A Tail-Recursive Machine with Stack Inspection

JOHN CLEMENTS and MATTHIAS FELLEISEN
Northeastern University

Security folklore holds that a security mechanism based on stack inspection is incompatible with a global tail call optimization policy; that an implementation of such a language must allocate memory for a source-code tail call, and a program that uses only tail calls (and no other memory-allocating construct) may nevertheless exhaust the available memory. In this article, we prove this widely held belief wrong. We exhibit an abstract machine for a language with security stack inspection whose space consumption function is equivalent to that of the canonical tail call optimizing abstract machine. Our machine is surprisingly simple and suggests that tail calls are as easy to implement in a security setting as they are in a conventional one.

Categories and Subject Descriptors: D.3.1 [**Programming Languages**]: Formal Definitions and Theory—*Semantics*; D.3.3 [**Programming Languages**]: Language Constructs and Features—*Dynamic storage management*

General Terms: Languages, Security

Additional Key Words and Phrases: Stack inspection, tail call optimization, tail recursion

1. STACKS, SECURITY, AND TAIL CALLS

Over the last 10 years, programming language implementors have spent significant effort on security issues. This effort takes many forms; one is the implementation of a strategy known as *stack inspection* [Wallach et al. 1997]. It starts from the premise that trusted components may authorize potentially insecure actions for the dynamic extent of some expression, provided that all intermediate calls are made by and to trusted code.

In its conventional implementation, stack inspection is incompatible with a traditional language semantics, because it clashes with the well-established idea of modeling function calls with a β or β_v reduction [Plotkin 1975]. A β reduction replaces a function's application with the body of that function, with the function's parameters replaced by the application's arguments. In a language with stack inspection, a β or β_v reduction thus disposes of information that is necessary to evaluate the security primitives.

An earlier version of this article was presented at the 2003 European Symposium on Programming, in Warsaw, Poland.
Authors' address: College of Computer Science, Northeastern University, 202 West Village H, Boston, MA 02115; email: {clements,Matthias}@ccs.neu.edu.

For this reason, Fournet and Gordon [2002] modeled function calls with a nonstandard β reduction. To be more precise, β does not hold as an equation for source terms. Instead, abstraction bodies are wrapped with context-building primitives. Unfortunately, this formalization inhibits a transformation of this semantics into a tail-call-optimizing (TCO) implementation. Fournet and Gordon recognized this fact and stated that "[S]tack inspection profoundly affects the semantics of all programs. In particular, it invalidates . . . tail call optimizations" [Fournet and Gordon 2002, p. 307].

This understanding of the stack inspection protocol also pervades the implementation of existing runtime systems. The Java design team, for example, chose not to provide a TCO implementation in part because of the perceived incompatibility between tail call optimizations and stack inspection.[1] The .NET effort at Microsoft provides a runtime system that is properly TCO—except in the presence of security primitives, which disable it. Microsoft's documentation [Microsoft 2002] states that "[t]he current frame cannot be discarded when control is transferred from untrusted code to trusted code, since this would jeopardize code identity security."

Wallach et al. [2000] suggested an alternative implementation of stack inspection that might accommodate TCO. They added an argument to each function call that represents the security context as a statement in their belief logic. Statements in this belief logic can be unraveled to determine whether an operation is permitted. However, the details of their memory behavior are opaque. In particular, they did not attempt to present a model in which memory usage can be analyzed.

Our work fills the gap between Fournet and Gordon's [2002] formal model and Wallach's alternative implementation of stack inspection. Specifically, our security model exploits a novel mechanism for lightweight stack inspection [Flatt 1995–2002]. We demonstrate the equivalence between our model and Fournet and Gordon's, and prove our claims of TCO. More precisely, our abstract implementation can transform *all* tail calls in the source program into instructions that do not consume any stack (or store) space. Moreover, our abstract implementation represents a relatively minor change to the models used by current implementations, suggesting that these implementations might accommodate TCO with minimal effort.

We proceed as follows. First, we derive a CESK machine from Fournet and Gordon's [2002] semantics. Second, we develop a different, but extensionally equivalent CESK machine that uses a variant of Flatt [1995–2002] lightweight stack inspection mechanism. Third, we show that our machine uses strictly less space than the machine derived from Fournet and Gordon's semantics and that our machine uses as much space as Clinger's canonical TCO CESK machine [Clinger 1998].

The article consists of nine sections. The second section introduces the λ_{sec} language: its syntax, semantics, and security mechanisms. The third section shows how a pair of tail calls between system and applet code can allocate an unbounded amount of space. In the fourth section, we derive an extensionally

[1]Private communication between Guy Steele and second author at POPL 1996.

equivalent CESK machine from Fournet and Gordon's [2002] semantics; in the fifth section, we modify this machine so that it implements all tail calls in a properly optimized fashion. The sixth section provides a precise analysis of the space consumption of these machines and shows that our new machine is indeed TCO. In the seventh section, we discuss the extension of our models for λ_{sec} to the richer environments of existing languages. The last two sections place our work into context.

2. THE λ_{SEC} LANGUAGE

Fournet and Gordon [2002] worked from the λ_{sec}-calculus [Pottier et al. 2001; Skalka and Smith 2000]. This calculus is a simple model of a programming language with security annotations. They presented two languages: a source language, in which program components are written, and a target language, which includes an additional form for security annotations. A trusted annotator performs the translation from the source to the target, annotating each component with the appropriate permissions.

In this security model, all code is statically annotated with a given set of permissions, chosen from a fixed set $\mathcal{P}$. A program component that has permissions R may choose to enable some or all of these permissions. The set of enabled permissions at any point during execution is determined by taking the intersection of the permissions enabled for the caller and the set of permissions contained in the callee's annotation. That is, a permission is considered enabled only if two conditions are met: first, it must have been legally and explicitly enabled by some calling procedure, and second, all intervening callers must have been annotated with this permission.

A program component consists of a set of permissions and a λ-expression from the source language, (M_s). This language adds three expressions to the basic call-by-value λ-calculus. The test expression checks to see whether a given set of permissions is currently enabled, and branches based on that decision. The grant expression enables a privilege, provided that the context endows it with those permissions. Finally, the fail expression causes the program to halt immediately, signaling a security failure. Our particular source language also changes the traditional presentation of the λ-calculus by adding an explicit name to each abstraction so that we get concise definitions of recursive procedures.

SYNTAX

$$
\begin{aligned}
C \in \text{Components} &= \langle R, \lambda_f x.M_s \rangle \\
M, N &= x \mid M\ N \mid \lambda_f x.M \mid \mathsf{grant}\ R\ \mathsf{in}\ M \\
&\quad \mid \mathsf{test}\ R\ \mathsf{then}\ M\ \mathsf{else}\ N \mid \mathsf{fail} \mid \underline{R[M]} \\
x &\in \text{Identifiers} \\
R &\subseteq \mathcal{P} \\
V \in \text{Values} &= x \mid \lambda_f x.M
\end{aligned}
$$

The target language (M) adds a framing expression to this source language (underlined in the grammar). A frame specifies the permissions of a component in the source text. To ensure that these framing expressions are present as the program is evaluated, we translate source components into target components by annotating the component's source term with its permissions. The annotator below performs this annotation, and simultaneously ensures that a grant expression refers only to those permissions to which it is entitled by its source location.

ANNOTATOR $\mathcal{A} : 2^{\mathcal{P}} \times M_s \to M$

$$\begin{aligned}
\mathcal{A}\langle R, [\![x]\!]\rangle &= x \\
\mathcal{A}\langle R, [\![\lambda_f x.M]\!]\rangle &= \lambda_f x.R[\mathcal{A}\langle R, [\![M]\!]\rangle] \\
\mathcal{A}\langle R, [\![M\ N]\!]\rangle &= \mathcal{A}\langle R, [\![M]\!]\rangle\ \mathcal{A}\langle R, [\![N]\!]\rangle \\
\mathcal{A}\langle R, [\![\mathsf{grant}\ S\ \mathsf{in}\ M]\!]\rangle &= \mathsf{grant}\ S \cap R\ \mathsf{in}\ \mathcal{A}\langle R, [\![M]\!]\rangle \\
\mathcal{A}\langle R, [\![\mathsf{test}\ S\ \mathsf{then}\ M\ \mathsf{else}\ N]\!]\rangle &= \mathsf{test}\ S\ \mathsf{then}\ \mathcal{A}\langle R, [\![M]\!]\rangle\ \mathsf{else}\ \mathcal{A}\langle R, [\![N]\!]\rangle \\
\mathcal{A}\langle R, [\![\mathsf{fail}]\!]\rangle &= \mathsf{fail}
\end{aligned}$$

The annotator $\mathcal{A}$ consumes two arguments: the set of permissions appropriate for the source and the source code; it produces a target expression. It commutes with all expression constructors except for λ and grant. For a λ expression, it adds a frame expression wrapping the body. For a grant expression, it replaces the permissions S that the expression specifies with the intersection $S \cap R$. So, if a component containing the expression grant $\{a, b\}$ in E were annotated with the permissions $\{b, c\}$, the resulting expression would read grant $\{b\}$ in E', where E' represents the recursive annotation of E.

We adapt Fournet and Gordon's [2002] semantics to our variant of λ_{sec} mutatis mutandis. Evaluation of programs is specified using a reduction semantics based on evaluation contexts [Felleisen and Friedman 1986]. In such a semantics, every expression is divided into an evaluation context containing a single hole (denoted by $\bullet$), and a redex. An evaluation context is composed with a redex by replacing the context's hole with the redex. The choice of evaluation contexts determines where evaluation can occur, and typically the evaluation contexts are chosen to enforce deterministic evaluation; that is, each expression has a unique decomposition into context and redex. Reduction rules in such a semantics take the form "$E[f] \mapsto E[g]$," where f is a redex, g is its contractum, and E is the context (which may be observable, as it is in the test rule).

