVM or the CLR.

Instead, we are probing its limitations. The limits of (1) appear in Sections 3 and 6 as we model complex interactions between trusted and untrusted code. The limits of (2) appear as we investigate standard program transformations in Sections 4 and 5. Although we use a formalism, we attempted throughout also to explain the issues in implementation terms. Inevitably, we leave aside important issues in the details of the implementations.

As well as casting doubt, the paper casts light on the semantics of stack inspection. The equational theory in Section 4.1 allows us to reason carefully about compiler transformations. The variations in Section 6 strike different balances between security requirements and their implementation cost. Still, these variations are exploratory, and so far purely theoretical. Implementation experiments remain future work. To the best of our knowledge, ours is the first work to analyse contextual equivalence in the presence of stack inspection, or to attempt to formulate high-level program-independent guarantees.

Wallach, Appel, and Felten [31] provide an alternative semantics, security-passing style, that makes explicit the security environment as an extra argument passed to every function; they clearly separate the security intent from its implementation mechanism; they also present a semantics in terms of authentication logic. Our security-indexed semantics amounts to a direct account of security-passing style.

Besson, Jensen, Le Métayer, and Thorn [16, 7] propose a logic for security properties of the control flow graph of a program. Their strategy is to identify specific properties, construct a flow graph, and apply a model-checker. Their logic can express the behaviour of stack inspection as a formula. Their work is notable for its success in proving interesting program-dependent guarantees.

Erlingsson and Schneider [9] implement two formulations of stack inspection by constructing an inlined reference monitor. They informally outline shortcomings of stack inspection with respect to thread creation and method inheritance.

Pottier, Skalka, and Smith [28, 30] introduce the λ_{sec}-calculus in their work on avoiding dynamic stack inspections by type-based static analysis.

Their types express detailed information on permissions, which may be useful in a typed equational theory.

Banerjee and Naumann [3] develop an eager denotational semantics for a λ-calculus similar to λ_{sec}, and show its correspondence to a lazy operational semantics. They present a static analysis, similar to but more abstract than the analysis of Pottier, Skalka, and Smith, that can safely eliminate certain stack inspections. They identify program transformations validated by their denotational semantics; this is the only other work we know of to analyse program equivalence in the presence of stack inspection. In subsequent work, Banerjee and Naumann extend their denotational semantics to model stack inspection in a Java-like class-based language [4]. An abstraction theorem for their semantics is the basis for ongoing work on proving security properties of programs.

Karjoth [17] gives a detailed operational semantics of the stack inspection mechanism in Java 2, but does not consider the effect of stack inspection on code optimisations.

Bartoletti, Degano, and Ferrari [5] analyse bytecode to approximate the set of permissions effectively granted or denied at run-time, and use this information to optimize stack inspection mechanisms.

We discussed in Section 6 the view that stack inspection approximates a flow analysis. Several authors consider flow analyses for security. For instance, Ørbæk and Palsberg model trust in a pure λ-calculus supplemented with *trust*, *distrust*, and *check* constructors. Trust and distrust annotations remain attached to values, much like labels or S-frames in Section 6.2, but they can cancel one another, with for instance $\mathit{trust}\ (\mathit{distrust}\ e) \to \mathit{trust}\ e$. Their semantics does not fix a particular evaluation strategy. They provide a type system that rules out erroneous expressions $\mathit{check}\,(\mathit{distrust}\,e)$. Myers [25] also proposes a flow analysis for protecting privacy and integrity properties in Java programs.

Grossman, Morrisett, and Zdancewic [13] model multiple principals within a typed λ-calculus, with a reduction semantics similar to the tracking semantics of Section 6. They are not concerned with access control, but prove various safety and abstraction properties.

Acknowledgments

This work benefited from discussion with Martín Abadi, Tony Hoare, Butler Lampson, and Erik Meijer.

A Semantics with Explicit Stack Inspection

Pottier, Skalka, and Smith [28] give two different semantics for λ_{sec}. The first semantics gives an explicit account of stack inspection: it closely models the complex inspection mechanism that occurs on demand when testing permissions, as an inductive predicate on the current evaluation context. Still, modulo minor syntactic differences, we can prove that our top-level reduction relation equals their reduction relation with stack inspection (Corollary 2). In short, our definition is equivalent but more abstract.

Their second semantics is by translation to a standard λ-calculus plus primitive operations on permission sets. This is the security-passing style transformation proposed by Wallach, Appel, and Felten [31]. Our security-indexed operational semantics represents this style directly rather than by translation; the dynamic permissions set D in $\rightarrow_D^S$ is essentially the additional parameter in security-passing style.

We recall the first semantics given in [28] for our variant of λ_{sec}. The semantics is given as reduction steps in evaluation context. Crucially, permission tests depend on a stack-inspection predicate that takes the current context as a parameter. (Evaluation contexts $\mathcal{E}$ are defined in Section 2.2.) For simplicity, we describe stack inspection independently for each requested permission and aggregate the results in (SI Test).

Next, we relate this semantics to the one given in Section 2. The first lemma states that the sets S and D passed in reductions $\rightarrow_D^S$ collect the static and dynamic permissions that can be read on demand from the stack, in order to process a permission test. As a corollary, we obtain agreement between the two semantics.

Lemma 5 (Stack Inspection vs Security Passing) *Let $\mathcal{E}$ be an evaluation context. Let $S = \{p \mid \mathcal{E} \vdash_s p\}$ and $D = \{p \mid \mathcal{E} \vdash p\}$. We have $\mathcal{E}(e) \rightarrow \mathcal{E}(e') \iff e \rightarrow_D^S e'$*

Proof The proof is by induction on the depth of $\mathcal{E}$, for all e and e'.

The base case $\mathcal{E}(\cdot) = (\cdot)$ follows from the definition of top-level reductions $\rightarrow \stackrel{\Delta}{=} \rightarrow_{\mathcal{P}}^{\mathcal{P}}$. With the notations of the lemma, we have $D = S = \mathcal{P}$ by rules (Walk Top) and (Find Top), respectively.

Inductively, we consider the inner constructor in $\mathcal{E}$:

- $\mathcal{E}(\cdot) = \mathcal{E}'(\mathit{grant}\ T\ \mathit{in}\ \cdot)$. Let S and D be defined as in the lemma, and let S' and D' be the permissions within $\mathcal{E}'$, that is, $S' = \{p \mid \mathcal{E}'(\cdot) \vdash_s p\}$ and $D' = \{p \mid \mathcal{E}'(\cdot) \vdash p\}$. By definition of stack inspection, we have $S = S'$ using rule (Find Further) and $D = D' \cup (T \cap S)$ using rules (Walk Grant) and (Walk Further). In particular, $D \subseteq S \iff D' \subseteq S$.

Reduction relations with stack inspection

$e \xrightarrow{w} e'$	top-level reduction relation
$\mathcal{E} \vdash p$	the context $\mathcal{E}$ yields permission p
$\mathcal{E} \vdash_s p$	the context $\mathcal{E}$ yields static permission p

Operational semantics with stack inspection

(SI Appl)
$$\mathcal{E}((\lambda x.e)\ v) \xrightarrow{w} \mathcal{E}(e\{x \leftarrow v\})$$

(SI Fail)
$$\mathcal{E}(\mathit{fail}\ e) \xrightarrow{w} \mathcal{E}(\mathit{fail})$$
$$\mathcal{E}(v\ \mathit{fail}) \xrightarrow{w} \mathcal{E}(\mathit{fail})$$

(SI Frame)
$$\mathcal{E}(R[o]) \xrightarrow{w} \mathcal{E}(o)$$

(SI Grant)
$$\mathcal{E}(\mathit{grant}\ R\ \mathit{in}\ o) \xrightarrow{w} \mathcal{E}(o)$$

(SI Test)
$$\mathcal{E}(\mathit{test}\ R\ \mathit{then}\ e_{\mathrm{true}}\ \mathit{else}\ e_{\mathrm{false}}) \xrightarrow{w} \mathcal{E}(e_{(\forall p \in R\,.\,\mathcal{E} \vdash p)})$$

(Walk Frame)
$$\frac{\mathcal{E} \vdash p \quad p \in S}{\mathcal{E}(S[\cdot]) \vdash p}$$

(Walk Top)
$$(\cdot) \vdash p$$

(Walk Further)
$$\frac{\mathcal{E} \vdash p}{\begin{array}{c}\mathcal{E}(\cdot\ e) \vdash p \\ \mathcal{E}(v\ \cdot) \vdash p \\ \mathcal{E}(\mathit{grant}\ T\ \mathit{in}\ \cdot) \vdash p\end{array}}$$

(Walk Grant)
$$\frac{\mathcal{E} \vdash_s p \quad p \in T}{\mathcal{E}(\mathit{grant}\ T\ \mathit{in}\ \cdot) \vdash p}$$

(Find Frame)
$$\frac{p \in S}{\mathcal{E}(S[\cdot]) \vdash_s p}$$

(Find Top)
$$(\cdot) \vdash_s p$$

(Find Further)
$$\frac{\mathcal{E} \vdash_s p}{\begin{array}{c}\mathcal{E}(\cdot\ e) \vdash_s p \\ \mathcal{E}(v\ \cdot) \vdash_s p \\ \mathcal{E}(\mathit{grant}\ T\ \mathit{in}\ \cdot) \vdash_s p\end{array}}$$

By rule (Ctx Grant), $e \to_D^S e' \iff \textit{grant } T \textit{ in } e \to_{D'}^S \textit{grant } T \textit{ in } e'$. By induction hypothesis for $(\mathcal{E}', \textit{grant } T \textit{ in } e, \textit{grant } T \textit{ in } e')$, we have $\mathcal{E}'(\textit{grant } T \textit{ in } e) \to \mathcal{E}'(\textit{grant } T \textit{ in } e') \iff \textit{grant } T \textit{ in } e \to_{D'}^S \textit{grant } T \textit{ in } e'$. We thus obtain $\mathcal{E}(e) \to \mathcal{E}(e') \iff e \to_D^S e'$.

- $\mathcal{E}(\cdot) = \mathcal{E}'(R[\cdot])$. We use the same notations S, D, S', and D' as above. By definition of stack inspection, we have $S = R$ using rule (Find Frame) and $D = D' \cap R$ using rule (Walk Frame). By rule (Ctx Frame) we have $e \to_{D' \cap R}^R e'$ iff $R[e] \to_{D'}^{S'} R[e]$, and we conclude by induction hypothesis for $(\mathcal{E}', R[e], R[e'])$.
- $\mathcal{E}(\cdot) = \mathcal{E}'(\cdot\ e)$, and $\mathcal{E}(\cdot) = \mathcal{E}'(v\ \cdot)$. These cases are similar but simpler, since the constructor leaves S and D unchanged. □

Corollary 2 (Top-Level Agreement) $e \xrightarrow{w} e' \iff e \to e'$

Proof By definition, the stack inspection semantics has the same evaluation contexts as our small-step semantics, so we just have to compare the base reduction rules.

Let $\mathcal{E}$ be an evaluation context, and let S, D be obtained from $\mathcal{E}$ as in Lemma 5. We prove that $\mathcal{E}(e) \xrightarrow{w} \mathcal{E}(e')$ using a base rule (SI -) iff $e \to_D^S e'$ using a base rule (Red -). We then apply Lemma 5 and conclude $\mathcal{E}(e) \xrightarrow{w} \mathcal{E}(e')$ iff $\mathcal{E}(e) \to \mathcal{E}(e')$ for all $\mathcal{E}$, e, and e'.

We get a syntactic correspondence for the rules (SI Appl) and (Red Appl); (SI Fail) and (Fail Rand),(Fail Rator); (SI Frame) and (Red Frame); (SI Grant) and (Red Grant). We detail the rules (Red Test) and (SI Test), since their respective test predicates are expressed differently:

(Red Test)
$$\textit{test } R \textit{ then } e_{\text{true}} \textit{ else } e_{\text{false}} \to_D^S e_{R \subseteq D}$$
(SI Test)
$$\mathcal{E}(\textit{test } R \textit{ then } e_{\text{true}} \textit{ else } e_{\text{false}}) \xrightarrow{w} \mathcal{E}(e_{(\forall p \in R\,.\,\mathcal{E} \vdash p)})$$

By definition, $D \triangleq \{p \mid \mathcal{E} \vdash p\}$, so the two predicates always agree. □

B Security-Indexed Evaluation Semantics

In Section 4.3, we defined a security-indexed evaluation relation, $e \Downarrow_D^S o$, in terms of the security-indexed reduction relation, $e \to_D^S e'$. We defined $e \Downarrow_D^S o$ to mean $e(\to_D^S)^* o$ and $D \subseteq S$. This section presents an alternative characterization—a direct inductive definition. The associated induction principle is useful for the proofs in Appendix C. We allow $e \Downarrow_D^S o$ only when $D \subseteq S$.

Evaluation relation

$e \Downarrow^S_D o$	security-indexed evaluation ($D \subseteq S$)

Security-indexed evaluation rules

(Eval Outcome)
$$\frac{}{o \Downarrow^S_D o}$$

(Eval Appl)
$$\frac{e_1 \Downarrow^S_D \lambda x.e_3 \quad e_2 \Downarrow^S_D v \quad e_3\{x \leftarrow v\} \Downarrow^S_D o}{e_1\, e_2 \Downarrow^S_D o}$$

(Eval Rator Fail)
$$\frac{e_1 \Downarrow^S_D \mathit{fail}}{e_1\, e_2 \Downarrow^S_D \mathit{fail}}$$

(Eval Rand Fail)
$$\frac{e_1 \Downarrow^S_D v \quad e_2 \Downarrow^S_D \mathit{fail}}{e_1\, e_2 \Downarrow^S_D \mathit{fail}}$$

(Eval Frame)
$$\frac{e \Downarrow^R_{D \cap R} o}{R[e] \Downarrow^S_D o}$$

(Eval Grant)
$$\frac{e \Downarrow^S_{D \cup (R \cap S)} o}{\mathit{grant}\ R\ \mathit{in}\ e \Downarrow^S_D o}$$

(Eval Test)
$$\frac{e_{R \subseteq D} \Downarrow^S_D o}{\mathit{test}\ R\ \mathit{then}\ e_{\mathrm{true}}\ \mathit{else}\ e_{\mathrm{false}} \Downarrow^S_D o}$$

Lemma 6 *$o \Downarrow^S_D o'$ if and only if $o = o'$ and $D \subseteq S$.*

Proof By (Eval Outcome), $o = o'$ and $D \subseteq S$ imply $o \Downarrow^S_D o'$. On the other hand, $o \Downarrow^S_D o'$ only holds when $D \subseteq S$, and the only rule to derive it is (Eval Outcome), and so $o \Downarrow^S_D o'$ implies $o = o'$. □

Lemma 7 *If $e \rightarrow^S_D e'$ and $e' \Downarrow^S_D o$ then $e \Downarrow^S_D o$.*

Proof The proof is by induction on the derivation of $e \rightarrow^S_D e'$. There is a case for each of the rules of the operational semantics.

(Ctx Rator) We have $e_1\, e_2 \rightarrow^S_D e'_1\, e_2$ derived from $e_1 \rightarrow^S_D e'_1$, and $e'_1\, e_2 \Downarrow^S_D o$. Only (Eval Appl) can derive the latter, so we have $e'_1 \Downarrow^S_D \lambda x.e_3$ and $e_2 \Downarrow^S_D u$ and $e_3\{x \leftarrow u\} \Downarrow^S_D o$. By induction hypothesis, $e_1 \rightarrow^S_D e'_1$ and $e'_1 \Downarrow^S_D \lambda x.e_3$ imply $e_1 \Downarrow^S_D \lambda x.e_3$. Hence, by (Eval Appl), we get $e_1\, e_2 \Downarrow^S_D o$.

(Ctx Rand) We have $v_1\, e_2 \rightarrow^S_D v_1\, e'_2$ derived from $e_2 \rightarrow^S_D e'_2$, and $v_1\, e'_2 \Downarrow^S_D o$. Only (Eval Appl) can derive the latter, so we have $v_1 \Downarrow^S_D \lambda x.e_3$ and $e'_2 \Downarrow^S_D u$ and $e_3\{x \leftarrow u\} \Downarrow^S_D o$. By induction hypothesis, $e_2 \rightarrow^S_D e'_2$ and $e'_2 \Downarrow^S_D u$ imply $e_2 \Downarrow^S_D u$. Hence, by (Eval Appl), we get $v_1\, e_2 \Downarrow^S_D o$.

(Red Appl) We have $(\lambda x.e)\, u \rightarrow^S_D e\{x \leftarrow u\}$ and $e\{x \leftarrow u\} \Downarrow^S_D o$. By (Eval Outcome), we have $\lambda x.e \Downarrow^S_D \lambda x.e$ and $u \Downarrow^S_D u$. By (Eval Appl), these facts and $e\{x \leftarrow u\} \Downarrow^S_D o$ imply $(\lambda x.e)\, u \Downarrow^S_D o$.

(Fail Rator) We have $\mathit{fail}\,e \to^S_D \mathit{fail}$ and $\mathit{fail} \Downarrow^S_D o$. By (Eval Outcome), we have $\mathit{fail} \Downarrow^S_D \mathit{fail}$, and so by Lemma 6, $o = \mathit{fail}$. By (Eval Rator Fail), $\mathit{fail} \Downarrow^S_D \mathit{fail}$ implies $\mathit{fail}\,e \Downarrow^S_D \mathit{fail}$, that is, $\mathit{fail}\,e \Downarrow^S_D o$.

(Fail Rand) We have $v\,\mathit{fail} \to^S_D \mathit{fail}$ and $\mathit{fail} \Downarrow^S_D o$. By (Eval Outcome), we have $v \Downarrow^S_D v$ and $\mathit{fail} \Downarrow^S_D \mathit{fail}$, and so by Lemma 6, $o = \mathit{fail}$. By (Eval Rand Fail), $v \Downarrow^S_D v$ and $\mathit{fail} \Downarrow^S_D \mathit{fail}$ imply $v\,\mathit{fail} \Downarrow^S_D o$.

(Ctx Frame) We have $R[e] \to^S_D R[e']$ derived from $e \to^R_{D\cap R} e'$, and the judgment $R[e'] \Downarrow^S_D o$. Only (Eval Frame) can derive the latter, so we have $e' \Downarrow^R_{D\cap R} o$. By induction hypothesis, $e \to^R_{D\cap R} e'$ and $e' \Downarrow^R_{D\cap R} o$ imply $e \Downarrow^R_{D\cap R} o$. By (Eval Frame), $R[e] \Downarrow^S_D o$.

(Ctx Grant) We have $\mathit{grant}\ R\ \mathit{in}\ e \to^S_D \mathit{grant}\ R\ \mathit{in}\ e'$ derived from the judgment $e \to^S_{D\cup(R\cap S)} e'$, and $\mathit{grant}\ R\ \mathit{in}\ e' \Downarrow^S_D o$. Only (Eval Grant) can derive the latter, so we have $e' \Downarrow^S_{D\cup(R\cap S)} o$. By induction hypothesis, $e \to^S_{D\cup(R\cap S)} e'$ and $e' \Downarrow^S_{D\cup(R\cap S)} o$ imply $e \Downarrow^S_{D\cup(R\cap S)} o$. By (Eval Grant), $\mathit{grant}\ R\ \mathit{in}\ e \Downarrow^S_D o$.

(Red Frame) We have $R[o'] \to^S_D o'$ and $o' \Downarrow^S_D o$. By Lemma 6, $o' = o$. By (Eval Frame), $R[o'] \Downarrow^S_D o$.

(Red Grant) We have $\mathit{grant}\ R\ \mathit{in}\ o' \to^S_D o'$ and $o' \Downarrow^S_D o$. By Lemma 6, $o' = o$. By (Eval Grant), $\mathit{grant}\ R\ \mathit{in}\ o' \Downarrow^S_D o$.

(Red Test) We have $\mathit{test}\ R\ \mathit{then}\ e_{\text{true}}\ \mathit{else}\ e_{\text{false}} \to^S_D e_{R\subseteq D}$ and $e_{R\subseteq D} \Downarrow^S_D o$. By (Eval Test), $\mathit{test}\ R\ \mathit{then}\ e_{\text{true}}\ \mathit{else}\ e_{\text{false}} \Downarrow^S_D o$. □

We can now show that the inductive definition of this section coincides with the definition from Section 4.3 in terms of the reduction semantics.

Proposition 4 *For all e, o, S, D, we have $e \Downarrow^S_D o$ if and only if $e\ (\to^S_D)^*\ o$ and $D \subseteq S$.*

Proof We can show that $e \Downarrow^S_D o$ implies $e\ (\to^S_D)^*\ o$ and $D \subseteq S$ by an easy induction on the derivation of $e \Downarrow^S_D o$. On the other hand, suppose $D \subseteq S$ and $e\ (\to^S_D)^*\ o$, that is, that $e \to^S_D e_2 \to^S_D \cdots \to^S_D e_n \to^S_D o$ for some expressions $e_2, \ldots, e_n$. By (Eval Outcome), $o \Downarrow^S_D o$. Hence, by applying Lemma 7 repeatedly, we get $e_n \Downarrow^S_D o, \ldots, e_2 \Downarrow^S_D o$, and $e \Downarrow^S_D o$. □

C Additional Proofs

C.1 Proof of Proposition 1

Restatement of Proposition 1 *The equations in the table of derived equations in Section 4.1 are indeed derivable.*

Proof There is a case for each equation in the table.

(Eq Refl) $e \equiv e$

The proof is by induction on the size of the expression e, using (Eq x) and (Eq Fail) in the base case, and using the other congruence rules in the inductive cases.

(Let Beta) $let\ x = v\ in\ e \equiv e\{x \leftarrow v\}$

$$\begin{array}{rcl} let\ x = v\ in\ e & = & (\lambda x.e_2)\, v \\ & \equiv & e_2\{x \leftarrow v\} \text{ by (Fun Beta)} \end{array}$$

(Frame Dup) $R[R[e]] \equiv R[e]$

This is an instance of (Frame Frame) with $R = R_1 = R_2$.

(Frame Appl) $R[e_1\, e_2] \equiv R[R[e_1]\, R[e_2]]$

$$\begin{array}{ll} R[e_1\, e_2] & \\ \quad \equiv & R[R[e_1\, e_2]] \text{ by (Frame Dup)} \\ \quad \equiv & R[R[(R[R[e_1]])\, (R[R[e_2]])]] \text{ by (Frame Frame Appl)} \\ \quad \equiv & R[R[e_1]\, R[e_2]] \text{ by (Frame Dup)} \end{array}$$

(Frame Frame $\cap$) $R_1[R_2[e]] \equiv (R_1 \cap R_2)[R_2[e]]$

$$\begin{array}{ll} R_1[R_2[e]] & \\ \quad \equiv & R_1[R_2[R_2[e]]] \text{ by (Frame Dup)} \\ \quad \equiv & (R_1 \cap R_2)[R_2[e]] \text{ by (Frame Frame Frame)} \end{array}$$

(Frame Grant $\cap$) $R_1[grant\ R_2\ in\ e] \equiv R_1[grant\ R_1 \cap R_2\ in\ R_1[e]]$

$$\begin{array}{ll} R_1[grant\ R_2\ in\ e] & \\ \quad \equiv & R_1[grant\ R_1 \cap R_2\ in\ e] \text{ by (Frame Grant)} \\ \quad \equiv & R_1[R_1[grant\ R_1 \cap R_2\ in\ e]] \text{ by (Frame Dup)} \\ \quad \equiv & R_1[grant\ R_1 \cap R_2\ in\ R_1[e]] \\ & \quad \text{by (Frame Grant Frame), since } R_1 \supseteq R_1 \cap R_2 \end{array}$$

(Frame Grant $\cap\,\varnothing$) $R_1 \cap R_2 = \varnothing \implies R_1[\mathit{grant}\ R_2\ \mathit{in}\ e] \equiv R_1[e]$

$$\begin{aligned}
&R_1[\mathit{grant}\ R_2\ \mathit{in}\ e]\\
&\quad\equiv\quad R_1[\mathit{grant}\ R_1{\cap}R_2\ \mathit{in}\ e] \text{ by (Frame Grant)}\\
&\quad=\quad R_1[\mathit{grant}\ \varnothing\ \mathit{in}\ e] \text{ since } R_1 \cap R_2 = \varnothing\\
&\quad\equiv\quad R_1[e] \text{ by (Grant } \varnothing\text{)}
\end{aligned}$$

(Frame Grant Frame $\cap$) $R_1[\mathit{grant}\ R_2\ \mathit{in}\ R_3[e]] \equiv R_1[R_3[\mathit{grant}\ R_1{\cap}R_2\ \mathit{in}\ e]]$

$$\begin{aligned}
&R_1[\mathit{grant}\ R_2\ \mathit{in}\ R_3[e]]\\
&\quad\equiv\quad R_1[\mathit{grant}\ R_1{\cap}R_2\ \mathit{in}\ R_3[e]] \text{ by (Frame Grant)}\\
&\quad\equiv\quad R_1[R_3[\mathit{grant}\ R_1{\cap}R_2\ \mathit{in}\ e]]\\
&\qquad\quad \text{by (Frame Grant Frame) since } R_1 \supseteq R_1 \cap R_2
\end{aligned}$$

(Frame Frame Test Else) $\neg(R_1 \supseteq R_3) \implies$
$$R_1[R_2[\mathit{test}\ R_3\ \mathit{then}\ e_1\ \mathit{else}\ e_2]] \equiv R_1[R_2[e_2]]$$

We argue by cases. First, suppose $R_2 \supseteq R_3$.

$$\begin{aligned}
&R_1[R_2[\mathit{test}\ R_3\ \mathit{then}\ e_1\ \mathit{else}\ e_2]]\\
&\quad\equiv\quad R_1[\mathit{test}\ R_3\ \mathit{then}\ R_2[e_1]\ \mathit{else}\ R_2[e_2]]\\
&\qquad\quad \text{by (Frame Test Then) since } R_2 \supseteq R_3\\
&\quad\equiv\quad R_1[R_2[e_2]] \text{ by (Frame Test Else) since } \neg(R_1 \supseteq R_3)
\end{aligned}$$

Second, suppose $\neg(R_2 \supseteq R_3)$.

$$\begin{aligned}
&R_1[R_2[\mathit{test}\ R_3\ \mathit{then}\ e_1\ \mathit{else}\ e_2]]\\
&\quad\equiv\quad R_1[R_2[e_2]] \text{ by (Frame Test Else) since } \neg(R_2 \supseteq R_3)
\end{aligned}$$

This completes the case analysis. □

C.2 Proof of Theorem 2

Theorem 2 asserts that the equational theory is complete in the sense that given a reduction $e \rightarrow^S_D e'$, then the equation $\mathcal{C}^S_D(e) \equiv \mathcal{C}^S_D(e')$ is derivable (with $\mathcal{C}^S_D((\cdot)) \triangleq D[\mathit{grant}\ D\ \mathit{in}\ S[(\cdot)]]$).

This section presents the proof. Beforehand, we prove three auxiliary lemmas, concerning the distribution of security-setter contexts over applications, frames and grants, respectively.