CONTEXTS

$$E = \bullet \mid E\ M \mid V\ E \mid \mathsf{grant}\ R\ \mathsf{in}\ E \mid R[E]$$

REDUCTION RULES

$$\begin{aligned}
E[\lambda_f x.M\ V] &\mapsto E[[\lambda_f x.M/f][V/x]M] \\
E[R[V]] &\mapsto E[V] \\
E[\mathsf{grant}\ R\ \mathsf{in}\ V] &\mapsto E[V] \\
E[\mathsf{test}\ R\ \mathsf{then}\ M\ \mathsf{else}\ N] &\mapsto \begin{cases} E[M] \text{ if } \mathcal{OK}\langle R, [\![E]\!]\rangle \\ E[N] \text{ otherwise} \end{cases} \\
E[\mathsf{fail}] &\mapsto \mathsf{fail}
\end{aligned}$$

where

$$\begin{array}{rl}
\mathcal{OK}\langle\emptyset, [\![E]\!]\rangle & \\
\mathcal{OK}\langle R, [\![\bullet]\!]\rangle & \\
\mathcal{OK}\langle R, [\![E[\bullet\ M]]\!]\rangle & \text{iff } \mathcal{OK}\langle R, [\![E]\!]\rangle \\
\mathcal{OK}\langle R, [\![E[V\ \bullet]]\!]\rangle & \text{iff } \mathcal{OK}\langle R, [\![E]\!]\rangle \\
\mathcal{OK}\langle R, [\![E[S[\bullet]]]\!]\rangle & \text{iff } R \subseteq S \text{ and } \mathcal{OK}\langle R, [\![E]\!]\rangle \\
\mathcal{OK}\langle R, [\![E[\mathsf{grant}\ S\ \mathsf{in}\ \bullet]]\!]\rangle & \text{iff } \mathcal{OK}\langle (R - S), [\![E]\!]\rangle
\end{array}$$

This semantics is an extension of a standard call-by-value reduction semantics. The hole and the two application contexts are standard and enforce left-to-right evaluation of arguments. The reduction rule for applications is also standard. The added contexts and reduction rules for frame and grant expressions are largely transparent; evaluation may proceed inside of either form, and each one disappears when its expression is a value. These expressions affect the evaluation only when a test expression occurs as a redex. In this case, the result of the reduction depends on the $\mathcal{OK}$ predicate, which is applied to the current context and the desired permissions.

The $\mathcal{OK}$ predicate recurs over the evaluation context from the inside out, succeeding either when the permissions remaining to check are empty or when the context is exhausted.[2] The $\mathcal{OK}$ predicate commutes with both kinds of application context. In the case of a frame annotation, the desired permissions must occur in the frame, and the predicate must succeed recursively. Finally, a grant expression removes all permissions it grants from the set of those that need to be checked.

Finally, the Eval function determines the meaning of a source program. A program consists of a list of components. Evaluation is performed by annotating each λ-expression with the permissions of its component, and combining all such expressions into a single application. This application uses the traditional abbreviation of a curried application as a single one.

Definition (*Eval*).

$$\text{Eval}(C, \ldots) = V \text{ if } (\mathcal{A}(C)\ \cdots) \overset{*}{\mapsto} V$$

Since the first component is applied to the rest, it is presumed to represent the runtime system, or at least a linker. Eval is undefined for programs that diverge or enter a stuck state.

Minor Differences. The semantics we present differs from that of Fournet and Gordon [2002] in three ways. First, it reduces programs containing fail to a final fail state in one step, rather than propagating the fail upward one expression at a time. We consider this difference trivial and ignore it. Second, our language includes a named lambda, to simplify the presentation of recursive examples. Since we present an untyped language, a recursive function always

[2]In fact, success on an empty permission set may be derived from the other rules in the definition; the direct statement of this is nevertheless included to simplify understanding.

has an equivalent form as an application of the Y combinator. Third, our semantics replaces a runtime check in Fournet and Gordon's semantics with a static check. Appendix A presents a proof of the equivalence of our evaluator and theirs.

3. TAIL CALL OPTIMIZATION

Modern functional programming languages avoid looping constructs in favor of recursion. Doing so keeps the language smaller and simplifies its implementation. Furthermore, it empowers programmers to match functions and data structures, which makes programs more comprehensible than random mixtures of loops and function calls. Even modern object-oriented programmers have recognized this fact, as indicated by the inclusion of tail call instructions in Microsoft's CLR [Box 2002] and the promotion of traversal strategies such as the interpreter, composite, or visitor patterns [Gamma et al. 1995].

Of course, if function calls were implemented naïvely, this strategy would introduce an unacceptably large overhead on iterative computations. Each iteration would consume a stack frame and long loops would quickly run out of space. As Guy Steele pointed out in the late 1970s, however, language designers can have efficiency and a small language if they translate so-called tail calls into instruction sequences that do not consume any space [Steele Jr. 1977]. Typically, such function calls turn into plain jumps, and hence the translation of a tail-recursive function equals the translation of an equivalent looping construct. Using this reasoning, the language definitions for Scheme require that correct implementations must optimize all tail calls and thereby "support an unbounded number of active tail calls" [Kelsey et al. 1998, p. 7].

At first glance, tail call optimization seems inherently incompatible with stack inspection. To see this, consider a mutually recursive loop between applet and library code.

ABBREVIATIONS

$$\begin{aligned}
\text{UserFn} &\overset{\Delta}{=} \lambda_{user} sys.sys\ user \\
\text{SystemFn} &\overset{\Delta}{=} \lambda_{sys} user.user\ sys \\
\mathcal{A}\langle R_{\mathsf{A}}, [\![\text{UserFn}]\!]\rangle &= \lambda_{user} sys.R_{\mathsf{A}}[sys\ user] \\
\mathcal{A}\langle R_{\mathsf{S}}, [\![\text{SystemFn}]\!]\rangle &= \lambda_{sys} user.R_{\mathsf{S}}[user\ sys]
\end{aligned}$$

REDUCTION (WITH ANNOTATIONS)

$$\begin{aligned}
&\mathcal{A}\langle R_{\mathsf{A}}, [\![\text{UserFn}]\!]\rangle\ \mathcal{A}\langle R_{\mathsf{S}}, [\![\text{SystemFn}]\!]\rangle \\
&\mapsto R_{\mathsf{A}}[\mathcal{A}\langle R_{\mathsf{S}}, [\![\text{SystemFn}]\!]\rangle\ \mathcal{A}\langle R_{\mathsf{A}}, [\![\text{UserFn}]\!]\rangle] \\
&\mapsto R_{\mathsf{A}}[R_{\mathsf{S}}[\mathcal{A}\langle R_{\mathsf{A}}, [\![\text{UserFn}]\!]\rangle\ \mathcal{A}\langle R_{\mathsf{S}}, [\![\text{SystemFn}]\!]\rangle]] \\
&\mapsto R_{\mathsf{A}}[R_{\mathsf{S}}[R_{\mathsf{A}}[\mathcal{A}\langle R_{\mathsf{S}}, [\![\text{SystemFn}]\!]\rangle\ \mathcal{A}\langle R_{\mathsf{A}}, [\![\text{UserFn}]\!]\rangle]]] \\
&\mapsto R_{\mathsf{A}}[R_{\mathsf{S}}[R_{\mathsf{A}}[R_{\mathsf{S}}[\mathcal{A}\langle R_{\mathsf{A}}, [\![\text{UserFn}]\!]\rangle\ \mathcal{A}\langle R_{\mathsf{S}}, [\![\text{SystemFn}]\!]\rangle]]]] \\
&\quad\ldots
\end{aligned}$$

REDUCTION (WITHOUT ANNOTATIONS)

$$\begin{aligned}
&\text{UserFn SystemFn} \\
&\mapsto \text{SystemFn UserFn}
\end{aligned}$$

$\mapsto$ UserFn SystemFn
$\mapsto$ SystemFn UserFn
$\mapsto$ UserFn SystemFn
...

This program consists of two copies of a mutually recursive loop function, one a "user" component and one a "system" component. Each takes the other as an argument, and then calls it, passing itself as the sole argument. To simplify the presentation of the looping functions, we introduce abbreviations for the user and system procedures.

This program is a toy example, but it represents the core of many interactions between user and system code. For instance, any co-routine-style interaction between producer and consumer exhibits this behavior—unfortunately, programmers are forced to avoid this powerful and natural style in Java precisely because of the lack of TCO. Perhaps the most common examples of this kind of interaction occur in OO-style traversals of data structures, such as the above-mentioned patterns.

The first reduction sequence in the figure shows how λ_{sec} evaluates the given program, where the two procedures are annotated with their permissions. The context quickly grows without bound in this example. A functional programmer would expect to see a sequence more like the second one. This series is also a reduction sequence in λ_{sec}, but one that is obtained by evaluating the program's pure source, without the security annotations.

As Fournet and Gordon [2002] pointed out in their article, all is not lost. They introduced an additional reduction into their abstract machine that explicitly removed a frame before performing a call. Unfortunately, as they pointed out, indiscriminate application of this rule changes the semantics of the language. They addressed this problem with a partial list of circumstances in which the reduction is legal. By casting tail-call elimination as a specific reduction rather than a property of an abstract machine, Fournet and Gordon failed to realize that a fully tail-recursive implementation of the language is possible.

4. AN ABSTRACT MACHINE FOR λ_{SEC}

Following Clinger's [1998] work on defining tail-optimized languages via space complexity classes, we first reformulate the λ_{sec} semantics as a CESK machine [Felleisen and Friedman 1986; Felleisen and Flatt 1989–2002]. We can then measure the space consumed by machine configurations, programs, and machines. Furthermore, we can determine whether the space consumption function of an implementation is in the same complexity class as Clinger's machine.

4.1 The fg Machine

We begin with a direct translation of λ_{sec}'s semantics into a CESK machine, which we call *frame-generating* or *fg* (see Figure 1). A CESK abstract machine takes its name from its four registers: the control string, the environment, the store, and the continuation. The control string indicates which program instruction is being reduced. In conventional machines, this is called the *program*

THE FG MACHINE

$$
\begin{array}{rl}
\text{Configurations} & = \langle M, \rho, \sigma, \kappa\rangle \mid \langle V, \rho, \sigma, \kappa\rangle \mid \langle V, \sigma\rangle \mid \mathsf{fail} \\
\text{Final Configurations} & = \langle V, \sigma\rangle \mid \mathsf{fail} \\
\kappa \in \text{Continuations} & = \langle\rangle \mid \langle \text{push} : M, \rho, \kappa\rangle \mid \langle \text{call} : V, \kappa\rangle \mid \langle \text{frame} : R, \kappa\rangle \mid \langle \text{grant} : R, \kappa\rangle \\
V \in \text{Values} & = \langle \text{closure} : M, \rho\rangle \\
\rho \in \text{Environments} & = \text{Identifiers} \rightarrow_f \text{Locations} \\
\alpha, \beta & \in \text{Locations} \\
\sigma \in \text{Stores} & = \text{Locations} \rightarrow_f \text{Values} \\
\text{empty}_{\mathsf{fg}} & = \langle\rangle
\end{array}
$$

$$
\begin{array}{rll}
\langle \lambda_f x.M, \rho, \sigma, \kappa\rangle & \mapsto_{\mathsf{fg}} & \langle\langle \text{closure} : \lambda_f x.M, \rho\rangle, \rho, \sigma, \kappa\rangle \\
\langle x, \rho, \sigma, \kappa\rangle & \mapsto_{\mathsf{fg}} & \langle \sigma(\rho(x)), \rho, \sigma, \kappa\rangle \\
\langle M\ N, \rho, \sigma, \kappa\rangle & \mapsto_{\mathsf{fg}} & \langle M, \rho, \sigma, \langle \text{push} : N, \rho, \kappa\rangle\rangle \\
\langle R[M], \rho, \sigma, \kappa\rangle & \mapsto_{\mathsf{fg}} & \langle M, \rho, \sigma, \langle \text{frame} : R, \kappa\rangle\rangle \\
\langle \mathsf{grant}\ R\ \mathsf{in}\ M, \rho, \sigma, \kappa\rangle & \mapsto_{\mathsf{fg}} & \langle M, \rho, \sigma, \langle \text{grant} : R, \kappa\rangle\rangle \\
\langle \mathsf{test}\ R\ \mathsf{then}\ M\ \mathsf{else}\ N, \rho, \sigma, \kappa\rangle & \mapsto_{\mathsf{fg}} & \begin{cases} \langle M, \rho, \sigma, \kappa\rangle \text{ if } \mathcal{OK}_{\mathsf{fg}}\langle R, [\![\kappa]\!]\rangle \\ \langle N, \rho, \sigma, \kappa\rangle \text{ otherwise} \end{cases} \\
\langle \mathsf{fail}, \rho, \sigma, \kappa\rangle & \mapsto_{\mathsf{fg}} & \mathsf{fail} \\
\\
\langle V, \rho, \sigma, \langle\rangle\rangle & \mapsto_{\mathsf{fg}} & \langle V, \sigma\rangle \\
\langle V, \rho, \sigma, \langle \text{push} : M, \rho', \kappa\rangle\rangle & \mapsto_{\mathsf{fg}} & \langle M, \rho', \sigma, \langle \text{call} : V, \kappa\rangle\rangle \\
\langle V, \rho, \sigma, \langle \text{call} : V', \kappa\rangle\rangle & \mapsto_{\mathsf{fg}} & \langle M, \rho'[f \mapsto \beta][x \mapsto \alpha], \sigma[\alpha \mapsto V][\beta \mapsto V'], \kappa\rangle \\
 & & \text{if } V' = \langle \text{closure} : \lambda_f x.M, \rho'\rangle \text{ and } \alpha, \beta \notin \text{dom}(\sigma) \\
\langle V, \rho, \sigma, \langle \text{frame} : R, \kappa\rangle\rangle & \mapsto_{\mathsf{fg}} & \langle V, \rho, \sigma, \kappa\rangle \\
\langle V, \rho, \sigma, \langle \text{grant} : R, \kappa\rangle\rangle & \mapsto_{\mathsf{fg}} & \langle V, \rho, \sigma, \kappa\rangle \\
\\
\langle V, \rho, \sigma[\beta, \ldots \mapsto V', \ldots], \kappa\rangle & \mapsto_{\mathsf{fg}} & \langle V, \rho, \sigma, \kappa\rangle \\
 & & \text{if } \{\beta, \ldots\} \text{ is nonempty and} \\
 & & \beta, \ldots \text{ do not occur in } V, \rho, \sigma, \text{ or } \kappa
\end{array}
$$

where

$$
\begin{array}{rl}
\mathcal{OK}_{\mathsf{fg}}\langle \emptyset, [\![\kappa]\!]\rangle & \\
\mathcal{OK}_{\mathsf{fg}}\langle R, [\![\langle\rangle]\!]\rangle & \\
\mathcal{OK}_{\mathsf{fg}}\langle R, [\![\langle \text{push} : M, \rho, \kappa\rangle]\!]\rangle & \text{iff } \mathcal{OK}_{\mathsf{fg}}\langle R, [\![\kappa]\!]\rangle \\
\mathcal{OK}_{\mathsf{fg}}\langle R, [\![\langle \text{call} : V, \kappa\rangle]\!]\rangle & \text{iff } \mathcal{OK}_{\mathsf{fg}}\langle R, [\![\kappa]\!]\rangle \\
\mathcal{OK}_{\mathsf{fg}}\langle R, [\![\langle \text{frame} : R', \kappa\rangle]\!]\rangle & \text{iff } R \subseteq R' \text{ and } \mathcal{OK}_{\mathsf{fg}}\langle R, [\![\kappa]\!]\rangle \\
\mathcal{OK}_{\mathsf{fg}}\langle R, [\![\langle \text{grant} : R', \kappa\rangle]\!]\rangle & \text{iff } \mathcal{OK}_{\mathsf{fg}}\langle R - R', [\![\kappa]\!]\rangle
\end{array}
$$

Fig. 1. The FG machine.

counter. The environment binds variable names to values, much like the current stack frame of an assembly language machine. The store, like a heap, contains shared values.[3] Finally, the continuation represents the instruction's control context; it is analogous to the stack.

The derivation of a CESK machine from a reduction semantics is straightforward [Felleisen and Flatt 1989–2002]. In particular, the proof of equivalence of the two models is a refinement of Felleisen and Flatt's proof, which proceeds by a series of transformations from a simple reduction semantics to a register

[3]The store in our model is necessitated by Clinger's [1998] model of tail call optimization; a machine with no store can grow without bound due to copying.

machine. At each step, we must strengthen the induction hypothesis by adding a claim about the value of the $\mathcal{OK}$ predicate when applied to the current context.

As a result, most of the steps that can be taken in such a machine correspond either to the reductions of the source semantics or to the mechanical identification of the next expression to be reduced. The first group of reductions in Figure 1 contains those that refocus the evaluation on subexpressions and correspondingly extend the continuation. The second, complementary group contains those that fire when a value shows up as the control string, and these correspond both to changes of focus in the control string and to actual reductions. Finally, a machine with a store must also model garbage collection, if its configurations are to be used in space computations. The final reduction therefore provides garbage collection.

The new Eval_x function is abstracted over a transition relation and an empty context. Applying this to $\mapsto_{\mathsf{fg}}$ and $\mathrm{empty}_{\mathsf{fg}}$ yields the evaluation function $\mathrm{Eval}_{\mathsf{fg}}$.

In order to ensure that Eval and $\mathrm{Eval}_{\mathsf{fg}}$ are indeed the same function, the Eval_x function must employ "load" and "unload" functions. The "load" function, $\mathcal{L}$, coerces the target program to a valid machine configuration. The "unload" function, $\mathcal{U}$, recursively substitutes values bound in the environment for the variables that represent them.

Definition (*Eval$_x$*).

$$\mathrm{Eval}_x(C, \ldots) = \mathcal{U}(V, \sigma) \text{ if } \mathcal{L}_x(C, \ldots) \overset{*}{\mapsto}_x \langle V, \sigma \rangle$$

where

$$\mathcal{L}_x(\langle \lambda_f x.M_{u0}, R_0 \rangle, \ldots) = \langle (\mathcal{A}\langle R_0, [\![\lambda_f x.M_{u0}]\!]\rangle \ \ldots), \emptyset, \emptyset, \mathrm{empty}_x \rangle$$

and

$$\mathcal{U}(\langle \mathrm{closure} : M, \{\langle x_1, \alpha_1 \rangle, \ldots, \langle x_n, \alpha_n \rangle\}\rangle, \sigma) = [\mathcal{U}(\sigma(\alpha_1))/x_1] \ldots [\mathcal{U}(\sigma(\alpha_n))/x_n]M$$

THEOREM (MACHINE FIDELITY). *For all* $(C, \ldots)$,

$$\mathrm{Eval}_{\mathsf{fg}}(C, \ldots) = V \textit{ iff } \mathrm{Eval}(C, \ldots) = V$$

The proof proceeds by induction on the length of a reduction sequence.

4.2 The fg Machine Is Not TCO

To see that this implementation of the λ_{sec} language is not TCO, we show the reduction sequence in the fg machine for the program from Section 3, and validate that the space taken by the configuration is growing without bound.

$$\begin{aligned}
\mathrm{UserClo} &\triangleq \langle \mathrm{closure} : \lambda_{user} sys.\mathcal{A}\langle R_{\mathsf{A}}, [\![\mathrm{UserFn}]\!]\rangle, \emptyset \rangle \\
\mathrm{SystemClo} &\triangleq \langle \mathrm{closure} : \lambda_{sys} user.\mathcal{A}\langle R_{\mathsf{S}}, [\![\mathrm{SystemFn}]\!]\rangle, \emptyset \rangle \\
\rho_0 &\triangleq [sys \mapsto \alpha, user \mapsto \beta] \\
\sigma_0 &\triangleq [\alpha \mapsto \mathrm{SystemClo}, \beta \mapsto \mathrm{UserClo}]
\end{aligned}$$