Lemma 8 *Assuming $D \subseteq S$, we have $\mathcal{C}^S_D(e_1\, e_2) \equiv \mathcal{C}^S_D(\mathcal{C}^S_D(e_1)\, \mathcal{C}^S_D(e_2))$.*

Proof

$\mathcal{C}^S_D(e_1\, e_2)$

$=$ $D[\mathit{grant}\ D\ \mathit{in}\ S[e_1\, e_2]]$ by definition

$\equiv$ $D[S[\mathit{grant}\ D\ \mathit{in}\ (e_1\, e_2)]]$ by (Frame Grant Frame)

$\equiv$ $D[S[\mathit{grant}\ D\ \mathit{in}\ ((\mathit{grant}\ D\ \mathit{in}\ e_1)\ \mathit{grant}\ D\ \mathit{in}\ e_2)]]$
by (Grant Appl)

$\equiv$ $D[\mathit{grant}\ D\ \mathit{in}\ S[(\mathit{grant}\ D\ \mathit{in}\ e_1)\ \mathit{grant}\ D\ \mathit{in}\ e_2]]$
by (Frame Grant Frame) since $D \subseteq D$

$\equiv$ $D[\mathit{grant}\ D\ \mathit{in}\ D[S[(\mathit{grant}\ D\ \mathit{in}\ e_1)\ \mathit{grant}\ D\ \mathit{in}\ e_2]]]$
by (Frame Grant $\cap$) since $D = D \cap D$

$\equiv$ $D[\mathit{grant}\ D\ \mathit{in}\ D[S[D[S[\mathit{grant}\ D\ \mathit{in}\ e_1]]\ D[S[\mathit{grant}\ D\ \mathit{in}\ e_2]]]]]$
by (Frame Frame Appl)

$\equiv$ $D[\mathit{grant}\ D\ \mathit{in}\ D[S[\mathcal{C}^S_D(e_1)\, \mathcal{C}^S_D(e_2)]]]$
by (Frame Grant Frame) and by definition

$\equiv$ $D[\mathit{grant}\ D\ \mathit{in}\ S[\mathcal{C}^S_D(e_1)\, \mathcal{C}^S_D(e_2)]]$ by (Frame Grant $\cap$)

$=$ $\mathcal{C}^S_D(\mathcal{C}^S_D(e_1)\, \mathcal{C}^S_D(e_2))$ by definition

□

Lemma 9 *Assuming $D \subseteq S$, we have $\mathcal{C}^S_D(R[e]) \equiv \mathcal{C}^R_{D\cap R}(e)$.*

Proof

$D[\mathit{grant}\ D\ \mathit{in}\ S[R[e]]]$

$\equiv$ $D[S[\mathit{grant}\ D\ \mathit{in}\ R[e]]]$ by (Frame Grant Frame) since $D \subseteq D$

$\equiv$ $D[S[R[\mathit{grant}\ D\ \mathit{in}\ e]]]$ by (Frame Grant Frame) since $D \subseteq S$

$\equiv$ $(D \cap S \cap R)[R[\mathit{grant}\ D \cap R\ \mathit{in}\ e]]$
by (Frame Frame Frame), (Frame Frame $\cap$), (Frame Grant)

$\equiv$ $(D \cap R)[\mathit{grant}\ D \cap R\ \mathit{in}\ R[e]]$
since $D \cap S \cap R = D \cap R$ and by (Frame Grant Frame)

□

Lemma 10 *Assuming $D \subseteq S$, we have $\mathcal{C}^S_D(\mathit{grant}\ R\ \mathit{in}\ e) \equiv \mathcal{C}^S_{D\cup(R\cap S)}(e)$.*

Proof

$$\begin{array}{ll} & D[\mathit{grant}\ D\ \mathit{in}\ S[\mathit{grant}\ R\ \mathit{in}\ e]] \\ \equiv & D[S[\mathit{grant}\ D\ \mathit{in}\ \mathit{grant}\ R\ \mathit{in}\ e]] \\ & \text{by (Frame Grant Frame) since } D \subseteq D \\ \equiv & D[S[\mathit{grant}\ D{\cup}(R{\cap}S)\ \mathit{in}\ e]] \\ & \text{by (Grant Grant) and (Frame Grant) since } D \subseteq S \\ \equiv & D{\cup}(R{\cap}S)[S[\mathit{grant}\ D{\cup}(R{\cap}S)\ \mathit{in}\ e]] \text{ by (Frame Frame Grant)} \\ \equiv & D{\cup}(R{\cap}S)[\mathit{grant}\ D{\cup}(R{\cap}S)\ \mathit{in}\ S[e]] \text{ by (Frame Grant Frame)} \end{array}$$

□

Restatement of Theorem 2 *If* $e \rightarrow^S_D e'$ *then* $\mathcal{C}^S_D(e) \equiv \mathcal{C}^S_D(e')$.

Proof The proof is by induction on the derivation of $e \rightarrow^S_D e'$. By definition, $e \rightarrow^S_D e'$ implies that $D \subseteq S$.

(Ctx Rator) We have $e_1\, e_2 \rightarrow^S_D e'_1\, e_2$ derived from $e_1 \rightarrow^S_D e'_1$.

$$\begin{array}{ll} & \mathcal{C}^S_D(e_1\, e_2) \\ \equiv & \mathcal{C}^S_D(\mathcal{C}^S_D(e_1)\, \mathcal{C}^S_D(e_2)) \text{ by Lemma 8 given } D \subseteq S \\ \equiv & \mathcal{C}^S_D(\mathcal{C}^S_D(e'_1)\, \mathcal{C}^S_D(e_2)) \\ & \text{since } \mathcal{C}^S_D(e_1) \equiv \mathcal{C}^S_D(e'_1) \text{ by induction hypothesis} \\ \equiv & \mathcal{C}^S_D(e'_1\, e_2) \text{ by Lemma 8 given } D \subseteq S \end{array}$$

(Ctx Rand) We have $v_1\, e_2 \rightarrow^S_D v_1\, e'_2$ derived from $e_2 \rightarrow^S_D e'_2$.

$$\begin{array}{ll} & \mathcal{C}^S_D(v_1\, e_2) \\ \equiv & \mathcal{C}^S_D(\mathcal{C}^S_D(v_1)\, \mathcal{C}^S_D(e_2)) \text{ by Lemma 8 given } D \subseteq S \\ \equiv & \mathcal{C}^S_D(\mathcal{C}^S_D(v_1)\, \mathcal{C}^S_D(e'_2)) \\ & \text{since } \mathcal{C}^S_D(e_2) \equiv \mathcal{C}^S_D(e'_2) \text{ by induction hypothesis} \\ \equiv & \mathcal{C}^S_D(v_1\, e'_2) \text{ by Lemma 8 given } D \subseteq S \end{array}$$

(Red Appl) We have $(\lambda x.e)\, v \rightarrow^S_D e\{x{\leftarrow}v\}$. Then

$$D[\mathit{grant}\ D\ \mathit{in}\ S[(\lambda x.e)\, v]] \equiv D[\mathit{grant}\ D\ \mathit{in}\ S[e\{x{\leftarrow}v\}]]$$

follows from (Fun Beta), (Eq Frame), and (Eq Grant).

(Fail Rator) We have $\mathit{fail}\ e \rightarrow^S_D \mathit{fail}$. Then

$$D[\mathit{grant}\ D\ \mathit{in}\ S[\mathit{fail}\ e]] \equiv D[\mathit{grant}\ D\ \mathit{in}\ S[\mathit{fail}]]$$

follows from (Eq Fail Rator), (Eq Frame), and (Eq Grant).

(Fail Rand) We have $v\ \mathit{fail} \rightarrow^S_D \mathit{fail}$. Then

$$D[\mathit{grant}\ D\ \mathit{in}\ S[v\ \mathit{fail}]] \equiv D[\mathit{grant}\ D\ \mathit{in}\ S[\mathit{fail}]]$$

follows from (Eq Fail Rand), (Eq Frame), and (Eq Grant).

(Ctx Frame) We have $R[e] \rightarrow^S_D R[e']$ derived from $e \rightarrow^R_{D \cap R} e'$. By induction hypothesis and Lemma 9.

(Ctx Grant) We have $\mathit{grant}\ R\ \mathit{in}\ e \rightarrow^S_D \mathit{grant}\ R\ \mathit{in}\ e'$ derived from $e \rightarrow^S_{D \cup (R \cap S)} e'$. By induction hypothesis and Lemma 10.

(Red Frame) We have $R[o] \rightarrow^S_D o$. Then

$$D[\mathit{grant}\ D\ \mathit{in}\ S[R[o]]] \equiv D[\mathit{grant}\ D\ \mathit{in}\ S[o]]$$

follows from (Frame o), (Eq Frame), and (Eq Grant).

(Red Grant) We have $\mathit{grant}\ R\ \mathit{in}\ o \rightarrow^S_D o$. Then

$$D[\mathit{grant}\ D\ \mathit{in}\ S[\mathit{grant}\ R\ \mathit{in}\ o]] \equiv D[\mathit{grant}\ D\ \mathit{in}\ S[o]]$$

follows from (Grant o), (Eq Frame), and (Eq Grant).

(Red Test) We have $\mathit{test}\ R\ \mathit{then}\ e_{\text{true}}\ \mathit{else}\ e_{\text{false}} \rightarrow^S_D e_{R \subseteq D}$. First, suppose that $R \subseteq D$.

$$\begin{aligned}
& D[\mathit{grant}\ D\ \mathit{in}\ S[\mathit{test}\ R\ \mathit{then}\ e_{\text{true}}\ \mathit{else}\ e_{\text{false}}]] \\
& \quad \equiv \quad D[S[\mathit{grant}\ D\ \mathit{in}\ \mathit{test}\ R\ \mathit{then}\ e_{\text{true}}\ \mathit{else}\ e_{\text{false}}]] \\
& \qquad\quad \text{by (Frame Grant Frame) given } D \subseteq S \\
& \quad \equiv \quad D[S[\mathit{grant}\ D\ \mathit{in}\ e_{\text{true}}]] \\
& \qquad\quad \text{by (Frame Grant Test) given } S \supseteq D \supseteq R \\
& \quad \equiv \quad D[\mathit{grant}\ D\ \mathit{in}\ S[e_{\text{true}}]] \\
& \qquad\quad \text{by (Frame Grant Frame) given } D \subseteq S
\end{aligned}$$

Second, suppose that $\neg(R \subseteq D)$.

$$
\begin{aligned}
& D[\mathit{grant}\ D\ \mathit{in}\ S[\mathit{test}\ R\ \mathit{then}\ e_{\text{true}}\ \mathit{else}\ e_{\text{false}}]] \\
& \quad \equiv \ D[\mathit{grant}\ D\ \mathit{in}\ D[S[\mathit{test}\ R\ \mathit{then}\ e_{\text{true}}\ \mathit{else}\ e_{\text{false}}]]] \\
& \qquad \text{by (Frame Grant } \cap \text{) given } D \cap D = D \\
& \quad \equiv \ D[\mathit{grant}\ D\ \mathit{in}\ D[S[e_{\text{false}}]]] \\
& \qquad \text{by (Frame Frame Test Else) given } \neg(R \subseteq D) \\
& \quad \equiv \ D[\mathit{grant}\ D\ \mathit{in}\ S[e_{\text{false}}]] \\
& \qquad \text{by (Frame Grant } \cap \text{) given } D \cap D = D
\end{aligned}
$$

□

C.3 Proof of Theorem 3

Theorem 3 states that bisimilarity is a congruence, that is, it is reflexive, symmetric, transitive, and preserved by all contexts (by which we mean that $e \sim^\circ e'$ implies $\mathcal{C}(e) \sim^\circ \mathcal{C}(e')$ for every context $\mathcal{C}$). For the purposes of proof, it is convenient to adopt an equivalent definition of congruence.

If $\mathcal{S}$ is a relation on expressions, let its *compatible refinement*, $\widehat{\mathcal{S}}$, be the least relation on expressions defined by the rules in the following table.

Compatible refinement

(Comp x)
$$x\ \widehat{\mathcal{S}}\ x$$

(Comp Fun)
$$\frac{e\ \mathcal{S}\ e'}{\lambda x.e\ \widehat{\mathcal{S}}\ \lambda x.e'}$$

(Comp Appl)
$$\frac{e\ \mathcal{S}\ e' \quad f\ \mathcal{S}\ f'}{e\ f\ \widehat{\mathcal{S}}\ e'\ f'}$$

(Comp Frame)
$$\frac{e\ \mathcal{S}\ e'}{R[e]\ \widehat{\mathcal{S}}\ R[e']}$$

(Comp Grant)
$$\frac{e\ \mathcal{S}\ e'}{\mathit{grant}\ R\ \mathit{in}\ e\ \widehat{\mathcal{S}}\ \mathit{grant}\ R\ \mathit{in}\ e'}$$

(Comp Test)
$$\frac{e\ \mathcal{S}\ e' \quad f\ \mathcal{S}\ f'}{\mathit{test}\ R\ \mathit{then}\ e\ \mathit{else}\ f\ \widehat{\mathcal{S}}\ \mathit{test}\ R\ \mathit{then}\ e'\ \mathit{else}\ f'}$$

(Comp Fail)
$$\mathit{fail}\ \widehat{\mathcal{S}}\ \mathit{fail}$$

Let a relation $\mathcal{S}$ be *compatible* if and only if $\widehat{\mathcal{S}} \subseteq \mathcal{S}$. A compatible relation is one that enjoys a congruence rule for each expression constructor in the syntax. Hence, we can easily show that a relation is a congruence if and only if it is reflexive, symmetric, transitive, and compatible. We use this alternative definition in the following proof that bisimilarity is a congruence.

We define similarity, the asymmetric form of bisimilarity.

- Let *ground applicative similarity*, $\lesssim$, be the greatest applicative simulation, that is, the union of all applicative simulations.

- Let *(applicative) similarity*, $\lesssim^\circ$, be such that $e \lesssim^\circ e'$ if and only if $e\sigma \sim e'\sigma$ for all substitutions σ such that $\sigma = \{x_1 \leftarrow v_1\} \cdots \{x_n \leftarrow v_n\}$ for some closed $v_1, \ldots, v_n$ where $\{x_1, \ldots, x_n\} = \mathit{fv}(e\, e')$.

Lemma 11 *Similarity is reflexive and transitive.*

Proof By standard arguments; see Milner [22], for instance. □

We proceed by Howe's method. We define a candidate relation, $e \lesssim^\bullet e'$, that is by definition compatible, and show it equals similarity. The congruence of bisimilarity itself then follows easily. The candidate relation is the least relation on open expressions to satisfy the following rule.

The candidate precongruence

(Cand)
$$\frac{e \;\widehat{\lesssim^\bullet}\; e' \quad e' \lesssim^\circ e''}{e \lesssim^\bullet e''}$$

Lemma 12 *The following rules are derivable.*

(Cand Refl)
$$e \lesssim^\bullet e$$

(Cand Comp)
$$\frac{e \;\widehat{\lesssim^\bullet}\; e'}{e \lesssim^\bullet e'}$$

(Cand Sim)
$$\frac{e \lesssim^\circ e'}{e \lesssim^\bullet e'}$$

(Cand Right)
$$\frac{e \lesssim^\bullet e' \quad e' \lesssim^\circ e''}{e \lesssim^\bullet e''}$$

(Cand Subst)
$$\frac{e \lesssim^\bullet e' \quad v \lesssim^\bullet v'}{e\{x \leftarrow v\} \lesssim^\bullet e'\{x \leftarrow v'\}}$$

Proof By standard arguments; see Gordon [12], for instance. □

Two auxiliary definitions are useful. We define $\lesssim^\bullet_c$ to be the candidate relation restricted to closed expressions. Moreover, if $\mathcal{S}$ is a relation on closed expressions, we let $\overline{\mathcal{S}}$ be the relation on closed expressions defined below.