$$\langle \mathcal{A}\langle R_{\mathsf{A}}, [\![\text{UserFn}]\!]\rangle\ \mathcal{A}\langle R_{\mathsf{S}}, [\![\text{SystemFn}]\!]\rangle, \emptyset, \emptyset, \langle\rangle\rangle \qquad \text{(0 frames)}$$
$$\mapsto_{\mathsf{fg}} \langle \mathcal{A}\langle R_{\mathsf{A}}, [\![\text{UserFn}]\!]\rangle, \emptyset, \emptyset, \langle \text{push} : \mathcal{A}\langle R_{\mathsf{S}}, [\![\text{SystemFn}]\!]\rangle, \emptyset, \langle\rangle\rangle\rangle$$
$$\mapsto_{\mathsf{fg}} \langle \text{UserClo}, \emptyset, \emptyset, \langle \text{push} : \mathcal{A}\langle R_{\mathsf{S}}, [\![\text{SystemFn}]\!]\rangle, \emptyset, \langle\rangle\rangle\rangle$$
$$\mapsto_{\mathsf{fg}} \langle \mathcal{A}\langle R_{\mathsf{S}}, [\![\text{SystemFn}]\!]\rangle, \emptyset, \emptyset, \langle \text{call} : \text{UserClo}, \langle\rangle\rangle\rangle$$
$$\mapsto_{\mathsf{fg}} \langle \text{SystemClo}, \emptyset, \emptyset, \langle \text{call} : \text{UserClo}, \langle\rangle\rangle\rangle$$
$$\mapsto_{\mathsf{fg}} \langle R_{\mathsf{A}}[\mathit{sys\ user}], \rho_0, \sigma_0, \langle\rangle\rangle$$
$$\mapsto_{\mathsf{fg}} \langle \mathit{sys\ user}, \rho_0, \sigma_0, \langle \text{frame} : R_{\mathsf{A}}, \langle\rangle\rangle\rangle \qquad \text{(1 frame)}$$
$$\mapsto_{\mathsf{fg}} \langle \mathit{sys}, \rho_0, \sigma_0, \langle \text{push} : \mathit{user}, \rho_0, \langle \text{frame} : R_{\mathsf{A}}, \langle\rangle\rangle\rangle\rangle$$
$$\mapsto_{\mathsf{fg}} \langle \text{SystemClo}, \rho_0, \sigma_0, \langle \text{push} : \mathit{user}, \rho_0, \langle \text{frame} : R_{\mathsf{A}}, \langle\rangle\rangle\rangle\rangle$$
$$\mapsto_{\mathsf{fg}} \langle \mathit{user}, \rho_0, \sigma_0, \langle \text{call} : \text{SystemClo}, \langle \text{frame} : R_{\mathsf{A}}, \langle\rangle\rangle\rangle\rangle$$
$$\mapsto_{\mathsf{fg}} \langle \text{UserClo}, \rho_0, \sigma_0, \langle \text{call} : \text{SystemClo}, \langle \text{frame} : R_{\mathsf{A}}, \langle\rangle\rangle\rangle\rangle$$
$$\overset{2}{\mapsto}_{\mathsf{fg}} \langle R_{\mathsf{S}}[\mathit{user\ sys}], \rho_0, \sigma_0, \langle \text{frame} : R_{\mathsf{A}}, \langle\rangle\rangle\rangle$$
$$\mapsto_{\mathsf{fg}} \langle \mathit{user\ sys}, \rho_0, \sigma_0, \langle \text{frame} : R_{\mathsf{S}}, \langle \text{frame} : R_{\mathsf{A}}, \langle\rangle\rangle\rangle\rangle \qquad \text{(2 frames)}$$
$$\mapsto_{\mathsf{fg}} \langle \mathit{user}, \rho_0, \sigma_0, \langle \text{push} : \mathit{sys}, \rho_0, \langle \text{frame} : R_{\mathsf{S}}, \langle \text{frame} : R_{\mathsf{A}}, \langle\rangle\rangle\rangle\rangle\rangle$$
$$\mapsto_{\mathsf{fg}} \langle \text{UserClo}, \rho_0, \sigma_0, \langle \text{push} : \mathit{sys}, \rho_0, \langle \text{frame} : R_{\mathsf{S}}, \langle \text{frame} : R_{\mathsf{A}}, \langle\rangle\rangle\rangle\rangle\rangle$$
$$\mapsto_{\mathsf{fg}} \langle \mathit{sys}, \rho_0, \sigma_0, \langle \text{call} : \text{UserClo}, \langle \text{frame} : R_{\mathsf{S}}, \langle \text{frame} : R_{\mathsf{A}}, \langle\rangle\rangle\rangle\rangle\rangle$$
$$\mapsto_{\mathsf{fg}} \langle \text{SystemClo}, \rho_0, \sigma_0, \langle \text{call} : \text{UserClo}, \langle \text{frame} : R_{\mathsf{S}}, \langle \text{frame} : R_{\mathsf{A}}, \langle\rangle\rangle\rangle\rangle\rangle$$
$$\overset{7}{\mapsto}_{\mathsf{fg}} \langle \text{UserClo}, \rho_0, \sigma_0, \qquad \text{(3 frames)}$$
$$\quad \langle \text{call} : \text{SystemClo}, \langle \text{frame} : R_{\mathsf{A}}, \langle \text{frame} : R_{\mathsf{S}}, \langle \text{frame} : R_{\mathsf{A}}, \langle\rangle\rangle\rangle\rangle\rangle\rangle$$
$$\overset{7}{\mapsto}_{\mathsf{fg}} \langle \text{SystemClo}, \rho_0, \sigma_0, \qquad \text{(4 frames)}$$
$$\quad \langle \text{call} : \text{UserClo}, \langle \text{frame} : R_{\mathsf{S}}, \langle \text{frame} : R_{\mathsf{A}}, \langle \text{frame} : R_{\mathsf{S}}, \langle \text{frame} : R_{\mathsf{A}}, \langle\rangle\rangle\rangle\rangle\rangle\rangle\rangle$$
$$\ldots$$

5. AN ALTERNATIVE IMPLEMENTATION

5.1 How Security Inspections Really Work

A close look at λ_{sec} shows that frame ($\ell[\bullet]$) and grant contexts affect the computation only when they are observed by a test expression. That is, a program with no test expressions may be simplified by removing all frame and grant expressions without changing its meaning. Furthermore, the observations possible with the test expression are limited by the $\mathcal{OK}$ function.

In particular, any sequence of frame and grant expressions may be collapsed into a canonical table that provides a partial map from the set of permissions to one of two conditions: "no," indicating that the permission is not granted by the sequence, and "grant," indicating that the permission is granted (and legally so) by some grant frame in the sequence.

To derive update rules for this table, we consider evaluation of the $\mathcal{OK}$ function as the recognition of a context-free grammar over the alphabet of frame and grant expressions. We start by simplifying the model to one with a single permission. Then each frame is either empty or contains the desired permission. Likewise, there is only one possible grant. All other continuation frames are irrelevant. So a full evaluation context can be seen as an arbitrary string in the alphabet $\Sigma = \{y, n, g\}$, where y and n represent frames that contain or are missing the given permission, and g represents a grant. Assume the ordering of the letters in the word places the outermost frames at the left end of the string.

With the grammar in place, the $\mathcal{OK}_{\mathsf{fg}}$ predicate can easily be interpreted as a finite-state machine that recognizes the regular expression $\Sigma^{*}gy^{*}$, that is, a string ending with a grant followed by any number of y's. The resulting FSA has just two states, one accepting and one nonaccepting. A g always transitions to the accepting state, and a n always transitions to the nonaccepting state. A y causes a (trivial) transition to the current state.

This last observation leads us to a further simplification of the grammar. Since the presence of the character y does not affect the decision of the FSA, we may ignore the continuation frames that generate them, and consider only the grant frames and those security frames that do not include the desired permission. The regular expression indicating the success of $\mathcal{OK}_{\mathsf{fg}}$ becomes simply $\Sigma^{*}g$.

This simplification leads to an insight about the security model of λ_{sec}. In the automaton that λ_{sec} induces, the y may be ignored. In the security model, then, callers with a given permission do not affect the result of a check for that permission. Rather, it is the callers *without* that permission that might change its status, and grants of that permission. This suggests that what the security model really tracks is the *absence* of certain permissions. At runtime, then, it is the complement of the permissions attributed to a caller that matters.

Applying the simplified grammar to our reduction semantics allows us to collapse uninterrupted sequences of frame and grant expressions that occur in the evaluation context. A substring ending in a g results in an accepting state, a substring ending in an n results in a nonaccepting state, and the empty substring does not alter the decision. To extend this to the whole language, we must expand our single-permission state to a full table of permissions.

5.2 The cm Machine

In the cm (continuation-marks) machine, each continuation frame contains a table of permissions, called a *mark*. The evaluation steps for frame and grant expressions update the table in the enclosing continuation, rather than increasing the length of the continuation itself. The $\mathcal{OK}_{\mathsf{cm}}$ predicate now inspects these marks, rather than the frame and grant elements of the continuation. Otherwise, the cm machine is the same as the fg machine.

Figure 2 shows the definition of the cm machine. Note that the framing operation takes the complement of the set R, in accordance with the insight of the prior section. Also, the mark mappings are extended pointwise across sets of permissions; that is, $m[R \rightarrow c](p) = c$ if $p \in R$, and $m(p)$ otherwise.

The $\mathrm{Eval}_{\mathsf{cm}}$ function is another instance of Eval_{x}. That is, $\mathrm{Eval}_{\mathsf{cm}}$ is the same as $\mathrm{Eval}_{\mathsf{fg}}$, except that it uses $\mapsto_{\mathsf{cm}}$ as its transition function and $\mathrm{empty}_{\mathsf{cm}}$ as its empty continuation.

The two machines produce the same results.

THEOREM (MACHINE EQUIVALENCE). *For all* $(C, \ldots)$,

$$\mathrm{Eval}_{\mathsf{fg}}(C, \ldots) = V \textit{ iff } \mathrm{Eval}_{\mathsf{cm}}(C, \ldots) = V$$

THE CM MACHINE

$$\begin{array}{rl}
m \in \text{marks} & = \mathcal{P} \rightarrow_f \{\text{grant}, \text{no}\} \\
\text{Configurations} & = \langle M, \rho, \sigma, \kappa \rangle \mid \langle V, \rho, \sigma, \kappa \rangle \mid \langle V, \sigma \rangle \mid \mathsf{fail} \\
\text{Final Configurations} & = \langle V, \sigma \rangle \mid \mathsf{fail} \\
\kappa \in \text{continuations} & = \langle \text{empty} : m \rangle \mid \langle \text{push} : M, \rho, \kappa, m \rangle \mid \langle \text{call} : V, \kappa, m \rangle \\
V \in \text{Values} & = \langle \text{closure} : M, \rho \rangle \\
\rho \in \text{Environments} & = \text{Identifiers} \rightarrow_f \text{Locations} \\
\alpha, \beta & \in \text{Locations} \\
\sigma \in \text{Stores} & = \text{Locations} \rightarrow_f \text{Values} \\
\text{empty}_{\mathsf{cm}} & = \langle \text{empty} : \emptyset \rangle
\end{array}$$

$$\begin{array}{rll}
\langle \lambda_f x.M, \rho, \sigma, \kappa \rangle & \longmapsto_{\mathsf{cm}} & \langle \langle \text{closure} : \lambda_f x.M, \rho \rangle, \rho, \sigma, \kappa \rangle \\
\langle x, \rho, \sigma, \kappa \rangle & \longmapsto_{\mathsf{cm}} & \langle \sigma(\rho(x)), \rho, \sigma, \kappa \rangle \\
\langle M\ N, \rho, \sigma, \kappa \rangle & \longmapsto_{\mathsf{cm}} & \langle M, \rho, \sigma, \langle \text{push} : N, \rho, \kappa, \emptyset \rangle \rangle \\
\langle R[M], \rho, \sigma, \kappa \rangle & \longmapsto_{\mathsf{cm}} & \langle M, \rho, \sigma, \kappa[\overline{R} \mapsto \text{no}] \rangle \\
\langle \mathsf{grant}\ R\ \mathsf{in}\ M, \rho, \sigma, \kappa \rangle & \longmapsto_{\mathsf{cm}} & \langle M, \rho, \sigma, \kappa[R \mapsto \text{grant}] \rangle \\
\langle \mathsf{test}\ R\ \mathsf{then}\ M\ \mathsf{else}\ N, \rho, \sigma, \kappa \rangle & \longmapsto_{\mathsf{cm}} & \begin{cases} \langle M, \rho, \sigma, \kappa \rangle \text{ if } \mathcal{OK}_{\mathsf{cm}} \langle R, [\![\kappa]\!] \rangle \\ \langle N, \rho, \sigma, \kappa \rangle \text{ otherwise} \end{cases} \\
\langle \mathsf{fail}, \rho, \sigma, \kappa \rangle & \longmapsto_{\mathsf{cm}} & \mathsf{fail} \\
 & & \\
\langle V, \rho, \sigma, \langle \text{empty} : m \rangle \rangle & \longmapsto_{\mathsf{cm}} & \langle V, \sigma \rangle \\
\langle V, \rho, \sigma, \langle \text{push} : M, \rho', \kappa, m \rangle \rangle & \longmapsto_{\mathsf{cm}} & \langle M, \rho', \sigma, \langle \text{call} : V, \kappa, \emptyset \rangle \rangle \\
\langle V, \rho, \sigma, \langle \text{call} : V', \kappa, m \rangle \rangle & \longmapsto_{\mathsf{cm}} & \langle M, \rho'[f \mapsto \beta][x \mapsto \alpha], \sigma[\alpha \mapsto V][\beta \mapsto V'], \kappa \rangle \\
 & & \text{if } V' = \langle \text{closure} : \lambda_f x.M, \rho' \rangle \text{ and } \alpha, \beta \notin \text{dom}(\sigma) \\
 & & \\
\langle V, \rho, \sigma[\beta, \ldots \mapsto V, \ldots], \kappa \rangle & \longmapsto_{\mathsf{cm}} & \langle V, \rho, \sigma, \kappa \rangle \\
 & & \text{if } \{\beta, \ldots\} \text{ is nonempty and} \\
 & & \beta, \ldots \text{ do not occur in } V, \rho, \sigma, \text{ or } \kappa
\end{array}$$