- Let $e \lesssim^\bullet_c e'$ if and only if $e \lesssim^\bullet e'$ and the expressions e and e' are closed.

- Let $o_1 \;\overline{\mathcal{S}}\; o_2$ if and only if the outcomes o_1 and o_2 are closed and

  (1) if $o_1 = \mathit{fail}$ then $o_2 = \mathit{fail}$;

(2) if $o_1 = \lambda x.f_1$ then there is f_2 such that $o_2 = \lambda x.f_2$ and for every closed value v, $f_1\{x\leftarrow v\}\ \mathcal{S}\ f_2\{x\leftarrow v\}$.

Lemma 13 *If $o\ \overline{\lesssim^\bullet_c}\ o'$ and $o'\ \overline{\lesssim}\ o''$ then $o\ \overline{\lesssim^\bullet_c}\ o''$.*

Proof

- Suppose $o = \mathit{fail}$. Then $\mathit{fail}\ \overline{\lesssim^\bullet}\ o'$ implies $o' = \mathit{fail}$, and $\mathit{fail}\ \overline{\lesssim^\bullet}\ o''$ implies $o'' = \mathit{fail}$. So $o\ \overline{\lesssim^\bullet}\ o''$ follows.

- Suppose $o = \lambda x.e$. Then $\lambda x.e\ \overline{\lesssim^\bullet}\ o'$ implies there is e' such that $o' = \lambda x.e'$ and $e\{x\leftarrow v\} \lesssim^\bullet e'\{x\leftarrow v\}$ for every closed v. Then $\lambda x.e'\ \overline{\lesssim}\ o''$ implies there is e' such that $o' = \lambda x.e'$ and $e\{x\leftarrow v\} \lesssim e'\{x\leftarrow v\}$ for every closed v. By (Cand Right), $e\{x\leftarrow v\} \lesssim^\bullet e''\{x\leftarrow v\}$ for every closed v. Hence, we have $\lambda x.e\ \overline{\lesssim^\bullet}\ \lambda x.e''$, that is, $o\ \overline{\lesssim^\bullet}\ o''$. □

Given our new notation, a relation $\mathcal{S}$ on closed expressions is an applicative simulation if and only if $e\ \mathcal{S}\ e'$ and $e \Downarrow^S_D o$ implies there is o' such that $e' \Downarrow^S_D o'$ and $o\ \overline{\mathcal{S}}\ o'$. In particular, the next lemma asserts that $\lesssim^\bullet_c$ is an applicative simulation.

Lemma 14 *If $e \lesssim^\bullet_c e'$ and $e \Downarrow^S_D o$ then there is o' such that $e' \Downarrow^S_D o'$ and $o\ \overline{\lesssim^\bullet_c}\ o'$.*

Proof The proof is by induction on the derivation of $e \Downarrow^S_D o$. By definition, $e \Downarrow^S_D o$ implies that $D \subseteq S$.

(Eval Outcome)

$$\frac{}{o \Downarrow^S_D o}$$

- Suppose $o = \mathit{fail}$. Since only (Cand) and (Comp Fail) can have derived $\mathit{fail} \lesssim^\bullet_c e'$, it must be that $\mathit{fail}\ \widehat{\lesssim^\bullet_c}\ \mathit{fail}$ and $\mathit{fail} \lesssim e'$. From the latter, $e' \Downarrow^S_D \mathit{fail}$. By definition, $\mathit{fail}\ \overline{\lesssim^\bullet_c}\ \mathit{fail}$.

- Suppose $o = \lambda x.e_1$. Since only (Cand) and (Comp Fun) can have derived $\lambda x.e_1 \lesssim^\bullet_c e'$ there must be e_1' such that $e_1 \lesssim^\bullet e_1'$ and $\lambda x.e_1' \lesssim e'$. From the latter, there exists $o' = \lambda x.e_2'$ such that $e' \Downarrow^S_D o'$ and $\lambda x.e_1'\ \overline{\lesssim}\ o'$. By (Cand Subst), $e_1 \lesssim^\bullet e_1'$ implies $e_1\{x\leftarrow v\} \lesssim^\bullet e_1'\{x\leftarrow v\}$ for all closed v, and therefore, $\lambda x.e_1\ \overline{\lesssim^\bullet_c}\ \lambda x.e_1'$. By Lemma 13, $\lambda x.e_1\ \overline{\lesssim^\bullet_c}\ \lambda x.e_1'$ and $\lambda x.e_1'\ \overline{\lesssim}\ o'$ imply $\lambda x.e_1\ \overline{\lesssim^\bullet_c}\ o'$.

(Eval Appl)

$$\frac{e_1 \Downarrow^S_D \lambda x.e_3 \quad e_2 \Downarrow^S_D v \quad e_3\{x\leftarrow v\} \Downarrow^S_D o}{e_1\, e_2 \Downarrow^S_D o}$$

Since only (Cand) and (Comp Appl) can have derived $e_1\, e_2 \lesssim^\bullet_c e'$, there must be some e_1' and e_2' such that $e_1 \lesssim^\bullet_c e_1'$, $e_2 \lesssim^\bullet_c e_2'$, and $e_1'\, e_2' \lesssim e'$. By induction hypothesis, $e_1 \Downarrow^S_D \lambda x.e_3$ and $e_1 \lesssim^\bullet_c e_1'$ imply there is e_3' such that $e_1' \Downarrow^S_D \lambda x.e_3'$ and $\lambda x.e_3 \overline{\lesssim^\bullet_c} \lambda x.e_3'$. The latter implies that $e_3\{x\leftarrow v\} \lesssim^\bullet_c e_3'\{x\leftarrow v\}$. By induction hypothesis, $e_2 \Downarrow^S_D v$ and $e_2 \lesssim^\bullet_c e_2'$ imply there is v' such that $e_2' \Downarrow^S_D v'$ and $v \overline{\lesssim^\bullet_c} v'$. By induction hypothesis, $e_3\{x\leftarrow v\} \Downarrow^S_D o$ and $e_3\{x\leftarrow v\} \lesssim^\bullet_c e_3'\{x\leftarrow v\}$ imply there is o_{12} such that $e_3'\{x\leftarrow v\} \Downarrow^S_D o_{12}$ and $o \overline{\lesssim^\bullet_c} o_{12}$. By (Eval Appl), $e_1'\, e_2' \Downarrow^S_D o_{12}$. Given $e_1'\, e_2' \lesssim e'$, there must be o' such that $e' \Downarrow^S_D o'$ and $o_{12} \overline{\lesssim} o'$. By Lemma 13, $o \overline{\lesssim^\bullet_c} o_{12}$ and $o_{12} \overline{\lesssim} o'$ imply $o \overline{\lesssim^\bullet_c} o'$.

(Eval Rator Fail)

$$\frac{e_1 \Downarrow^S_D \mathit{fail}}{e_1\, e_2 \Downarrow^S_D \mathit{fail}}$$

Since only (Cand) and (Comp Appl) can have derived $e_1\, e_2 \lesssim^\bullet_c e'$, there must be some e_1' and e_2' such that $e_1 \lesssim^\bullet_c e_1'$, $e_2 \lesssim^\bullet_c e_2'$, and $e_1'\, e_2' \lesssim e'$. By induction hypothesis, $e_1 \Downarrow^S_D \mathit{fail}$ and $e_1 \lesssim^\bullet_c e_1'$ imply that $e_1' \Downarrow^S_D \mathit{fail}$. By (Eval Rator Fail), $e_1'\, e_2' \Downarrow^S_D \mathit{fail}$. Given $e_1'\, e_2' \lesssim e'$, it must be that $e' \Downarrow^S_D \mathit{fail}$.

(Eval Rand Fail)

$$\frac{e_1 \Downarrow^S_D v \quad e_2 \Downarrow^S_D \mathit{fail}}{e_1\, e_2 \Downarrow^S_D \mathit{fail}}$$

Since only (Cand) and (Comp Appl) can have derived $e_1\, e_2 \lesssim^\bullet_c e'$, there must be some e_1' and e_2' such that $e_1 \lesssim^\bullet_c e_1'$, $e_2 \lesssim^\bullet_c e_2'$, and $e_1'\, e_2' \lesssim e'$. By induction hypothesis, $e_1 \Downarrow^S_D v$ and $e_1 \lesssim^\bullet_c e_1'$ imply there is v' such that $e_1' \Downarrow^S_D v'$ and $v \overline{\lesssim^\bullet_c} v'$. By induction hypothesis, $e_2 \Downarrow^S_D \mathit{fail}$ and $e_2 \lesssim^\bullet_c e_2'$ imply that $e_2' \Downarrow^S_D \mathit{fail}$. By (Eval Rand Fail), $e_1'\, e_2' \Downarrow^S_D \mathit{fail}$. Given $e_1'\, e_2' \lesssim e'$, it must be that $e' \Downarrow^S_D \mathit{fail}$.

(Eval Frame)

$$\frac{e_1 \Downarrow^R_{D\cap R} o}{R[e_1] \Downarrow^S_D o}$$

Since only (Cand) and (Comp Frame) can have derived $R[e_1] \lesssim^\bullet_c e'$, there must be some e_1' such that $e_1 \lesssim^\bullet_c e_1'$ and $R[e_1'] \lesssim e'$. By induction hypothesis, $e_1 \Downarrow^R_{D\cap R} o$ and $e_1 \lesssim^\bullet_c e_1'$ imply there is o_1 such that $e_1' \Downarrow^R_{D\cap R} o_1$ and $o \overline{\lesssim^\bullet_c} o_1$. By (Eval Frame), $R[e_1'] \Downarrow^S_D o_1$. Given $R[e_1'] \lesssim e'$, there must be o' such that $e' \Downarrow^S_D o'$ and $o_1 \overline{\lesssim} o'$. By Lemma 13, $o \overline{\lesssim^\bullet_c} o_1$ and $o_1 \overline{\lesssim} o'$ imply $o \overline{\lesssim^\bullet_c} o'$.

(Eval Grant)

$$\frac{e_1 \Downarrow^S_{D\cup(R\cap S)} o}{\textit{grant } R \textit{ in } e_1 \Downarrow^S_D o}$$

Since only (Cand) and (Comp Grant) can have derived $\textit{grant } R \textit{ in } e_1 \lesssim^\bullet_c e'$, there must be some e'_1 such that $e_1 \lesssim^\bullet_c e'_1$ and $\textit{grant } R \textit{ in } e'_1 \lesssim e'$. By induction hypothesis, $e_1 \Downarrow^S_{D\cup(R\cap S)} o$ and $e_1 \lesssim^\bullet_c e'_1$ imply there is o_1 such that $e'_1 \Downarrow^S_{D\cup(R\cap S)} o_1$ and $o \overline{\lesssim^\bullet_c} o_1$. By (Eval Grant), $\textit{grant } R \textit{ in } e'_1 \Downarrow^S_D o_1$. Given $\textit{grant } R \textit{ in } e'_1 \lesssim e'$, there must be o' such that $e' \Downarrow^S_D o'$ and $o_1 \overline{\lesssim} o'$. By Lemma 13, $o \overline{\lesssim^\bullet_c} o_1$ and $o_1 \overline{\lesssim} o'$ imply $o \overline{\lesssim^\bullet_c} o'$.

(Eval Test)

$$\frac{e_{R\subseteq D} \Downarrow^S_D o}{\textit{test } R \textit{ then } e_{\text{true}} \textit{ else } e_{\text{false}} \Downarrow^S_D o}$$

Since only (Cand) and (Comp Test) can have derived

$$\textit{test } R \textit{ then } e_{\text{true}} \textit{ else } e_{\text{false}} \lesssim^\bullet_c e'$$

there must be some e'_{true} and e'_{false} such that $e_{\text{true}} \lesssim^\bullet_c e'_{\text{true}}$, $e_{\text{false}} \lesssim^\bullet_c e'_{\text{false}}$ and $\textit{test } R \textit{ then } e'_{\text{true}} \textit{ else } e'_{\text{false}} \lesssim e'$. By induction hypothesis, $e_{R\subseteq D} \lesssim^\bullet_c e'_{R\subseteq D}$ and $e_{R\subseteq D} \Downarrow^S_D o$ imply there is $o'_{R\subseteq D}$ such that $e'_{R\subseteq D} \Downarrow^S_D o'_{R\subseteq D}$ and $o \overline{\lesssim^\bullet_c} o'_{R\subseteq D}$. By (Eval Test), $\textit{test } R \textit{ then } e'_{\text{true}} \textit{ else } e'_{\text{false}} \Downarrow^S_D o'_{R\subseteq D}$. Given $\textit{test } R \textit{ then } e'_{\text{true}} \textit{ else } e'_{\text{false}} \lesssim e'$, there must be o' such that $e' \Downarrow^S_D o'$ and $o'_{R\subseteq D} \overline{\lesssim} o'$. By Lemma 13, $o \overline{\lesssim^\bullet_c} o'_{R\subseteq D}$ and $o'_{R\subseteq D} \overline{\lesssim} o'$ imply $o \overline{\lesssim^\bullet_c} o'$. □

Restatement of Theorem 3 *Bisimilarity is a congruence, that is, it is reflexive, symmetric, transitive, and preserved by all contexts.*

Proof By standard arguments, we have that bisimilarity, $\sim^\circ$, is reflexive, transitive, and symmetric, and that it is the symmetrisation of $\lesssim^\circ$, that is, $e \sim^\circ e'$ if and only if both $e \lesssim^\circ e'$ and $e' \lesssim^\circ e$.

Lemma 14 says that $\lesssim^\bullet_c$ is an applicative simulation, and therefore that $\lesssim^\bullet_c \subseteq \lesssim$. Using (Cand Subst) we can extend this inclusion to open expressions, that is, show that $\lesssim^\bullet \subseteq \lesssim^\circ$. (Cand Sim) is the reverse inclusion, that is, $\lesssim^\circ \subseteq \lesssim^\bullet$. Therefore, we have $\lesssim^\bullet = \lesssim^\circ$. By (Cand Comp), $\lesssim^\bullet$ is compatible, and therefore, so is $\lesssim^\circ$. Given that $\sim^\circ$ is the symmetrisation of $\lesssim^\circ$ it follows that $\sim^\circ$ is also compatible.

Finally, as we discussed at the beginning of this section, it is standard to show that a reflexive, transitive, and compatible relation is preserved by arbitrary contexts. □

C.4 Proof of Theorem 4

Theorem 4 asserts that the relations $\sim^\circ$ and $\simeq$ are in fact identical. One inclusion follows immediately from the fact that $\sim^\circ$ is a congruence.

Lemma 15 *If $e \sim^\circ e'$ then $e \simeq e'$.*

Proof Suppose that $e \sim^\circ e'$. Consider any context $\mathcal{C}$, such that both $\mathcal{C}(e)$ and $\mathcal{C}(e')$ are closed. By Theorem 3, $e \sim^\circ e'$ implies $\mathcal{C}(e) \sim^\circ \mathcal{C}(e')$. Since both $\mathcal{C}(e)$ and $\mathcal{C}(e')$ are closed, $\mathcal{C}(e) \sim \mathcal{C}(e')$. Now, $\sim$ is an applicative bisimulation, so $\mathcal{C}(e)\Downarrow \iff \mathcal{C}(e')\Downarrow$. It follows that $e \simeq e'$. □

The strategy for proving the other inclusion, that $e \simeq e'$ implies $e \sim^\circ e'$, is to show that the restriction of $\simeq$ to closed expressions is an applicative bisimulation. We need several auxiliary lemmas.

The next two lemmas state some properties of the operational semantics.

Lemma 16 *If $e \Downarrow_D^S o$ and $e \Downarrow_D^S o'$ then $o = o'$.*

Proof The proof is by induction on the derivation of $e \Downarrow_D^S o$. □

Lemma 17 *If $D \subseteq S$ and $V \subseteq U$ then $e \Downarrow_D^S o \iff \mathcal{C}_D^S(e) \Downarrow_V^U o$.*

Proof Noting that $D \cap S = D$ and $(V \cap D) \cup D = D$, we may calculate:

$$\begin{aligned}
e \Downarrow_D^S o &\iff S[e] \Downarrow_{(V\cap D)\cup D}^D o \\
&\iff \mathit{grant}\ D\ \mathit{in}\ S[e] \Downarrow_{(V\cap D)}^D o \\
&\iff D[\mathit{grant}\ D\ \mathit{in}\ S[e]] \Downarrow_V^U o
\end{aligned}$$

□

Using the previous two lemmas, we can show that the operational semantics preserves applicative bisimulation, in the following sense:

Lemma 18 *If e closed and $e \Downarrow_D^S o$ then $\mathcal{C}_D^S(e) \sim o$.*

Proof By definition, $e \Downarrow_D^S o$ implies $S \subseteq D$. We show the following relation is an applicative bisimulation, where *Id* is the identity relation on closed expressions.

$$\mathcal{S} \triangleq \{(\mathcal{C}_D^S(e), o) \mid e \Downarrow_D^S o\} \cup \mathit{Id}$$

To see that $\mathcal{S}$ is an applicative bisimulation, it suffices to check the following.

- Suppose $\mathcal{C}_D^S(e) \Downarrow_V^U o'$ and hence $U \subseteq V$. By Lemma 17, $e \Downarrow_D^S o'$. By Lemma 16, $o = o'$. By (Eval Outcome), $o \Downarrow_V^U o'$. We have $o' \overline{Id}\, o'$ and therefore $o' \overline{\mathcal{S}}\, o'$.

- Suppose $o \Downarrow_V^U o'$ and hence $U \subseteq V$. By (Eval Outcome), $o \Downarrow_V^U o$, and so, by Lemma 16, $o = o'$. By Lemma 17, $e \Downarrow_D^S o'$ implies $\mathcal{C}_D^S(e) \Downarrow_V^U o'$. Again, we have $o' \overline{\mathcal{S}}\, o'$. □

Next, we show that (Fun Beta) holds for bisimilarity.

Lemma 19 $(\lambda x.e)\, v \sim^\circ e\{x\leftarrow v\}$.

Proof Consider the following relation on closed expressions.

$$\mathcal{S} \triangleq \{((\lambda x.e)\, v, e\{x\leftarrow v\})\} \cup Id$$

To see that $\mathcal{S}$ is an applicative bisimulation, it suffices to check the following.

- Suppose $(\lambda x.e)\, v \Downarrow_D^S o$. Only (Eval Appl) can derive the latter, from $e\{x\leftarrow v\} \Downarrow_D^S o$. We have $o\, \overline{Id}\, o$ and therefore $o\, \overline{\mathcal{S}}\, o$.

- Suppose $e\{x\leftarrow v\} \Downarrow_D^S o$. By (Eval Appl), $(\lambda x.e)\, v \Downarrow_D^S o$. Again, we have $o\, \overline{\mathcal{S}}\, o$.

Now, consider any substitution σ, sending the free variables of $\lambda x.e$ and v to closed values. Since x is bound, we may assume x is not in the domain of σ. We have $((\lambda x.e)\, v)\sigma = (\lambda x.e\sigma)\, v\sigma\ \mathcal{S}\ e\sigma\{x\leftarrow v\sigma\} = (e\{x\leftarrow v\})\sigma$. Since $\mathcal{S}$ is an applicative bisimulation, the lemma follows. □

We now establish a substitution principle for contextual equivalence.

Lemma 20 *If $e \simeq e'$ then $e\{x\leftarrow v\} \simeq e'\{x\leftarrow v\}$.*

Proof From $e \simeq_c e'$ it follows that $(\lambda x.e)\, v \simeq (\lambda x.e')\, v$. By Lemma 15, Lemma 19 and transitivity of $\simeq$, we get $e\{x\leftarrow v\} \simeq e'\{x\leftarrow v\}$. □

We are now in a position to define the restriction of contextual equivalence to closed expressions, and show it to be an applicative simulation.

- Let $e \simeq_c e'$ if and only if $e \simeq e'$ and the expressions e and e' are closed.

Lemma 21 *If $e \simeq_c e'$ and $e \Downarrow_D^S o$ then there is o' such that $e' \Downarrow_D^S o'$ and $o\, \overline{\simeq_c}\, o'$.*

Proof By definition, $e \Downarrow_D^S o$ implies $S \subseteq D$. By Lemma 18, $e \Downarrow_D^S o$ implies $\mathcal{C}_D^S(e) \sim o$, and also, by Lemma 17, it implies $\mathcal{C}_D^S(e) \Downarrow_{\mathcal{P}}^{\mathcal{P}} o$, that is, $\mathcal{C}_D^S(e) \Downarrow o$. By definition, contextual equivalence is preserved by contexts, so $e \simeq_c e'$ implies $\mathcal{C}_D^S(e) \simeq_c \mathcal{C}_D^S(e')$. This and $\mathcal{C}_D^S(e)\Downarrow$ imply $\mathcal{C}_D^S(e')\Downarrow$, that is, there is o' such that $\mathcal{C}_D^S(e') \Downarrow o'$. By Lemma 17, this implies $e' \Downarrow_D^S o'$, and then Lemma 18 implies $\mathcal{C}_D^S(e') \sim o'$. We have $o \sim \mathcal{C}_D^S(e) \simeq_c \mathcal{C}_D^S(e') \sim o'$, so given Lemma 15, we get $o \simeq_c o'$.

Now, notice that for all closed values u, $\mathit{fail} \not\simeq_c u$. To see this, consider the context $\mathcal{C}(\cdot) \triangleq \mathit{let}\ x = \cdot\ \mathit{in}\ \Omega$. We have $\mathcal{C}(\mathit{fail}) \Downarrow$ but not $\mathcal{C}(u) \Downarrow$.

Therefore, $o \simeq_c o'$ implies that either $o = o' = \mathit{fail}$ or that there are f and f' such that $o = \lambda x.f$ and $o' = \lambda x.f'$. Consider any closed value v. From $\lambda x.f \simeq_c \lambda x.f'$ it follows that $(\lambda x.f)\ v \simeq_c (\lambda x.f')\ v$, and by Lemma 19 and Lemma 15 we get $f\{x{\leftarrow}v\} \simeq_c f'\{x{\leftarrow}v\}$. It follows then that $o \overline{\simeq_c} o'$. □

Restatement of Theorem 4 *The relations $\simeq$ and $\sim^\circ$ are equal.*

Proof According to Lemma 15, $\sim^\circ \subseteq \simeq$. On the other hand, suppose that $e \simeq e'$. Let $\{x_1, \ldots, x_n\} = \mathit{fv}(e\ e')$ and consider the substitution $\sigma = \{x_1{\leftarrow}v_1\} \cdots \{x{\leftarrow}v_n\}$ for any closed values $v_1, \ldots, v_n$. By Lemma 20, $e\sigma \simeq_c e\sigma$. According to Lemma 21, $\simeq_c$ is an applicative simulation. Since it is symmetric, it is also an applicative bisimulation, and therefore contained in ground applicative bisimilarity. Hence, $e\sigma \simeq_c e'\sigma$ implies $e\sigma \sim e'\sigma$. Since σ is an arbitrary closing substitution, it follows that $e \sim^\circ e'$. This shows that $\simeq \subseteq \sim^\circ$. □

C.5 Proof of Proposition 3

Restatement of Proposition 3 *For any expressions e_1 and e_2, $e_1 \sim^\circ e_2$ if for all D and S such that $D \subseteq S$, and for all substitutions σ sending variables to closed values with $\mathit{dom}(\sigma) = \mathit{fv}(e_1\ e_2)$ and for all o, $e_1\sigma \Downarrow_D^S o \Longleftrightarrow e_2\sigma \Downarrow_D^S o$.*

Proof Consider the relation $\mathcal{S}$ on closed expressions such that $e\ \mathcal{S}\ e'$ if and only if for all D and S such that $D \subseteq S$, and for all o, $e_1 \Downarrow_D^S o \Longleftrightarrow e_2 \Downarrow_D^S o$. Clearly, the relation $\mathcal{S} \cup \mathit{Id}$ is an applicative bisimulation, and therefore $\mathcal{S} \subseteq \sim$. Now, consider any expressions e_1 and e_2 as specified in the statement of the proposition. To show that $e_1 \sim^\circ e_2$ it suffices to establish $e_1\sigma \sim e_2\sigma$ for an arbitrary substitution $\sigma = \{x_1{\leftarrow}v_1\} \cdots \{x_n{\leftarrow}v_n\}$ for arbitrary closed $v_1, \ldots, v_n$ where $\{x_1, \ldots, x_n\} = \mathit{fv}(e_1\ e_2)$. By definition, $e_1\sigma\ \mathcal{S}\ e_2\sigma$, and therefore $e_1\sigma \sim e_2\sigma$ follows. □

C.6 Proof of Theorem 1

Lemma 22 *For all expressions let $x = e_1$ in e_2 and outcomes o, and all S and D, let $x = e_1$ in $e_2 \Downarrow_D^S o$ if and only if either:*

(1) *$e_1 \Downarrow_D^S fail$ and $o = fail$, or*

(2) *$e_1 \Downarrow_D^S v$ and $e_2\{x \leftarrow v\} \Downarrow_D^S o$ for some value v.*

Proof By inspecting the rules (Eval Appl), (Eval Rator Fail), and (Eval Rand Fail). □

Restatement of Theorem 1 *If $e \equiv e'$ then $e \simeq e'$.*

Proof The proof is by induction on the derivation of $e \equiv e'$. There is a case for each of the rules. The cases for the congruence rules rely on the fact that contextual equivalence is a congruence, that is, reflexive, symmetric, transitive, and compatible, which follows formally from Theorems 3 and 4.

(Eq Symm) $e' \equiv e \Longrightarrow e \equiv e'$

By induction hypothesis, $e' \equiv e$ implies $e' \simeq e$. Contextual equivalence is symmetric, so $e \simeq e'$.

(Eq Trans) $e \equiv e', e' \equiv e'' \Longrightarrow e \equiv e''$

By induction hypothesis, $e \equiv e'$ implies $e \simeq e'$, and $e' \equiv e''$ implies $e' \simeq e''$. Contextual equivalence is transitive, so $e \simeq e''$.

(Eq x) $x \equiv x$.

Contextual equivalence is reflexive, so $x \simeq x$.

(Eq Fun) $e \equiv e' \Longrightarrow \lambda x.e \equiv \lambda x.e'$

By induction hypothesis, $e \equiv e'$ implies $e \simeq e'$, and by (Comp Fun), $\lambda x.e \mathrel{\widehat{\simeq}} \lambda x.e'$. Since contextual equivalence is compatible, that is, $\widehat{\simeq} \subseteq \simeq$, we have $\lambda x.e \simeq \lambda x.e'$.

(Eq Appl), (Eq Frame), (Eq Grant), (Eq Test),(Eq Fail)
These cases follow easily by applying the induction hypothesis, and the facts that contextual equivalence is compatible and transitive, much as for (Eq Fun).

(Fun Beta) $(\lambda x.e)\, v \equiv e\{x \leftarrow v\}$

By Lemma 19, $(\lambda x.e)v \sim^\circ e\{x \leftarrow v\}$. By Theorem 4, $(\lambda x.e)v \simeq e\{x \leftarrow v\}$.

(Fun Eta) $x \notin fv(v) \Longrightarrow \lambda x.v\, x \equiv v$

Consider any substitution σ sending variables to closed values with $dom(\sigma) = fv(v)$. Since v is a value, that is, either a variable or a function, the expression $v\sigma$ must be a closed value, that is, a closed function. Therefore, $v\sigma = \lambda y.e$ for some closed function $\lambda y.e$. By Lemma 19 and Theorem 3, $\lambda x.(v\sigma\, x) \sim^\circ \lambda x.(e\{y{\leftarrow}x\})$. Given $x \notin dom(v)$, we have $(\lambda x.v\, x)\sigma = \lambda x.(v\sigma\, x)$ and $v\sigma = \lambda y.e = \lambda x.(e\{y{\leftarrow}x\})$. Since $\sim$ and $\sim^\circ$ agree on closed expressions, we have $(\lambda x.v\, x)\sigma \sim v\sigma$. By definition, $\lambda x.v\, x \sim^\circ v$. By Theorem 4, $\lambda x.v\, x \simeq v$.