where

$$\langle \ldots, m \rangle [R \mapsto c] = \langle \ldots, m[R \mapsto c] \rangle \text{ (pointwise extension)}$$

and

$$\begin{array}{rl}
\mathcal{OK}_{\mathsf{cm}} \langle \emptyset, [\![\kappa]\!] \rangle & \\
\mathcal{OK}_{\mathsf{cm}} \langle R, [\![\langle \text{empty} : m \rangle]\!] \rangle & \text{iff } (R \cap m^{-1}(\text{no}) = \emptyset) \\
\left.\begin{array}{r} \mathcal{OK}_{\mathsf{cm}} \langle R, [\![\langle \text{push} : M, \rho, \kappa, m \rangle]\!] \rangle \\ \mathcal{OK}_{\mathsf{cm}} \langle R, [\![\langle \text{call} : V, \kappa, m \rangle]\!] \rangle \end{array}\right\} & \text{iff } (R \cap m^{-1}(\text{no}) = \emptyset) \text{ and } \mathcal{OK}_{\mathsf{cm}} \langle R - m^{-1}(\text{grant}), [\![\kappa]\!] \rangle
\end{array}$$

Fig. 2. The CM machine.

To prove this theorem, we must show that if the fg machine terminates, the cm machine terminates with the same value, and that if the fg machine does not terminate in a final state, then the cm machine also fails to terminate.

For the purposes of the proof, we assume that no garbage collection steps are taken, because garbage collection cannot affect the result of the evaluation.

LEMMA (NO GARBAGE COLLECTION). *For every evaluation sequence in either the* fg *or* cm *machine, removing every garbage-collection step produces another legal sequence, and no divergent computation is made finite by such a removal.*

To compare the machines, we introduce the function $\mathcal{T}$.

$\mathcal{T} : C_{\mathsf{fg}} \to C_{\mathsf{cm}}$

$$\begin{aligned}
\mathcal{T}\langle M, \rho, \sigma, \kappa\rangle &= \langle M, \rho, \sigma, \mathcal{T}(\kappa)\rangle\\
\mathcal{T}\langle V, \rho, \sigma, \kappa\rangle &= \langle V, \rho, \sigma, \mathcal{T}(\kappa)\rangle\\
\mathcal{T}\langle V, \sigma\rangle &= \langle V, \sigma\rangle\\
\mathcal{T}(\mathsf{fail}) &= \mathsf{fail}\\
\mathcal{T}\langle\rangle &= \langle \text{empty} : \emptyset\rangle\\
\mathcal{T}\langle \text{push} : M, \rho, \kappa\rangle &= \langle \text{push} : M, \rho, \mathcal{T}(\kappa), \emptyset\rangle\\
\mathcal{T}\langle \text{call} : V, \kappa\rangle &= \langle \text{call} : V, \mathcal{T}(\kappa), \emptyset\rangle\\
\mathcal{T}\langle \text{frame} : R, \kappa\rangle &= \mathcal{T}(\kappa)[\overline{R} \mapsto \text{no}]\\
\mathcal{T}\langle \text{grant} : R, \kappa\rangle &= \mathcal{T}(\kappa)[R \mapsto \text{grant}]
\end{aligned}$$

The function $\mathcal{T}$ maps configurations of the fg machine to configurations of the cm machine. A step in the fg machine corresponds to either no steps or one step in the cm machine.

LEMMA (SIMULATION). *Given a configuration C_{cm}, with $C_{\mathsf{cm}} = \mathcal{T}(C_{\mathsf{fg}})$, one of the following holds:*

(1) *C_{fg} is either* fail *or $\langle V, \sigma\rangle$.*
(2) *C_{fg} and C_{cm} are both stuck.*
(3) *$C_{\mathsf{fg}} \mapsto_{\mathsf{fg}} C'_{\mathsf{fg}}$ and $\mathcal{T}(C'_{\mathsf{fg}}) = C_{\mathsf{cm}}$.*
(4) *$C_{\mathsf{fg}} \mapsto_{\mathsf{fg}} C'_{\mathsf{fg}}$ and $C_{\mathsf{cm}} \mapsto_{\mathsf{cm}} \mathcal{T}(C'_{\mathsf{fg}})$.*

The proof is a case analysis on the four cases and the configurations of the machine. The fg machine takes extra steps only when popping frame and grant continuations after reducing their arguments to values.

The cm machine can always represent a sequence of frame and grant expressions with a single mark. The sequence of steps below illustrates this for the divergent mutually recursive computation shown in Section 3.

$$R_{\mathsf{S}} \stackrel{\triangle}{=} \{b, c\}$$
$$R_{\mathsf{A}} \stackrel{\triangle}{=} \{a, b\}$$

$$\begin{aligned}
&\langle \mathcal{A}\langle R_{\mathsf{A}}, [\![\text{UserFn}]\!]\rangle\ \mathcal{A}\langle R_{\mathsf{S}}, [\![\text{SystemFn}]\!]\rangle, \emptyset, \emptyset, \langle \text{empty} : \emptyset\rangle\rangle\\
\mapsto_{\mathsf{cm}}\ &\langle \mathcal{A}\langle R_{\mathsf{A}}, [\![\text{UserFn}]\!]\rangle, \emptyset, \emptyset, \langle \text{push} : \mathcal{A}\langle R_{\mathsf{S}}, [\![\text{SystemFn}]\!]\rangle, \emptyset, \langle \text{empty} : \emptyset\rangle, \emptyset\rangle\rangle\\
\mapsto_{\mathsf{cm}}\ &\langle \text{UserClo}, \emptyset, \emptyset, \langle \text{push} : \mathcal{A}\langle R_{\mathsf{S}}, [\![\text{SystemFn}]\!]\rangle, \emptyset, \langle \text{empty} : \emptyset\rangle, \emptyset\rangle\rangle\\
\mapsto_{\mathsf{cm}}\ &\langle \mathcal{A}\langle R_{\mathsf{S}}, [\![\text{SystemFn}]\!]\rangle, \emptyset, \emptyset, \langle \text{call} : \text{UserClo}, \langle \text{empty} : \emptyset\rangle, \emptyset\rangle\rangle\\
\mapsto_{\mathsf{cm}}\ &\langle \text{SystemClo}, \emptyset, \emptyset, \langle \text{call} : \text{UserClo}, \langle \text{empty} : \emptyset\rangle, \emptyset\rangle\rangle\\
\mapsto_{\mathsf{cm}}\ &\langle R_{\mathsf{A}}[\mathit{sys\ user}], \rho_0, \sigma_0, \langle \text{empty} : \emptyset\rangle\rangle\\
\mapsto_{\mathsf{cm}}\ &\langle \mathit{sys\ user}, \rho_0, \sigma_0, \langle \text{empty} : [\{c\} \mapsto \text{no}]\rangle\rangle\\
\mapsto_{\mathsf{cm}}\ &\langle \mathit{sys}, \rho_0, \sigma_0, \langle \text{push} : \mathit{user}, \rho_0, \langle \text{empty} : [\{c\} \mapsto \text{no}]\rangle\rangle\rangle\\
\mapsto_{\mathsf{cm}}\ &\langle \text{SystemClo}, \rho_0, \sigma_0, \langle \text{push} : \mathit{user}, \rho_0, \langle \text{empty} : [\{c\} \mapsto \text{no}]\rangle, \emptyset\rangle\rangle\\
\mapsto_{\mathsf{cm}}\ &\langle \mathit{user}, \rho_0, \sigma_0, \langle \text{call} : \text{SystemClo}, \langle \text{empty} : [\{c\} \mapsto \text{no}]\rangle, \emptyset\rangle\rangle\\
\mapsto_{\mathsf{cm}}\ &\langle \text{UserClo}, \rho_0, \sigma_0, \langle \text{call} : \text{SystemClo}, \langle \text{empty} : [\{c\} \mapsto \text{no}]\rangle, \emptyset\rangle\rangle\\
\overset{2}{\mapsto}_{\mathsf{cm}}\ &\langle R_{\mathsf{S}}[\mathit{user\ sys}], \rho_0, \sigma_0, \langle \text{empty} : [\{c\} \mapsto \text{no}]\rangle\rangle
\end{aligned}$$

$$
\begin{array}{l}
\mapsto_{\mathsf{cm}} \langle \mathit{user\ sys}, \rho_0, \sigma_0, \langle \text{empty} : [\{a, c\} \mapsto \text{no}] \rangle\rangle \\
\mapsto_{\mathsf{cm}} \langle \mathit{user}, \rho_0, \sigma_0, \langle \text{push} : \mathit{sys}, \rho_0, \langle \text{empty} : [\{a, c\} \mapsto \text{no}] \rangle\rangle\rangle \\
\mapsto_{\mathsf{cm}} \langle \text{UserClo}, \rho_0, \sigma_0, \langle \text{push} : \mathit{sys}, \rho_0, \langle \text{empty} : [\{a, c\} \mapsto \text{no}] \rangle, \emptyset\rangle\rangle \\
\mapsto_{\mathsf{cm}} \langle \mathit{sys}, \rho_0, \sigma_0, \langle \text{call} : \text{UserClo}, \langle \text{empty} : [\{a, c\} \mapsto \text{no}] \rangle, \emptyset\rangle\rangle \\
\mapsto_{\mathsf{cm}} \langle \text{SystemClo}, \rho_0, \sigma_0, \langle \text{call} : \text{UserClo}, \langle \text{empty} : [\{a, c\} \mapsto \text{no}] \rangle, \emptyset\rangle\rangle \\
\overset{7}{\mapsto}_{\mathsf{cm}} \langle \text{UserClo}, \rho_0, \sigma_0, \langle \text{call} : \text{SystemClo}, \langle \text{empty} : [\{a, c\} \mapsto \text{no}] \rangle, \emptyset\rangle\rangle \\
\overset{7}{\mapsto}_{\mathsf{cm}} \langle \text{SystemClo}, \rho_0, \sigma_0, \langle \text{call} : \text{UserClo}, \langle \text{empty} : [\{a, c\} \mapsto \text{no}] \rangle, \emptyset\rangle\rangle \\
\ldots
\end{array}
$$

6. SPACE CONSUMPTION

In "Proper Tail Recursion and Space Efficiency," Clinger [1998] described a framework that characterizes the memory behavior of a language implementation as a mapping from programs to the maximum memory that the implementation consumes while evaluating that program. He demonstrated the difference between various named classes of implementation ("tail-recursive," "safe-for-space," etc.), and defined asymptotic space complexity classes for each, based on abstract machine definitions.

In order to apply Clinger's [1998] analytic framework of tail recursion to the fg and cm machines, we must define a memory measure that maps a machine configuration to a real number. The measure for the fg machine is straightforward.

$$
\begin{array}{rl}
\text{space}(\mathsf{fail}) & = 1 \\
\text{space}(\langle V, \sigma\rangle) & = \text{space}(V) + \text{space}(\sigma) \\
\text{space}(\langle M, \rho, \sigma, \kappa\rangle) & = |\text{dom}(\rho)| + \text{space}(\kappa) + \text{space}(\sigma) \\
\text{space}(\langle V, \rho, \sigma, \kappa\rangle) & = \text{space}(V) + |\text{dom}(\rho)| + \text{space}(\kappa) + \\
 & \quad \text{space}(\sigma) \\
 & \\
\text{space}(\langle \text{closure} : \lambda_f x.M, \rho\rangle) & = 1 + |\text{dom}(\rho)| \\
 & \\
\text{space}(\sigma) & = \sum_{\alpha \in \text{dom}(\sigma)} 1 + \text{space}(\sigma(\alpha)) \\
 & \\
\text{space}(\langle\rangle) & = 1 \\
\text{space}(\langle \text{push} : M, \rho, \kappa\rangle) & = 1 + |\text{dom}(\rho)| + \text{space}(\kappa) \\
\text{space}(\langle \text{call} : V, \kappa\rangle) & = 1 + \text{space}(V) + \text{space}(\kappa) \\
\text{space}(\langle \text{frame} : R, \kappa\rangle) & = 1 + |R| + \text{space}(\kappa) \\
\text{space}(\langle \text{grant} : R, \kappa\rangle) & = 1 + |R| + \text{space}(\kappa)
\end{array}
$$

To accommodate the cm machine, we extend this function with rules for the size of a mark, and for the size of continuations that contain a mark.

$$
\begin{array}{rl}
\text{space}(\langle \text{empty} : m\rangle) & = 1 + \text{space}(m) \\
\text{space}(\langle \text{push} : M, \rho, \kappa, m\rangle) & = 1 + \text{space}(\rho) + \text{space}(\kappa) + |\text{dom}(m)| \\
\text{space}(\langle \text{call} : V, \kappa, m\rangle) & = 1 + \text{space}(V) + \text{space}(\kappa) + |\text{dom}(m)|
\end{array}
$$

The space functions $\mathcal{S}_{\mathsf{fg}}$ and $\mathcal{S}_{\mathsf{cm}}$ are defined as the maximum amount of memory consumed during the evaluation of a program. In order to ensure that unreachable store values do not affect the space function, Clinger [1998] defined a "space-efficient" computation as a sequence of steps where the garbage-collection rule is applied as often as possible.

Definition (*Space-Efficient Computations*). A space-efficient computation in an implementation x is a finite or countably infinite sequence of configurations $\{C_i\}$ such that

—If C_i and C_{i+1} are in the sequence, then $C_i \mapsto_x C_{i+1}$.
—If the sequence is finite, then it ends with a final configuration.
—If the garbage collection rule is applicable to C_i, then $C_i \mapsto_x C_{i+1}$ by the garbage collection rule.

Definition (*Supremum*). If $R \subseteq \Re$, then the supremum $\mathrm{Sup}(R)$ is the least upper bound of R, or ∞ if no such bound exists.

Definition (*Space Consumption S_x*). The space consumption function of an implementation x is $S_x : \mathsf{Program} \to \mathbb{R} \cup \{\infty\}$ defined by

$$\begin{aligned} S_x(P) = |P| + &\\ &\sup\{\sup\{\mathrm{space}(\{C_i\})\}| \\ &\quad \{C_i\} \text{ is a space-efficient} \\ &\quad \text{computation in } x\text{, with} \\ &\quad C_0 = \mathcal{L}_x(P)\} \end{aligned}$$

where $|P|$ is the number of nodes in the abstract syntax tree of P.

Note that the outer "supremum" accommodates the possibility of an implementation that is observationally nondeterministic. This definition is therefore sufficient but overly general.

Following Clinger [1998], we extend the notion of a space function to one of asymptotic space complexity.

Definition (*Asymptotic Complexity, $O(f)$*). If A is any set, and $f : A \to \mathbb{R} \cup \{\infty\}$, then the asymptotic (upper bound) complexity class of f is $O(f)$, which is defined as the set of all functions $g : A \to \mathbb{R} \cup \{\infty\}$ for which there exist real constants c_1 and c_0 such that $c_1 > 0$ and

$$\forall a \in A . g(a) \leq c_1 f(a) + c_0$$

We can now prove that the asymptotic space consumption of the fg machine is strictly greater than that of the cm machine. Put differently, the class of implementations for λ_{sec} in $O(S_{\mathsf{cm}})$ is strictly smaller that those in $O(S_{\mathsf{fg}})$.

THEOREM (SPACE COMPARISON).

$$O(S_{\mathsf{cm}}) \subsetneq O(S_{\mathsf{fg}})$$

Proof Sketch. The proof has two parts. First, we must show that every function in $O(S_{\mathsf{cm}})$ is also in $O(S_{\mathsf{fg}})$. Second, we must show there is a function in $O(S_{\mathsf{fg}})$ that is not in $O(S_{\mathsf{cm}})$.

Since the set $O(S_{\mathsf{cm}})$ takes S_{cm} as its asymptotic upper bound, it suffices for the first half of the proof to show that S_{cm} is in the set $O(S_{\mathsf{fg}})$. That is, for all programs, the maximum space taken while evaluating the program in the cm machine is less than or equal to some constant times the space that the fg machine takes. For simplicity, we shall choose a loose upper bound, taking as our constant the size of the set of permissions $\mathcal{P}$.

The translation $\mathcal{T}$ removes frame and grant continuation frames and introduces marks on each remaining frame, leaving all else untouched. We can therefore show that our bound is satisfied by considering the worst case, in which the fg machine contains no frame or grant continuations, and each mark in the cm machine contains an entry for every possible permission. In this case, each frame is increased in size by the size of $\mathcal{P}$. Since each frame in the fg machine is at least of size 1, we may take $|\mathcal{P}|$ as our linear factor to satisfy the bound.

For the second half of the proof, it suffices to show a program whose machine configuration grows without bound in the fg machine and does not in the cm machine. The example given earlier satisfies this requirement. □

To prove that our implementation is tail-recursive by Clinger's [1998] definition, we must extend his language to include the new forms that appear in λ_{sec}. In order to produce the most stringent possible requirement on space consumption, we propose an implementation that includes a security oracle. That is, we compare ourselves to an implementation that consumes no space at all for the maintenance of stack-trace information. Because of the similarity between the fg machine and Clinger's, the easiest way to model this is to consider the space measure obtained by eliminating the size of the mark from the calculation. We therefore define the oracular space measure "space$_o$," which differs from the existing function only in its rules for continuations.

$$\begin{aligned}
\text{space}_o(\langle \text{empty} : m\rangle) &= 1 \\
\text{space}_o(\langle \text{push} : M, \rho, \kappa, m\rangle) &= 1 + \text{space}(\rho) + \text{space}_o(\kappa) \\
\text{space}_o(\langle \text{call} : V, \kappa, m\rangle) &= 1 + \text{space}(V) + \text{space}_o(\kappa)
\end{aligned}$$

Combining this space measure with the existing cm machine, we define the space class S_o of implementations that consume no memory at all for security information. We can now prove that the cm machine is tail-recursive.

Theorem (Tail Recursion).

$$O(S_{\mathsf{cm}}) = O(S_o)$$

Proof Sketch. To show set equality of asymptotic measures, it suffices to show that $S_{\mathsf{cm}} \in O(S_o)$ and that $S_o \in O(S_{\mathsf{cm}})$.

Since these two space classes use the same machine semantics, we may use the identity function to translate between configurations, and the proof of intensional language equality is trivial.

For the first half of the proof, we must show that the non-oracular measure of a configuration's space ($\text{space}_{\text{cm}}(C)$) is at most k times as large as the space taken by the oracular measurement ($\text{space}_o(C)$). As in the previous proof, we choose as our constant the size of the permission set ($|\mathcal{P}|$). The non-oracular space measure increases the size of a continuation frame by at most $|\mathcal{P}|$, and every continuation frame is of size at least 1, so the space taken by a configuration grows by no more than a factor of $|\mathcal{P}|$.

The other direction is much easier, since simple inspection of the rules for the oracular space function reveals that this function is uniformly smaller for a given configuration than the non-oracular space measure. □

7. THE RICHER MODELS OF JAVA AND .NET

The model of Fournet and Gordon [2002] is an abstraction of Java's and .Net's security model. Both Java and .Net associate program text with permissions and perform security checks by "walking the stack." Both Java and .Net, however, feature a richer security model than λ_{sec} does.

One difference is that the set of permissions is not fixed at compile or even at load time. In both Java and .Net, every object that subclasses a `Permission` class becomes a permission in the security system.

Also, these systems may permit the mappings from code to permissions to be mutated at runtime. In Java 1.2, however, the default implementation does not allow this. That is, the mapping from code to permissions is performed at class-loading time, and not subsequently altered [Gong 1999]. We are not aware of such a guarantee in .Net.

These differences, however, do not invalidate the principal invariant that underlies our proposed marriage of security and TCO. Specifically, the security information retained by the system can only be observed by a security-checking system call. This means that the representation of this information (and the implementation of the corresponding security-checking primitive) may be changed, as long as these changes preserve the observable behavior of the system.

In addition, all of the systems we have examined share the trait that the frame expressions—or the corresponding constructs in Java or .Net—may be reordered, up to the boundaries established by a grant (or .Net's "assert").

The first step in applying the lessons of our work to a given system is to formulate an appropriate model. For languages like Java, Java's JVM, or .Net's intermediate language, this model should probably extend λ_{sec} with assignment, objects with subclassing, exceptions, and dynamic loading.

The next step is to formulate the notion of TCO in such a system. This is fairly natural for any model that treats function application as the transfer of control to a given source location with an unchanged continuation.

The final step is to extend the system with mechanisms for maintaining security state, and to demonstrate that these additions do not affect the tail-recursive behavior of the model.

Fundamentally, our approach separates the implementation of function calls from the implementation of security primitives. In other words, it decouples the security mechanism from the memory behavior of stack frames. This differs from the existing implementations, which capitalize on unanticipated observations at the machine level and therefore restrict the set of possible language implementations.