(Let Eta) $let\ x = e\ in\ x \equiv e$

Consider any D and S such that $D \subseteq S$, and any substitution σ sending variables to closed values with $dom(\sigma) = fv(e)$, and any o. By (Eval Grant) and Lemma 22, we can see that the following two conditions are equivalent,

- $let\ x = e\sigma\ in\ x \Downarrow_D^S o$
- $e\sigma \Downarrow_D^S o$

since they are both equivalent to the following disjunction.

(1) $o = fail$ and $e\sigma \Downarrow_D^S fail$, or

(2) $e\sigma \Downarrow_D^S v$ for some v.

Hence, by Proposition 3 and Theorem 4, $let\ x = e\ in\ x \simeq e$.

(Let Let) $x_1 \notin fv(e_3) \Longrightarrow$

$$let\ x_1 = e_1\ in\ (let\ x_2 = e_2\ in\ e_3) \equiv$$
$$let\ x_2 = (let\ x_1 = e_1\ in\ e_2)\ in\ e_3$$

Consider any D and S such that $D \subseteq S$, and any substitution σ sending variables to closed values with $dom(\sigma) = fv(e_1) \cup (fv(e_2) - \{x_1\}) \cup (fv(e_3) - \{x_2\})$, and any o. By (Eval Grant) and Lemma 22, we can see that the following two conditions are equivalent,

- $let\ x_1 = e_1\sigma\ in\ (let\ x_2 = e_2\sigma\ in\ e_3\sigma) \Downarrow_D^S o$
- $let\ x_2 = (let\ x_1 = e_1\sigma\ in\ e_2\sigma)\ in\ e_3\sigma \Downarrow_D^S o$

since they are both equivalent to the following disjunction.

(1) $e_1\sigma \Downarrow_D^S fail$ and $o = fail$, or

(2) $e_1\sigma \Downarrow_D^S v$ and $let\ x_2 = e_2\sigma\{x{\leftarrow}v\}\ in\ e_3\sigma \Downarrow_D^S o$ for some value v.

Hence, by Proposition 3 and Theorem 4, $x_1 \notin fv(e_3) \implies let\ x_1 = e_1\ in\ (let\ x_2 = e_2\ in\ e_3) \simeq let\ x_2 = (let\ x_1 = e_1\ in\ e_2)\ in\ e_3$.

(Frame o) $R[o] \equiv o$

Consider any D and S such that $D \subseteq S$, and any substitution σ sending variables to closed values with $dom(\sigma) = fv(o)$, and any o'. Since the only evaluation rule applicable to $R[o\sigma]$ is (Eval Frame), we have $R[o\sigma] \Downarrow_D^S o' \iff o' = o\sigma \iff o\sigma \Downarrow_D^S o'$. By Proposition 3 and Theorem 4, $R[o] \simeq o$.

(Frame Frame Appl) $R_1[R_2[e_1\ e_2]] \equiv R_1[R_2[(R_1[R_2[e_1]])\ (R_1[R_2[e_2]])]]$

Consider any D and S such that $D \subseteq S$, and any substitution σ sending variables to closed values with $dom(\sigma) = fv(e_1\ e_2)$, and any o. By inspection of the evaluation rules, we can see that the following two conditions are equivalent,

- $R_1[R_2[e_1\sigma\ e_2\sigma]] \Downarrow_D^S o$
- $R_1[R_2[(R_1[R_2[e_1\sigma]])\ (R_1[R_2[e_2\sigma]])]] \Downarrow_D^S o$

since they are both equivalent to the following disjunction.

(1) $o = fail$ and $e_1\sigma \Downarrow_{D\cap R_1\cap R_2}^{R_1} fail$, or

(2) $o = fail$ and $e_1\sigma \Downarrow_{D\cap R_1\cap R_2}^{R_1} v$ and $e_2\sigma \Downarrow_{D\cap R_1\cap R_2}^{R_1} fail$ for some v, or

(3) $e_1\sigma \Downarrow_{D\cap R_1\cap R_2}^{R_1} \lambda x.e$ and $e_2\sigma \Downarrow_{D\cap R_1\cap R_2}^{R_1} v$ and $e\{x \leftarrow v\} \Downarrow_{D\cap R_1\cap R_2}^{R_1} o$ for some $\lambda x.e$ and v.

Hence, by Proposition 3 and Theorem 4,

$$R_1[R_2[e_1\ e_2]] \simeq R_1[R_2[(R_1[R_2[e_1]])\ (R_1[R_2[e_2]])]]$$

(Frame Let) $R[let\ x = e_1\ in\ e_2] \equiv let\ x = R[e_1]\ in\ R[e_2]$

Consider any D and S such that $D \subseteq S$, and any substitution σ sending variables to closed values with $dom(\sigma) = fv(let\ x = e_1\ in\ e_2)$, and any o. Since x is bound, we may assume $x \notin fv(e_2)$. By (Eval Frame) and Lemma 22, we can see that the following two conditions are equivalent,

- $R[let\ x = e_1\sigma\ in\ e_2\sigma] \Downarrow_D^S o$
- $let\ x = R[e_1\sigma]\ in\ R[e_2\sigma] \Downarrow_D^S o$

since they are both equivalent to the following disjunction.

(1) $o = \mathit{fail}$ and $e_2\sigma \Downarrow^R_{D\cap R} \mathit{fail}$, or

(2) $e_2\sigma \Downarrow^R_{D\cap R} v$ and $e_1\sigma\{x{\leftarrow}v\} \Downarrow^R_{D\cap R} o$ for some v.

Hence, by Proposition 3 and Theorem 4, $R[\mathit{let}\ x = e_1\ \mathit{in}\ e_2] \simeq \mathit{let}\ x = R[e_1]\ \mathit{in}\ R[e_2]$.

(Frame Frame) $R_1 \supseteq R_2 \Longrightarrow R_1[R_2[e]] \equiv R_2[e]$

Consider any D and S such that $D \subseteq S$, and any substitution σ sending variables to closed values with $\mathit{dom}(\sigma) = \mathit{fv}(e)$, and any o. Since $R_1 \supseteq R_2$, $(D \cap R_1) \cap R_2 = D \cap R_2$. Since (Eval Frame) is the only evaluation rule applicable to a framed expression, we have:

$$\begin{array}{rcl}
R_1[R_2[e\sigma]] \Downarrow^S_D o & \Longleftrightarrow & R_2[e\sigma] \Downarrow^{R_1}_{D\cap R_1} o \\
& \Longleftrightarrow & e\sigma \Downarrow^{R_2}_{(D\cap R_1)\cap R_2} o \\
& \Longleftrightarrow & e\sigma \Downarrow^{R_2}_{D\cap R_2} o \\
& \Longleftrightarrow & R_2[e\sigma] \Downarrow^S_D o
\end{array}$$

Hence, by Proposition 3 and Theorem 4, $R_1 \supseteq R_2 \Longrightarrow R_1[R_2[e]] \equiv R_2[e]$.

(Frame Frame Frame) $R_1[R_2[R_3[e]]] \equiv (R_1{\cap}R_2)[R_3[e]]$

Consider any D and S such that $D \subseteq S$, and any substitution σ sending variables to closed values with $\mathit{dom}(\sigma) = \mathit{fv}(e)$, and any o. Since (Eval Frame) is the only evaluation rule applicable to a framed expression, we have:

$$\begin{array}{rcl}
R_1[R_2[R_3[e\sigma]]] \Downarrow^S_D o & \Longleftrightarrow & R_2[R_3[e\sigma]] \Downarrow^{R_1}_{D\cap R_1} o \\
& \Longleftrightarrow & R_3[e\sigma] \Downarrow^{R_2}_{D\cap R_1\cap R_2} o \\
& \Longleftrightarrow & e\sigma \Downarrow^{R_3}_{D\cap R_1\cap R_2\cap R_3} o \\
& \Longleftrightarrow & R_3[e\sigma] \Downarrow^{R_1\cap R_2}_{D\cap R_1\cap R_2} o \\
& \Longleftrightarrow & (R_1 \cap R_2)[R_3[e\sigma]] \Downarrow^S_D o
\end{array}$$

Hence, by Proposition 3 and Theorem 4, $R_1[R_2[R_3[e]]] \simeq (R_1{\cap}R_2)[R_3[e]]$.

(Frame Frame Grant) $R_1[R_2[\mathit{grant}\ R_3\ \mathit{in}\ e]] \equiv (R_1{\cup}R_3)[R_2[\mathit{grant}\ R_3\ \mathit{in}\ e]]$

Consider any D and S such that $D \subseteq S$, and any substitution σ sending variables to closed values with $\mathit{dom}(\sigma) = \mathit{fv}(e)$, and any o. Note that:

$$\begin{array}{rl}
\multicolumn{2}{l}{(D \cap (R_1 \cup R_3) \cap R_2) \cup (R_3 \cap R_2)} \\
= & (D \cap R_1 \cap R_2) \cup (D \cap R_3 \cap R_2) \cup (R_3 \cap R_2) \\
= & (D \cap R_1 \cap R_2) \cup (R_3 \cap R_2)
\end{array}$$

Then, since (Eval Frame) and (Eval Grant) are the only evaluation rules applicable to a framed expression and a grant, respectively, we have:

$$\begin{aligned}
R_1[R_2[\mathit{grant}\ R_3\ \mathit{in}\ e\sigma]] \Downarrow_D^S o &\iff R_2[\mathit{grant}\ R_3\ \mathit{in}\ e\sigma] \Downarrow_{D\cap R_1}^{R_1} o\\
&\iff \mathit{grant}\ R_3\ \mathit{in}\ e\sigma \Downarrow_{D\cap R_1\cap R_2}^{R_2} o\\
&\iff e\sigma \Downarrow_{(D\cap R_1\cap R_2)\cup(R_3\cap R_2)}^{R_2} o\\
&\iff e\sigma \Downarrow_{(D\cap(R_1\cup R_3)\cap R_2)\cup(R_3\cap R_2)}^{R_2} o\\
&\iff \mathit{grant}\ R_3\ \mathit{in}\ e\sigma \Downarrow_{D\cap(R_1\cup R_3)\cap R_2}^{R_2} o\\
&\iff R_2[\mathit{grant}\ R_3\ \mathit{in}\ e\sigma] \Downarrow_{D\cap(R_1\cup R_3)}^{R_1\cup R_3} o\\
&\iff (R_1\cup R_3)[R_2[\mathit{grant}\ R_3\ \mathit{in}\ e\sigma]] \Downarrow_D^S o
\end{aligned}$$

Hence, by Proposition 3 and Theorem 4, $R_1[R_2[\mathit{grant}\ R_3\ \mathit{in}\ e]] \simeq (R_1\cup R_3)[R_2[\mathit{grant}\ R_3\ \mathit{in}\ e]]$.

(Frame Grant) $R_1[\mathit{grant}\ R_2\ \mathit{in}\ e] \equiv R_1[\mathit{grant}\ R_1\cap R_2\ \mathit{in}\ e]$

Consider any D and S such that $D \subseteq S$, and any substitution σ sending variables to closed values with $\mathit{dom}(\sigma) = \mathit{fv}(e)$, and any o. Since (Eval Frame) and (Eval Grant) are the only evaluation rules applicable to a framed expression and a grant, respectively, we have:

$$\begin{aligned}
R_1[\mathit{grant}\ R_2\ \mathit{in}\ e\sigma] \Downarrow_D^S o &\iff \mathit{grant}\ R_2\ \mathit{in}\ e\sigma \Downarrow_{D\cap R_1}^{R_1} o\\
&\iff e\sigma \Downarrow_{(D\cap R_1)\cup(R_2\cap R_1)}^{R_1} o\\
&\iff \mathit{grant}\ R_1\cap R_2\ \mathit{in}\ e\sigma \Downarrow_{D\cap R_1}^{R_1} o\\
&\iff R_1[\mathit{grant}\ R_1\cap R_2\ \mathit{in}\ e\sigma] \Downarrow_D^S o
\end{aligned}$$

Hence, by Proposition 3 and Theorem 4:

$$R_1[\mathit{grant}\ R_2\ \mathit{in}\ e] \simeq R_1[\mathit{grant}\ R_1\cap R_2\ \mathit{in}\ e]$$

(Frame Grant Frame) $R_1 \supseteq R_2 \implies$
$R_1[\mathit{grant}\ R_2\ \mathit{in}\ R_3[e]] \equiv R_1[R_3[\mathit{grant}\ R_2\ \mathit{in}\ e]]$

Consider any D and S such that $D \subseteq S$, and any substitution σ sending variables to closed values with $\mathit{dom}(\sigma) = \mathit{fv}(e)$, and any o. Given $R_1 \supseteq R_2$, we have $R_2 \cap R_1 = R_2$, and therefore:

$$((D \cap R_1) \cup (R_2 \cap R_1)) \cap R_3 \quad = \quad (D \cap R_1 \cap R_3) \cup (R_2 \cap R_3)$$

Hence, since (Eval Frame) and (Eval Grant) are the only evaluation rules applicable to a framed expression and a grant, respectively, we have:

$$
\begin{array}{rcl}
R_1[\mathit{grant}\ R_2\ \mathit{in}\ R_3[e\sigma]] \Downarrow^S_D o & \iff & \mathit{grant}\ R_2\ \mathit{in}\ R_3[e\sigma] \Downarrow^{R_1}_{D \cap R_1} o \\
& \iff & R_3[e\sigma] \Downarrow^{R_1}_{(D\cap R_1)\cup(R_2\cap R_1)} o \\
& \iff & e\sigma \Downarrow^{R_3}_{((D\cap R_1)\cup(R_2\cap R_1))\cap R_3} o \\
& \iff & e\sigma \Downarrow^{R_3}_{(D\cap R_1\cap R_3)\cup(R_2\cap R_3)} o \\
& \iff & \mathit{grant}\ R_2\ \mathit{in}\ e\sigma \Downarrow^{R_3}_{D\cap R_1\cap R_3} o \\
& \iff & R_3[\mathit{grant}\ R_2\ \mathit{in}\ e\sigma] \Downarrow^{R_1}_{D\cap R_1} o \\
& \iff & R_1[R_3[\mathit{grant}\ R_2\ \mathit{in}\ e\sigma]] \Downarrow^S_D o
\end{array}
$$

Hence, by Proposition 3 and Theorem 4:

$$R_1 \supseteq R_2 \Longrightarrow R_1[\mathit{grant}\ R_2\ \mathit{in}\ R_3[e]] \simeq R_1[R_3[\mathit{grant}\ R_2\ \mathit{in}\ e]]$$