We conjecture that the security-policy implementation that results from such an analysis is likely to have much in common with our CESK machines. That is, the security information will be attached to the stack frame of the parent, rather than the stack frame of the child. Garbage-collection-like strategies for the management of such information could be employed to maintain asymptotic memory behavior while delaying runtime costs until the application of a security check.

8. RELATED WORK

This article was directly inspired by the POPL presentation of a semantics for stack inspection by Fournet and Gordon [2002], and by our earlier research on an algebraic stepper for DrScheme [Clements et al. 2001], where we produced a portable and provably correct algebraic stepper, based on a novel, lightweight stack inspection mechanism. Using a primitive function, a program can place continuation marks on the stack and inquire about existing marks. If a function places two marks on the stack, the runtime environment replaces the first with the second. Hence, the manipulation of continuation marks automatically preserves tail call optimizations. The key difference between our earlier work and this article is that continuation marks for security permissions contain negative rather than positive information. Once we understood this, we could derive the rest of the ideas here in a straightforward manner.

8.1 Security-Passing Style

Another implementation strategy for stack inspection was due to Wallach et al. [1997, 2000]. In security-passing style, each procedure accepts an additional argument that represents the security context accumulated thus far.

Essentially, this implementation derives from the observation that the security information computed on some context does not change while that context remains active. Therefore, an implementation can compute the security information (or some representation thereof) once, at each call site, and pass it along during the computation.

As a tool of semantic definition, we believe that security-passing style succeeds. That is, the given transformation, when combined with a semantics for the underlying target language, does specify a meaning for each of the security primitives. However, we find the semantics of Fournet and Gordon [2002] to be simpler, as they directly interpret the language of security primitives.

As a tool of implementation, we believe security-passing style leaves something to be desired. In particular, the simple overhead of adding an argument to each call is prohibitive. It is certainly the case that a variety of optimizations may be applied to lower the runtime cost, but in this case we contend that our implementation strategy is a more direct route to a similar optimized machine.

8.2 Other Work

Several others [Benton et al. 1998; Schinz and Odersky 2001] have considered the problem of adding tail calls to the JVM, which does support stack inspection. However, none of these specifically addressed stack inspection or security, and all of them made the simplifying assumption that TCO was only possible between procedures of the same component; that is, none of them considered calls between code from distinct security domains.

Karjoth [2000] presented a semantics for access control in Java 2; his model presents rules for the maintenance of access control information, but leaves the rules for the evaluation of the language itself unspecified. Because he included rules for matching "call" and "return" expressions, his system cannot be the foundation for a TCO implementation.

Erlingsson and Schneider [2000] showed how to implement the stack inspection primitives with no support from the runtime system. That is, they used an annotation-based approach to transform a program with security primitives into one without them. However, this work winds up simulating the stack on the side, with two unfortunate consequences: exceptions become much more difficult, and TCO is destroyed.

9. CONCLUSIONS

Our article invalidates the widely held belief among programming language researchers that a global tail call optimization policy is incompatible with stack inspection for security policies. We have developed an alternative implementation of stack inspection; we proved that it preserves the observable behavior of all programs; and we showed that its notion of tail call is consistent with Clinger's [1998] mathematical notion of tail call optimization. It is our belief that translating our ideas into a compiler or a virtual machine imposes no additional cost on the implementation of any other construct. We expect that such an implementation will perform as well or better than a conventional stack inspection implementation.

This article also outlines a technique for the development of a security model in a tail-recursive language. That is, by recasting the security model in the terms of continuations and observations of continuations, an implementor can naturally derive a tail-recursive implementation of his language. Finally, our article offers lessons for language designers about the coexistence of stack inspection and TCO. The natural next step is to apply these lessons to produce implementations of existing languages that support both stack inspection and TCO and that perform well.

APPENDIX: EQUIVALENCE OF FOURNET AND GORDON'S EVALUATOR AND THE ONE PRESENTED IN THIS ARTICLE

The model we use for reductions on λ_{sec} differs from that given by Fournet and Gordon [2002] in three minor ways, as mentioned in Section 2. Of these, the only difference that requires explanation is our substitution of a static rewriting step for their dynamic check on grant expressions. Briefly, this static check is possible because all program code is a part of some component, and is therefore initially annotated with permissions. Inspection of the semantics shows that a frame expression always accompanies each grant expression, and therefore that part of its behavior can be predicted. In particular, the dynamic restriction of a grant's permissions to that of the nearest enclosing frame expression can be performed at the time of annotation.

To argue this equivalence more formally, we model the original semantics with a modified reduction function $\mapsto_o$. The definition of $\mapsto_o$ differs from that of $\mapsto$ only in that the reference to $\mathcal{OK}$ is replaced by a reference to $\mathcal{OK}_o$. This modified predicate dynamically restricts the permissions enabled by a grant to those appearing in its nearest enclosing frame expression, using an auxiliary Static function. The function $\mapsto_o$ therefore models the original semantics of Fournet and Gordon.

Original Permissions Check $\quad \mathcal{OK}_o \subseteq 2^{\mathcal{P}} \times E$

$$\begin{array}{rl}
\mathcal{OK}_o\langle R, [\![\bullet]\!]\rangle & \\
\mathcal{OK}_o\langle R, [\![E[\bullet\ M]]\!]\rangle & \text{iff } \mathcal{OK}_o\langle R, [\![E]\!]\rangle \\
\mathcal{OK}_o\langle R, [\![E[V\ \bullet]]\!]\rangle & \text{iff } \mathcal{OK}_o\langle R, [\![E]\!]\rangle \\
\mathcal{OK}_o\langle R, [\![E[S[\bullet]]]\!]\rangle & \text{iff } R \subseteq S \text{ and } \mathcal{OK}_o\langle R, [\![E]\!]\rangle \\
\mathcal{OK}_o\langle R, [\![E[\mathsf{grant}\ S\ \mathsf{in}\ \bullet]]\!]\rangle & \text{iff } \mathcal{OK}_o\langle R - \text{Static}\langle S, [\![E]\!]\rangle, [\![E]\!]\rangle
\end{array}$$

where

$$\begin{array}{rcl}
\text{Static}\langle R, [\![\bullet]\!]\rangle & = & R \\
\text{Static}\langle R, [\![E[\bullet\ M]]\!]\rangle & = & \text{Static}\langle R, [\![E]\!]\rangle \\
\text{Static}\langle R, [\![E[V\ \bullet]]\!]\rangle & = & \text{Static}\langle R, [\![E]\!]\rangle \\
\text{Static}\langle R, [\![E[S[\bullet]]]\!]\rangle & = & R \cap S \\
\text{Static}\langle R, [\![E[\mathsf{grant}\ S\ \mathsf{in}\ \bullet]]\!]\rangle & = & \text{Static}\langle R, [\![E]\!]\rangle
\end{array}$$

To extend the reduction function $\mapsto_o$ to an evaluation function, we must prefix evaluation with an annotator as before. This annotator, $\mathcal{A}_o$, differs from $\mathcal{A}$ in that it does not restrict the permissions in grant expressions to those attached to the entire component.

Definition (*Eval$_o$*).

$$\text{Eval}_o(C, \ldots) = V \text{ if } (\mathcal{A}_o(C)\ \cdots) \overset{*}{\mapsto}_o V$$

where

$$\begin{array}{rcl}
\mathcal{A}_o\langle R, [\![x]\!]\rangle & = & x \\
\mathcal{A}_o\langle R, [\![\lambda_f x.M]\!]\rangle & = & \lambda_f x.R[\mathcal{A}_o\langle R, [\![M]\!]\rangle] \\
\mathcal{A}_o\langle R, [\![M\ N]\!]\rangle & = & \mathcal{A}_o\langle R, [\![M]\!]\rangle\ \mathcal{A}_o\langle R, [\![N]\!]\rangle \\
\mathcal{A}_o\langle R, [\![\mathsf{grant}\ S\ \mathsf{in}\ M]\!]\rangle & = & \mathsf{grant}\ S\ \mathsf{in}\ \mathcal{A}_o\langle R, [\![M]\!]\rangle \\
\mathcal{A}_o\langle R, [\![\mathsf{test}\ S\ \mathsf{then}\ M\ \mathsf{else}\ N]\!]\rangle & = & \mathsf{test}\ S\ \mathsf{then}\ \mathcal{A}_o\langle R, [\![M]\!]\rangle\ \mathsf{else}\ \mathcal{A}_o\langle R, [\![N]\!]\rangle \\
\mathcal{A}_o\langle R, [\![\mathsf{fail}]\!]\rangle & = & \mathsf{fail}
\end{array}$$

With these definitions in place, we can state the claim that the Eval functions are equal, modulo Fournet and Gordon's [2002] notion of contextual equivalence ($\equiv_o$).

PROPOSITION (STATIC PERMISSIONS CHECK). *For any components* $(A_0, \ldots)$,

$$Eval(A_0, \ldots) = U \text{ iff } Eval_o(A_0, \ldots) = V \text{ where } U \equiv_o V$$

PROOF SKETCH. The proof proceeds in two steps. First, we establish the equivalence (modulo contextual equivalence) of Eval_o (the composition of $\mathcal{A}_o$ and $\mapsto_o^*$) and the composition of $\mathcal{A}$ and $\mapsto_o^*$. Second, we establish the equivalence of this composition and Eval (the composition of $\mathcal{A}$ and $\mapsto^*$) when applied to terms that satisfy a new predicate $\mathcal{B}$, which describes the results of $\mathcal{A}$.

The two steps require three lemmas:

(1) For all programs $(A, \ldots)$, $(\mathcal{A}(A), \ldots) \equiv_o (\mathcal{A}_o(A), \ldots)$.
(2) For all programs $(A, \ldots)$, $\mathcal{B}(\emptyset, [\![(\mathcal{A}(A)\ \cdots)]\!])$.
(3) For all terms M, $\mathcal{B}(\emptyset, [\![M]\!])$ implies that $M \mapsto_o V$ iff $M \mapsto V$.

Lemma 1 implements the first step, Lemma 3 corresponds to the third step, and Lemma 2 provides the glue.

LEMMA 1 PROOF SKETCH. The translations $\mathcal{A}_o$ and $\mathcal{A}$ differ in their treatment of grant; $\mathcal{A}$ restricts the permissions contained in the grant to those that appear in the component's permissions, and $\mathcal{A}$ doesn't. This alteration may be derived from the following equations in Fournet and Gordon's [2002] contextual equivalence theory:

SELECTED EQUATIONS

$$\begin{aligned}
\text{(Frame Frame Appl)}: &\ R_1[R_2[e_1\ e_2]] \equiv_o R_1[R_2[R_1[R_2[e_1]]\ R_1[R_2[e_2]]]] \\
\text{(Frame Frame)}: &\ R_1 \supseteq R_2 \Rightarrow R_1[R_2[e]] \equiv_o R_2[e] \\
\text{(Frame Grant)}: &\ R_1[\mathsf{grant}\ R_2\ \mathsf{in}\ e] \equiv_o R_1[\mathsf{grant}\ R_1 \cap R_2\ \mathsf{in}\ e] \\
\text{(Frame Grant Frame)}: &\ R_1 \supseteq R_2 \Rightarrow R_1[\mathsf{grant}\ R_2\ \mathsf{in}\ R_3[e]] \equiv_o R_1[R_3[\mathsf{grant}\ R_2\ \mathsf{in}\ e]] \\
\text{(Frame Test Then)}: &\ R_1 \supseteq R_2 \Rightarrow R_1[\mathsf{test}\ R_2\ \mathsf{then}\ e_1\ \mathsf{else}\ e_2] \\
&\qquad\qquad \equiv_o \mathsf{test}\ R_2\ \mathsf{then}\ R_1[e_1]\ \mathsf{else}\ R_1[e_2] \\
\text{(Frame Test Else)}: &\ \neg(R_1 \supseteq R_2) \Rightarrow R_1[\mathsf{test}\ R_2\ \mathsf{then}\ e_1\ \mathsf{else}\ e_2] \equiv_o R_1[e_2]
\end{aligned}$$

For a given component $\langle R, M\rangle$, its annotation in Fournet and Gordon's [2002] system is $\mathcal{A}_o\langle R, M\rangle$. Since each top-level component expression M must be a λ-expression, at least one frame expression lies outside any grant that the component contains. Using the contextual equivalence theory, we can propagate frame expressions inward past any syntactic constructions other than abstraction. The only interesting case is grant, where pushing the frame inward requires the application of the Frame-Grant, Frame-Frame, Frame-Grant-Frame, and Frame-Frame equations. Since all abstraction bodies are wrapped in frame expressions with the component's permissions, this calculation leaves each grant expression wrapped with a frame expression. Then, the Frame-Grant rule justifies the intersection of the two sets of permissions. Finally, a reversal of the calculation applied thus far may be used to remove the inserted frame expressions. This leaves us with $\mathcal{A}(R, [\![M]\!])$. By this reasoning, substituting $\mathcal{A}$ for $\mathcal{A}_o$ in the definition of Eval_o yields contextually equivalent results.

To make the second major step of the proof, we introduce the predicate $\mathcal{B}$.

Legal Grants Predicate $\mathcal{B} \subseteq 2^{\mathcal{P}} \times M$

$$\begin{array}{rl}
\mathcal{B}\langle R, [\![x]\!]\rangle & \\
\mathcal{B}\langle R, [\![\lambda_f x.M]\!]\rangle & \text{iff } \mathcal{B}\langle \emptyset, [\![M]\!]\rangle \\
\mathcal{B}\langle R, [\![M\ N]\!]\rangle & \text{iff } \mathcal{B}\langle R, [\![M]\!]\rangle\ \mathcal{B}\langle R, [\![N]\!]\rangle \\
\mathcal{B}\langle R, [\![S[M]]\!]\rangle & \text{iff } \mathcal{B}\langle S, [\![M]\!]\rangle \\
\mathcal{B}\langle R, [\![\mathsf{grant}\ S\ \mathsf{in}\ M]\!]\rangle & \text{iff } S \subseteq R \text{ and } \mathcal{B}\langle R, [\![M]\!]\rangle \\
\mathcal{B}\langle R, [\![\mathsf{test}\ S\ \mathsf{then}\ M\ \mathsf{else}\ N]\!]\rangle & \text{iff } \mathcal{B}\langle R, [\![M]\!]\rangle \text{ and } \mathcal{B}\langle R, [\![N]\!]\rangle \\
\mathcal{B}\langle R, [\![\mathsf{fail}]\!]\rangle &
\end{array}$$

The predicate $\mathcal{B}$ checks two arguments: a set of permissions and an expression. It is satisfied[4] when all grant expressions refer to permissions that appear in the nearest enclosing frame expression, looking no further than the nearest λ boundary.

Lemma 2 Proof Sketch. The predicate $\mathcal{B}$ formulates what the annotator $\mathcal{A}$ enforces; namely, that grant expressions refer only to permissions accorded to their components. The proof proceeds by induction on the size of the program. The natural induction hypothesis states that for any R and M, $\mathcal{B}(R, \mathcal{A}(R, [\![M]\!]))$. That is, the annotation of M with R satisfies $\mathcal{B}$ with permission set R. However, we must strengthen the induction hypothesis for λ-expressions to state that $\mathcal{B}(\emptyset, \mathcal{A}(R, [\![\lambda_f x.M]\!])$. In other words, the annotation of a lambda term satisfies $\mathcal{B}$ with the empty permissions set.

Lemma 3 Proof Sketch. We must now prove that the relations $\mapsto$ and $\mapsto_o$ act identically on terms that satisfy $\mathcal{B}$. First, we show that $\mathcal{OK}$ and $\mathcal{OK}_o$ are equivalent for evaluation contexts formed from expressions that satisfy $\mathcal{B}$. Second, a subject reduction proof shows that satisfaction of $\mathcal{B}$ is preserved by both $\mapsto$ and $\mapsto_o$.

Suppose M satisfies $\mathcal{B}$, and $M = E[N]$. Then for any R, $\mathcal{OK}_o(R, [\![E]\!])$ iff $\mathcal{OK}(R, [\![E]\!])$. The satisfaction of $\mathcal{B}$ guarantees that Static acts as the identity; the permissions attached to a grant are already restricted to those occurring in the nearest enclosing frame.

The subject reduction proof is largely mechanical. The only interesting cases are those in which a frame is removed and those involving a β_v reduction. In each case, the key observation is that λ expressions are self-contained. By this we mean that the value of $\mathcal{B}(R, [\![\lambda_f x.M]\!])$ does not depend on R at all. Therefore, substituting a value (that is, a lambda expression) that satisfies $\mathcal{B}$ for any other expression does not change an expression that satisfies $\mathcal{B}$ into one that doesn't. Furthermore, this argument applies to the bodies of abstractions as well. That is, if a term before a substitution satisfied $\mathcal{B}$ and contained the term $\lambda_f x.M$, we may conclude that $\mathcal{B}(\emptyset, [\![M]\!])$, which implies that for any choice of R, $\mathcal{B}(R, [\![M]\!])$ also holds, and therefore that the substitution of the term M for another term preserves satisfaction. Both of the reductions of interest consist entirely of one

[4]The statement "M satisfies $\mathcal{B}$" is taken to mean that $\mathcal{B}(\emptyset, [\![M]\!])$, or (equivalently) $\langle \emptyset, [\![M]\!]\rangle \in \mathcal{B}$.

or more such substitutions, and must therefore preserve satisfaction of $\mathcal{B}$. This argument applies without modification to both $\mapsto$ and $\mapsto_o$.

With these two pieces in hand, a simple case analysis shows that $\mapsto$ and $\mapsto_o$ behave identically on terms that satisfy $\mathcal{B}$.

Taken together, the three lemmas allow us to conclude that our proposition holds. □

ACKNOWLEDGMENTS

We are grateful to Cedric Fournet, Joe Marshall, and the members of Northeastern's Programming Languages Seminar for their comments on earlier versions of the article, and to Matthew Flatt for the design and implementation of continuation marks.

REFERENCES

BENTON, N., KENNEDY, A., AND RUSSELL, G. 1998. Compiling standard ML to Java bytecodes. In *Proceedings of the ACM SIGPLAN International Conference on Functional Programming*. 129–140.

BOX, D. 2002. *Essential .NET, Volume I: The Common Language Runtime*. Addison-Wesley, Reading, MA.

CLEMENTS, J., FLATT, M., AND FELLEISEN, M. 2001. Modeling an algebraic stepper. In *Proceedings of the European Symposium on Programming*. 320–334.

CLINGER, W. D. 1998. Proper tail recursion and space efficiency. In *Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation*. 174–185.

ERLINGSSON, U. AND SCHNEIDER, F. B. 2000. IRM enforcement of Java stack inspection. In *Proceedings of the IEEE Symposium on Security and Privacy*. 246–255.

FELLEISEN, M. AND FLATT, M. 1989–2002. Programming languages and their calculi. Unpublished manuscript. Available online at `<http://www.ccs.neu.edu/home/matthias/3810-w02/mono.ps.gz>`.

FELLEISEN, M. AND FRIEDMAN, D. P. 1986. Control operators, the SECD-machine, and the λ-calculus. In *Formal Description of Programming Concepts III*, M. Wirsing, Ed. Elsevier Science Publishers B.V., Amsterdam, The Netherlands, 193–217.

FLATT, M. 1995–2002. PLT MzScheme: Language manual. Available online at `<http://www.plt-scheme.org>`.

FOURNET, C. AND GORDON, A. D. 2002. Stack inspection: Theory and variants. In *Proceedings of the Symposium on Principles of Programming Languages*. 307–318.

GAMMA, E., HELM, R., JOHNSON, R., AND VLISSIDES, J. 1995. *Design Patterns*. Addison-Wesley, Reading, MA.

GONG, L. 1999. *Inside Java 2 Platform Security*. Sun Microsystems, Santa Clara, CA.

KARJOTH, G. 2000. An operational semantics of Java 2 access control. In *Proceedings of the Computer Security Foundations Workshop*. 224–232.

KELSEY, R., CLINGER, W. D., AND REES, J. 1998. Revised^5 report on the algorithmic language scheme. *ACM SIGPLAN Not. 33*, 9, 26–76.

MICROSOFT. 2002. Common language runtime SDK documentation. Available online at `http://www.microsoft.com`. Part of .NET SDK documentation.

PLOTKIN, G. D. 1975. Call-by-name, call-by-value and the λ-calculus. *Theoret. Comput. Sci.* 125–159.

POTTIER, F., SKALKA, C., AND SMITH, S. 2001. A systematic approach to static access control. In *Proceedings of the European Symposium on Programming*. 30–45.

SCHINZ, M. AND ODERSKY, M. 2001. Tail call elimination on the Java virtual machine. In *Proceedings of the SIGPLAN BABEL Workshop on Multi-Language Infrastructure and Interoperability*. 155–168.

Skalka, C. and Smith, S. 2000. Static enforcement of security with types. *ACM SIGPLAN Not. 35*, 9, 34–45.

Steele Jr., G. L. 1977. Debunking the 'expensive procedure call' myth. In *Proceedings of the 1977 Annual ACM Conference*. 153–162.

Wallach, D., Balfanz, D., Dean, D., and Felten, E. 1997. Extensible security architectures for Java. In *Proceedings of the 16th Symposium on Operating Systems Principles*. 116–128.

Wallach, D., Felten, E., and Appel, A. 2000. The security architecture formerly known as stack inspection: A security mechanism for language-based systems. *ACM Trans. Softw. Eng. Methodol. 9*, 4 (Oct.), 341–378.

Received February 2003; revised June 2003; accepted October 2003