(Frame Grant Test) $R_1 \supseteq R_2 \supseteq R_3 \Longrightarrow$
$R_1[\mathit{grant}\ R_2\ \mathit{in}\ \mathit{test}\ R_3\ \mathit{then}\ e_1\ \mathit{else}\ e_2] \equiv$
$R_1[\mathit{grant}\ R_2\ \mathit{in}\ e_1]$

Consider any D and S such that $D \subseteq S$, and any substitution σ sending variables to closed values with $\mathit{dom}(\sigma) = \mathit{fv}(e_1\ e_2)$, and any o. From $R_1 \supseteq R_2 \supseteq R_3$ it follows that:

$$R_3 \subseteq R_2 = R_2 \cap R_1 \subseteq (D \cap R_1) \cup (R_2 \cap R_1)$$

Hence, since (Eval Frame), (Eval Grant), and (Eval Test) are the only evaluation rules applicable to a framed expression, a grant, and a test, respectively, we have:

$$
\begin{array}{l}
R_1[\mathit{grant}\ R_2\ \mathit{in}\ \mathit{test}\ R_3\ \mathit{then}\ e_1\sigma\ \mathit{else}\ e_2\sigma] \Downarrow^S_D o \\
\quad \iff \mathit{grant}\ R_2\ \mathit{in}\ \mathit{test}\ R_3\ \mathit{then}\ e_1\sigma\ \mathit{else}\ e_2\sigma \Downarrow^{R_1}_{D\cap R_1} o \\
\quad \iff \mathit{test}\ R_3\ \mathit{then}\ e_1\sigma\ \mathit{else}\ e_2\sigma \Downarrow^{R_1}_{(D\cap R_1)\cup(R_2\cap R_1)} o \\
\quad \iff e_1\sigma \Downarrow^{R_1}_{(D\cap R_1)\cup(R_2\cap R_1)} o \\
\quad \iff \mathit{grant}\ R_2\ \mathit{in}\ e_1\sigma \Downarrow^{R_1}_{(D\cap R_1)} o \\
\quad \iff R_1[\mathit{grant}\ R_2\ \mathit{in}\ e_1\sigma] \Downarrow^S_D o
\end{array}
$$

Hence, by Proposition 3 and Theorem 4:

$$
\begin{array}{l}
R_1 \supseteq R_2 \supseteq R_3 \Longrightarrow \\
\quad R_1[\mathit{grant}\ R_2\ \mathit{in}\ \mathit{test}\ R_3\ \mathit{then}\ e_1\ \mathit{else}\ e_2] \simeq R_1[\mathit{grant}\ R_2\ \mathit{in}\ e_1]
\end{array}
$$

(Frame Test Then) $R_1 \supseteq R_2 \implies$
$$R_1[\mathit{test}\ R_2\ \mathit{then}\ e_1\ \mathit{else}\ e_2] \equiv \mathit{test}\ R_2\ \mathit{then}\ R_1[e_1]\ \mathit{else}\ R_1[e_2]$$

Consider any D and S such that $D \subseteq S$, and any substitution σ sending variables to closed values with $\mathit{dom}(\sigma) = \mathit{fv}(e_1\ e_2)$, and any o. From $R_1 \supseteq R_2$ it follows that $R_2 \subseteq D \cap R_1 \iff R_2 \subseteq D$. Hence, since (Eval Frame) and (Eval Test) are the only evaluation rules applicable to a framed expression and a test, respectively, we have:

$$\begin{aligned}
&R_1[\mathit{test}\ R_2\ \mathit{then}\ e_1\sigma\ \mathit{else}\ e_2\sigma] \Downarrow_D^S o \\
&\iff \mathit{test}\ R_2\ \mathit{then}\ e_1\sigma\ \mathit{else}\ e_2\sigma \Downarrow_{D\cap R_1}^{R_1} o \\
&\iff e_i\sigma \Downarrow_{D\cap R_1}^{R_1} o \quad \text{where } i = 1 \text{ if } R_2 \subseteq D \cap R_1, \text{ otherwise } i = 2 \\
&\iff R_1[e_i\sigma] \Downarrow_D^S o \quad \text{where } i = 1 \text{ if } R_2 \subseteq D, \text{ otherwise } i = 2 \\
&\iff \mathit{test}\ R_2\ \mathit{then}\ R_1[e_1\sigma]\ \mathit{else}\ R_1[e_2\sigma] \Downarrow_D^S o
\end{aligned}$$

Hence, by Proposition 3 and Theorem 4:

$$R_1 \supseteq R_2 \implies R_1[\mathit{test}\ R_2\ \mathit{then}\ e_1\ \mathit{else}\ e_2] \simeq \mathit{test}\ R_2\ \mathit{then}\ R_1[e_1]\ \mathit{else}\ R_1[e_2]$$

(Frame Test Else) $\neg(R_1 \supseteq R_2) \implies R_1[\mathit{test}\ R_2\ \mathit{then}\ e_1\ \mathit{else}\ e_2] \equiv R_1[e_2]$

Consider any D and S such that $D \subseteq S$, and any substitution σ sending variables to closed values with $\mathit{dom}(\sigma) = \mathit{fv}(e_1\ e_2)$, and any o. From $\neg(R_1 \supseteq R_2)$ it follows that $\neg(R_2 \subseteq D \cap R_1)$. Hence, since (Eval Frame) and (Eval Test) are the only evaluation rules applicable to a framed expression and a test, respectively, we have:

$$\begin{aligned}
&R_1[\mathit{test}\ R_2\ \mathit{then}\ e_1\sigma\ \mathit{else}\ e_2\sigma] \Downarrow_D^S o \\
&\iff \mathit{test}\ R_2\ \mathit{then}\ e_1\sigma\ \mathit{else}\ e_2\sigma \Downarrow_{D\cap R_1}^{R_1} o \\
&\iff e_1\sigma \Downarrow_{D\cap R_1}^{R_1} o \\
&\iff R_1[e_1\sigma] \Downarrow_D^S o
\end{aligned}$$

Hence, by Proposition 3 and Theorem 4:

$$\neg(R_1 \supseteq R_2) \implies R_1[\mathit{test}\ R_2\ \mathit{then}\ e_1\ \mathit{else}\ e_2] \simeq R_1[e_2]$$

(Grant $\varnothing$) $\mathit{grant}\ \varnothing\ \mathit{in}\ e \equiv e$

Consider any D and S such that $D \subseteq S$, and any substitution σ sending variables to closed values with $dom(\sigma) = fv(e_1\, e_2)$, and any o. Since (Eval Grant) is the only evaluation rule applicable to a grant, we have:

$$\begin{aligned} grant\ \varnothing\ in\ e\sigma \Downarrow_D^S o &\iff e\sigma \Downarrow_{D\cup(\varnothing\cap S)}^S o \\ &\iff e\sigma \Downarrow_D^S o \end{aligned}$$

Hence, by Proposition 3 and Theorem 4, $grant\ \varnothing\ in\ e \simeq e$.

(Grant o) $grant\ R\ in\ o \equiv o$

Consider any D and S such that $D \subseteq S$, and any substitution σ sending variables to closed values with $dom(\sigma) = fv(o)$, and any o'. Since (Eval Grant) is the only evaluation rule applicable to a grant, and by Lemma 6, we have:

$$\begin{aligned} grant\ R\ in\ o\sigma \Downarrow_D^S o' &\iff o\sigma \Downarrow_{D\cup(R\cap S)}^S o' \\ &\iff o\sigma = o' \\ &\iff o\sigma \Downarrow_D^S o' \end{aligned}$$

Hence, by Proposition 3 and Theorem 4, $grant\ R\ in\ o \simeq o$.

(Grant Appl) $grant\ R\ in\ (e_1\, e_2) \equiv$
$grant\ R\ in\ ((grant\ R\ in\ e_1)\ grant\ R\ in\ e_2)$

Consider any D and S such that $D \subseteq S$, and any substitution σ sending variables to closed values with $dom(\sigma) = fv(e_1\, e_2)$, and any o. By inspection of the evaluation rules, we can see that the following two conditions are equivalent,

- $grant\ R\ in\ (e_1\sigma\ e_2\sigma) \Downarrow_D^S o$
- $grant\ R\ in\ ((grant\ R\ in\ e_1\sigma)\ grant\ R\ in\ e_2\sigma) \Downarrow_D^S o$

since they are both equivalent to the following disjunction.

(1) $o = fail$ and $e_1\sigma \Downarrow_{D\cup(R\cap S)}^S fail$, or

(2) $o = fail$ and $e_1\sigma \Downarrow_{D\cup(R\cap S)}^S v$ and $e_2\sigma \Downarrow_{D\cup(R\cap S)}^S fail$ for some v, or

(3) $e_1\sigma \Downarrow_{D\cup(R\cap S)}^S \lambda x.e$ and $e_2\sigma \Downarrow_{D\cup(R\cap S)}^S v$ and $e\{x\leftarrow v\} \Downarrow_{D\cup(R\cap S)}^{R_1} o$ for some $\lambda x.e$ and v.

Hence, by Proposition 3 and Theorem 4,

$$\begin{aligned} &grant\ R\ in\ (e_1\, e_2) \equiv \\ &\quad grant\ R\ in\ ((grant\ R\ in\ e_1)\ grant\ R\ in\ e_2) \end{aligned}$$

(Grant Let) $\mathit{grant}\ R\ \mathit{in}\ (\mathit{let}\ x = e_1\ \mathit{in}\ e_2) \equiv$
$\mathit{let}\ x = (\mathit{grant}\ R\ \mathit{in}\ e_1)\ \mathit{in}\ (\mathit{grant}\ R\ \mathit{in}\ e_2)$

Consider any D and S such that $D \subseteq S$, and any substitution σ sending variables to closed values with $\mathit{dom}(\sigma) = \mathit{fv}(\mathit{let}\ x = e_1\ \mathit{in}\ e_2)$, and any o. Since x is bound, we may assume $x \notin \mathit{fv}(e_2)$. By (Eval Grant) and Lemma 22, we can see that the following two conditions are equivalent,

- $\mathit{grant}\ R\ \mathit{in}\ (\mathit{let}\ x = e_1\sigma\ \mathit{in}\ e_2\sigma) \Downarrow_D^S o$
- $\mathit{let}\ x = (\mathit{grant}\ R\ \mathit{in}\ e_1\sigma)\ \mathit{in}\ (\mathit{grant}\ R\ \mathit{in}\ e_2\sigma) \Downarrow_D^S o$

since they are both equivalent to the following disjunction.

(1) $o = \mathit{fail}$ and $e_2\sigma \Downarrow_{D\cup(R\cap S)}^S \mathit{fail}$, or

(2) $e_2\sigma \Downarrow_{D\cup(R\cap S)}^S v$ and $e_1\sigma\{x \leftarrow v\} \Downarrow_{D\cup(R\cap S)}^S o$ for some v.

Hence, by Proposition 3 and Theorem 4, $\mathit{grant}\ R\ \mathit{in}\ (\mathit{let}\ x = e_1\ \mathit{in}\ e_2) \equiv \mathit{let}\ x = (\mathit{grant}\ R\ \mathit{in}\ e_1)\ \mathit{in}\ (\mathit{grant}\ R\ \mathit{in}\ e_2)$.

(Grant Grant) $\mathit{grant}\ R_1\ \mathit{in}\ \mathit{grant}\ R_2\ \mathit{in}\ e \equiv \mathit{grant}\ R_1{\cup}R_2\ \mathit{in}\ e$

Consider any D and S such that $D \subseteq S$, and any substitution σ sending variables to closed values with $\mathit{dom}(\sigma) = \mathit{fv}(e)$, and any o. Since (Eval Grant) is the only evaluation rule applicable to a grant, we have:

$$\begin{aligned}
\mathit{grant}\ R_1\ \mathit{in}\ \mathit{grant}\ R_2\ \mathit{in}\ e\sigma \Downarrow_D^S o &\iff \mathit{grant}\ R_2\ \mathit{in}\ e\sigma \Downarrow_{D\cup(R_1\cap S)}^S o \\
&\iff e\sigma \Downarrow_{D\cup(R_1\cap S)\cup(R_2\cap S)}^S o \\
&\iff \mathit{grant}\ R_1{\cup}R_2\ \mathit{in}\ e\sigma \Downarrow_D^S o
\end{aligned}$$

Hence, by Proposition 3 and Theorem 4, $\mathit{grant}\ R_1\ \mathit{in}\ \mathit{grant}\ R_2\ \mathit{in}\ e \simeq \mathit{grant}\ R_1{\cup}R_2\ \mathit{in}\ e$.

(Grant Frame) $\mathit{grant}\ R_1\ \mathit{in}\ R_2[e] \equiv \mathit{grant}\ R_1{\cap}R_2\ \mathit{in}\ R_2[e]$

Consider any D and S such that $D \subseteq S$, and any substitution σ sending variables to closed values with $\mathit{dom}(\sigma) = \mathit{fv}(e)$, and any o. Note that:

$$(D \cup (R_1 \cap S)) \cap R_2 \quad = \quad (D \cup (R_1 \cap R_2 \cap S)) \cap R_2$$

Hence, since (Eval Frame) and (Eval Grant) are the only evaluation rules applicable to a framed expression and a grant, respectively, we have:

$$\begin{array}{rcl} \mathit{grant}\ R_1\ \mathit{in}\ R_2[e\sigma] \Downarrow_D^S o & \iff & R_2[e\sigma] \Downarrow_{D\cup(R_1\cap S)}^S o \\ & \iff & e\sigma \Downarrow_{(D\cup(R_1\cap S))\cap R_2}^{R_2} o \\ & \iff & e\sigma \Downarrow_{(D\cup(R_1\cap R_2\cap S))\cap R_2}^{R_2} o \\ & \iff & R_2[e\sigma] \Downarrow_{D\cup(R_1\cap R_2\cap S)}^S o \\ & \iff & \mathit{grant}\ R_1{\cap}R_2\ \mathit{in}\ R_2[e\sigma] \Downarrow_D^S o \end{array}$$

Hence, by Proposition 3 and Theorem 4:

$$\mathit{grant}\ R_1\ \mathit{in}\ R_2[e] \simeq \mathit{grant}\ R_1{\cap}R_2\ \mathit{in}\ R_2[e]$$

(Grant Frame Grant) $\mathit{grant}\ R_2\ \mathit{in}\ R_1[\mathit{grant}\ R_2\ \mathit{in}\ e] \equiv R_1[\mathit{grant}\ R_2\ \mathit{in}\ e]$

Consider any D and S such that $D \subseteq S$, and any substitution σ sending variables to closed values with $\mathit{dom}(\sigma) = \mathit{fv}(e)$, and any o. Note that:

$$\begin{array}{rcl} ((D \cup (R_2 \cap S)) \cap R_1) \cup (R_2 \cap R_1) & = & (D \cup (R_2 \cap S) \cup R_2) \cap R_1 \\ & = & (D \cup R_2) \cap R_1 \\ & = & (D \cap R_1) \cup (R_2 \cap R_1) \end{array}$$

Hence, since (Eval Frame) and (Eval Grant) are the only evaluation rules applicable to a framed expression and a grant, respectively, we have:

$$\begin{array}{l} \mathit{grant}\ R_2\ \mathit{in}\ R_1[\mathit{grant}\ R_2\ \mathit{in}\ e] \Downarrow_D^S o \\ \quad \iff R_1[\mathit{grant}\ R_2\ \mathit{in}\ e] \Downarrow_{D\cup(R_2\cap S)}^S o \\ \quad \iff \mathit{grant}\ R_2\ \mathit{in}\ e \Downarrow_{(D\cup(R_2\cap S))\cap R_1}^{R_1} o \\ \quad \iff e \Downarrow_{((D\cup(R_2\cap S))\cap R_1)\cup(R_2\cap R_1)}^{R_1} o \\ \quad \iff e \Downarrow_{(D\cap R_1)\cup(R_2\cap R_1)}^{R_1} o \\ \quad \iff \mathit{grant}\ R_2\ \mathit{in}\ e \Downarrow_{D\cap R_1}^{R_1} o \\ \quad \iff R_1[\mathit{grant}\ R_2\ \mathit{in}\ e] \Downarrow_D^S o \end{array}$$

Hence, by Proposition 3 and Theorem 4:

$$\mathit{grant}\ R_2\ \mathit{in}\ R_1[\mathit{grant}\ R_2\ \mathit{in}\ e] \simeq R_1[\mathit{grant}\ R_2\ \mathit{in}\ e]$$

(Test $\varnothing$) $\mathit{test}\ \varnothing\ \mathit{then}\ e_1\ \mathit{else}\ e_2 \equiv e_1$

Consider any D and S such that $D \subseteq S$, and any substitution σ sending variables to closed values with $\mathit{dom}(\sigma) = \mathit{fv}(e_1\, e_2)$, and any o. Since (Eval Test) is the only evaluation rule applicable to a test, and $\varnothing \subseteq D$, we have:

$$\mathit{test}\ \varnothing\ \mathit{then}\ e_1\sigma\ \mathit{else}\ e_2\sigma \Downarrow_D^S o \iff e_1\sigma \Downarrow_D^S o$$

Hence, by Proposition 3 and Theorem 4, $\mathit{test}\ \varnothing\ \mathit{then}\ e_1\ \mathit{else}\ e_2 \simeq e_1$.

(Test Refl) $\mathit{test}\ R\ \mathit{then}\ e\ \mathit{else}\ e \equiv e$

Consider any D and S such that $D \subseteq S$, and any substitution σ sending variables to closed values with $\mathit{dom}(\sigma) = \mathit{fv}(e_1\, e_2)$, and any o. Since (Eval Test) is the only evaluation rule applicable to a test, we have:

$$\mathit{test}\ R\ \mathit{then}\ e\sigma\ \mathit{else}\ e\sigma \Downarrow_D^S o \iff e\sigma \Downarrow_D^S o$$

Hence, by Proposition 3 and Theorem 4, $\mathit{test}\ R\ \mathit{then}\ e\ \mathit{else}\ e \simeq e$.

(Test $\cup$) $\mathit{test}\ R_1{\cup}R_2\ \mathit{then}\ e_1\ \mathit{else}\ e_2 \equiv$
$\mathit{test}\ R_1\ \mathit{then}\ (\mathit{test}\ R_2\ \mathit{then}\ e_1\ \mathit{else}\ e_2)\ \mathit{else}\ e_2$

Consider any D and S such that $D \subseteq S$, and any substitution σ sending variables to closed values with $\mathit{dom}(\sigma) = \mathit{fv}(e_1\, e_2)$, and any o. Since (Eval Test) is the only evaluation rule applicable to a test, we have:

$$\begin{aligned}
&\mathit{test}\ R_1{\cup}R_2\ \mathit{then}\ e_1\sigma\ \mathit{else}\ e_2\sigma \Downarrow_D^S o \\
&\iff e_i\sigma \Downarrow_D^S o \\
&\qquad \text{where } i = 1 \text{ if } R_1 \cup R_2 \subseteq D, \text{ otherwise } i = 2 \\
&\iff e_i\sigma \Downarrow_D^S o \\
&\qquad \text{where } i = 1 \text{ if } R_1 \subseteq D \text{ and } R_2 \subseteq D, \text{ otherwise } i = 2 \\
&\iff \mathit{test}\ R_1\ \mathit{then}\ (\mathit{test}\ R_2\ \mathit{then}\ e_1\sigma\ \mathit{else}\ e_2\sigma)\ \mathit{else}\ e_2\sigma \Downarrow_D^S o
\end{aligned}$$

Hence, by Proposition 3 and Theorem 4:

$$\begin{aligned}
&\mathit{test}\ R_1{\cup}R_2\ \mathit{then}\ e_1\ \mathit{else}\ e_2 \simeq \\
&\mathit{test}\ R_1\ \mathit{then}\ (\mathit{test}\ R_2\ \mathit{then}\ e_1\ \mathit{else}\ e_2)\ \mathit{else}\ e_2
\end{aligned}$$

(Test Grant) $\mathit{test}\ R\ \mathit{then}\ e_1\ \mathit{else}\ e_2 \equiv$
$\mathit{test}\ R\ \mathit{then}\ (\mathit{grant}\ R\ \mathit{in}\ e_1)\ \mathit{else}\ e_2$

Consider any D and S such that $D \subseteq S$, and any substitution σ sending variables to closed values with $dom(\sigma) = fv(e_1\ e_2)$, and any o. Since (Eval Test) and (Eval Grant) are the only evaluation rules applicable to tests and grants, respectively, we have:

$$
\begin{array}{ll}
\multicolumn{2}{l}{\mathit{test}\ R\ \mathit{then}\ e_1\sigma\ \mathit{else}\ e_2\sigma \Downarrow_D^S o} \\
\iff & (R \subseteq D \wedge e_1\sigma \Downarrow_D^S o) \vee (\neg(R \subseteq D) \wedge e_2\sigma \Downarrow_D^S o) \\
\iff & (R \subseteq D \wedge e_1\sigma \Downarrow_{D\cup(R\cap S)}^S o) \vee (\neg(R \subseteq D) \wedge e_2\sigma \Downarrow_D^S o) \\
\iff & (R \subseteq D \wedge \mathit{grant}\ R\ \mathit{in}\ e_1\sigma \Downarrow_D^S o) \vee (\neg(R \subseteq D) \wedge e_2\sigma \Downarrow_D^S o) \\
\iff & \mathit{test}\ R\ \mathit{then}\ (\mathit{grant}\ R\ \mathit{in}\ e_1)\ \mathit{else}\ e_2 \Downarrow_D^S o
\end{array}
$$

Hence, by Proposition 3 and Theorem 4:

$$\mathit{test}\ R\ \mathit{then}\ e_1\ \mathit{else}\ e_2 \simeq \mathit{test}\ R\ \mathit{then}\ (\mathit{grant}\ R\ \mathit{in}\ e_1)\ \mathit{else}\ e_2$$

(Eq Fail Rator) $\mathit{fail}\ e \equiv \mathit{fail}$

Consider any D and S such that $D \subseteq S$, and any substitution σ sending variables to closed values with $dom(\sigma) = fv(e)$, and any o. By inspection of the three evaluation rules applicable to an application, and by Lemma 6, we have:

$$
\begin{array}{rcl}
\mathit{fail}\ (e\sigma) \Downarrow_D^S o & \iff & o = \mathit{fail} \\
& \iff & \mathit{fail} \Downarrow_D^S o
\end{array}
$$

Hence, by Proposition 3 and Theorem 4, $\mathit{fail}\ e \simeq \mathit{fail}$.

(Eq Fail Rand) $v\ \mathit{fail} \equiv \mathit{fail}$

Consider any D and S such that $D \subseteq S$, and any substitution σ sending variables to closed values with $dom(\sigma) = fv(e)$, and any o. By inspection of the three evaluation rules applicable to an application, and by Lemma 6, we have:

$$
\begin{array}{rcl}
v\sigma\ \mathit{fail} \Downarrow_D^S o & \iff & o = \mathit{fail} \\
& \iff & \mathit{fail} \Downarrow_D^S o
\end{array}
$$

Hence, by Proposition 3 and Theorem 4, $v\ \mathit{fail} \simeq \mathit{fail}$. □

References

[1] M. Abadi, B. Lampson, and J.-J. Lévy. Analysis and caching of dependencies. In *First ACM SIGPLAN International Conference on Functional Programming (ICFP'96)*, pages 83–91, May 1996.

[2] S. Abramsky and L. Ong. Full abstraction in the lazy lambda calculus. *Information and Computation*, 105:159–267, 1993.

[3] A. Banerjee and D. Naumann. A simple semantics and static analysis for Java security. CS Report 2001–1, Stevens Institute of Technology, 2001.

[4] A. Banerjee and D. Naumann. Representation independence, confinement, and access control. In *29th ACM Symposium on Principles of Programming Languages (POPL'02)*, 2002. To appear.

[5] M. Bartoletti, P. Degano, and G. Ferrari. Static analysis for stack inspection. In *ConCoord: International Workshop on Concurrency and Coordination*, volume 54 of *ENTCS*. Elsevier, 2001.

[6] N. Benton, A. Kennedy, and G. Russell. Compiling Standard ML to Java bytecodes. In *Third ACM SIGPLAN International Conference on Functional Programming (ICFP'98)*, pages 129–140, 1998.

[7] F. Besson, T. Jensen, D. L. Métayer, and T. Thorn. Model checking security properties of control flow graphs. *Journal of Computer Security*, 9:217–250, 2001.

[8] D. Box. *Essential .NET Volume I: The Common Language Runtime*. Addison Wesley, 2002. To appear.

[9] Ú. Erlingsson and F. Schneider. IRM enforcement of Java stack inspection. In *Proceedings 2000 IEEE Symposium on Security and Privacy*, pages 246–255. IEEE Computer Society Press, 2000.

[10] L. Gong. *Inside Java™ 2 Platform Security*. Addison Wesley, 1999.

[11] A. Gordon and A. Pitts, editors. *Higher Order Operational Techniques in Semantics*, Publications of the Newton Institute. Cambridge University Press, 1998.

[12] A. D. Gordon. Bisimilarity as a theory of functional programming. *Theoretical Computer Science*, 228:5–47, 1999.

[13] D. Grossman, G. Morrisett, and S. Zdancewic. Syntactic type abstraction. *ACM Transactions on Programming Languages and Systems*, 22(6):1037–1080, 2000.

[14] N. Hardy. The confused deputy. *ACM Operating Systems Review*, 22(4):36–38, Oct 1988. http://www.cis.upenn.edu/~KeyKOS/ConfusedDeputy.html.

[15] D. J. Howe. Proving congruence of bisimulation in functional programming languages. *Information and Computation*, 124(2):103–112, 1996.

[16] T. Jensen, D. L. Metayer, and T. Thorn. Verification of control flow based security properties. In *Proceedings 1999 IEEE Symposium on Security and Privacy*, pages 89–103. IEEE Computer Society Press, 1999.

[17] G. Karjoth. An operational semantics for Java 2 access control. In *13th Computer Security Foundations Workshop*, pages 224–232. IEEE Computer Society Press, 2000.

[18] X. Leroy and F. Rouaix. Security properties of typed applets. In J. Vitek and C. Jensen, editors, *Secure Internet Programming – Security issues for Mobile and Distributed Objects*, volume 1603 of *LNCS*, pages 147–182. Springer-Verlag, 1999.

[19] T. Lindholm and F. Yellin. *The Java™ Virtual Machine Specification*. Addison Wesley, 1997.

[20] Microsoft Corporation. *.NET Framework Developer's Guide: Security Optimizations*, 2001. http://msdn.microsoft.com/library/en-us/cpguidnf/html/cpconsecurityoptimizations.asp.

[21] R. Milner. Fully abstract models of typed lambda-calculi. *Theoretical Computer Science*, 4:1–23, 1977.

[22] R. Milner. *Communication and Concurrency*. Prentice-Hall, 1989.

[23] E. Moggi. Notions of computations and monads. *Theoretical Computer Science*, 93:55–92, 1989.

[24] J. H. Morris. *Lambda-Calculus Models of Programming Languages*. PhD thesis, MIT, Dec. 1968.

[25] A. C. Myers. JFlow: Practical, mostly-static information flow control. In *26th ACM Symposium on Principles of Programming Languages (POPL'99)*, pages 228–241, 1999.

[26] P. Ørbæk and J. Palsberg. Trust in the λ-calculus. *Journal of Functional Programming*, 3(2):75–85, 1997.

[27] G. D. Plotkin. Call-by-name, call-by-value and the λ-calculus. *Theoretical Computer Science*, 1:125–159, 1975.

[28] F. Pottier, C. Skalka, and S. Smith. A systematic approach to access control. In *Programming Languages and Systems (ESOP 2001)*, volume 2028 of *LNCS*, pages 30–45. Springer, 2001.

[29] M. Schinz and M. Odersky. Tail call elimination on the Java Virtual Machine. In *SIGPLAN Workshop on Multi-Language Infrastructure and Interoperability (BABEL'01)*, volume 59(1) of *ENTCS*, pages 155–168. Elsevier, 2001.

[30] C. Skalka and S. Smith. Static enforcement of security with types. In *Fifth ACM SIGPLAN International Conference on Functional Programming (ICFP'00)*, pages 34–45, 2000.

[31] D. S. Wallach, A. W. Appel, and E. W. Felten. SAFKASI: A security mechanism for language-based systems. *ACM Transactions on Software Engineering and Methodology*, 9(4):341–378, 2000.