A Tail-Recursive Semantics for Stack Inspections

John Clements and Matthias Felleisen

Northeastern University
Boston, Massachusetts

Abstract. Security folklore holds that a security mechanism based on stack inspection is incompatible with a global tail call optimization policy. An implementation of such a language may have to 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 paper, 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.

1 Stacks, Security, and Tail Calls

Over the last ten 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 [17]. It starts from the premise that trusted components may authorize potentially insecure actions for the dynamic extent of some 'grant' 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 [13]. 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 disposes of information that is necessary to evaluate the security primitives.

For this reason, Fournet and Gordon [7] model function calls with a non-standard β-reduction. To be more precise, β does not hold as an equation for source terms. Abstraction bodies are wrapped with context-building primitives. Unfortunately, this formalization prohibits a transformation of this semantics into a tail-call optimizing (TCO) implementation. Fournet and Gordon recognize this fact and state that "[S]tack inspection profoundly affects the semantics of all programs. In particular, it invalidates [...] tail call optimizations." [7]

This understanding of the stack inspection protocol also pervades the implementation of existing run-time 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 [12] 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. [18] suggest an alternate security model that accommodates TCO implementations. They add 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. Unfortunately, this transformation is global; it cannot be applied in isolation to a single untrusted component, but requires the rewriting of all procedures in all system libraries. They also fail to provide a formal language semantics that allows a Fournet-Gordon style validation of their claims.

Our security model exploits a novel mechanism for lightweight stack inspection [6]. We demonstrate the equivalence between our model and Fournet & 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, the transformation that adds security annotations to the untrusted code is local.

We proceed as follows. First, we derive a CESK machine from Fournet & Gordon's semantics. Second, we develop a different, but extensionally equivalent CESK machine that uses a variant of Flatt's lightweight stack inspection mechanism [6]. 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 tail-call optimizing CESK machine [4].

The paper 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 equivalent CESK machine from Fournet and Gordon's 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 tail-call optimizing. In the seventh section, we discuss the compatibility of our model of λ_{sec} with Fournet and Gordon's, using their theory of contextual equivalence. The last two sections place our work into context.

2 The λ_{sec} Language

Fournet and Gordon use as their starting point the λ_{sec}-calculus [14,16], a simple model of a programming language with security annotations. They present two languages: a source language, in which programs are written, and a target language, which includes an additional form for security annotations. A trusted

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

annotator performs the translation from the source to the target, annotating each λ-expression 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 fragment 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 label. 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 stack frames must have been annotated with this permission.

The source language (M_s) 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

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

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 result with the source-appropriate permissions. In our case, components are λ-expressions. 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.[2]

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

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

[2] Fournet and Gordon present a semantics in which this check is performed at runtime. Section 7 discusses the differences between the two languages in more detail.

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 & Gordon's semantics to our variant of λ_{sec} mutatis mutandis. Evaluation of programs is specified using a reduction semantics based on evaluation contexts. 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 for instance 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

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

where

$$\begin{aligned}
\mathcal{OK}[\![\emptyset]\!][\![E]\!] &= \mathit{true} \\
\mathcal{OK}[\![R]\!][\![\bullet]\!] &= \mathit{true} \\
\mathcal{OK}[\![R]\!][\![E[\bullet\ M]]\!] &= \mathcal{OK}[\![R]\!][\![E]\!] \\
\mathcal{OK}[\![R]\!][\![E[V\ \bullet]]\!] &= \mathcal{OK}[\![R]\!][\![E]\!] \\
\mathcal{OK}[\![R]\!][\![E[S[\bullet]]]\!] &= R \subseteq S \wedge \mathcal{OK}[\![R]\!][\![E]\!] \\
\mathcal{OK}[\![R]\!][\![E[\mathsf{grant}\ S\ \mathsf{in}\ \bullet]]\!] &= \mathcal{OK}[\![R - S]\!][\![E]\!]
\end{aligned}$$

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 interesting in that they 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 continuation from the inside out, succeeding either when the permissions remaining to check are empty or when the context is exhausted. 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. The stack inspection protocol is, at heart, a lightweight form of continuation manipulation [3].

In Fournet and Gordon's framework, a program consists of a set of components, each one a closed λ-expression with its own set of permissions.

Definition 1 (Components). $A \in \text{Components} = \langle \lambda_f x.M_s, R \rangle$

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 2 (Eval).

$$\text{Eval}(\langle \lambda_f x.M_{u0}, R_0 \rangle \ldots) = V \text{ if } (\mathcal{A}R_0[\![\lambda_f x.M_{u0}]\!] \ \ldots) \overset{*}{\mapsto} V$$

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.

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 [2] and the promotion of traversal strategies such as the interpreter, composite, or visitor patterns [8].

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 1970's, 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 [9]. Typically, such function calls turn into plain jumps, and hence, the translation of a tail-recursive function equals the translation of a 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" [11].

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

$$\begin{aligned}
\text{UserFn} &\triangleq \lambda_{user}\, sys.sys\ user \\
\text{SystemFn} &\triangleq \lambda_{sys}\, user.user\ sys \\
\mathcal{AR}_{\mathsf{A}}[\![\text{UserFn}]\!] &= \lambda_{user}\, sys.R_{\mathsf{A}}[sys\ user] \\
\mathcal{AR}_{\mathsf{S}}[\![\text{SystemFn}]\!] &= \lambda_{sys}\, user.R_{\mathsf{S}}[user\ sys]
\end{aligned}$$

Reduction (w/ Annotations)

$$\begin{aligned}
&\mathcal{AR}_{\mathsf{A}}[\![\text{UserFn}]\!]\ \mathcal{AR}_{\mathsf{S}}[\![\text{SystemFn}]\!] \\
\mapsto\ &R_{\mathsf{A}}[\mathcal{AR}_{\mathsf{S}}[\![\text{SystemFn}]\!]\ \mathcal{AR}_{\mathsf{A}}[\![\text{UserFn}]\!]] \\
\mapsto\ &R_{\mathsf{A}}[R_{\mathsf{S}}[\mathcal{AR}_{\mathsf{A}}[\![\text{UserFn}]\!]\ \mathcal{AR}_{\mathsf{S}}[\![\text{SystemFn}]\!]]] \\
\mapsto\ &R_{\mathsf{A}}[R_{\mathsf{S}}[R_{\mathsf{A}}[\mathcal{AR}_{\mathsf{S}}[\![\text{SystemFn}]\!]\ \mathcal{AR}_{\mathsf{A}}[\![\text{UserFn}]\!]]]] \\
\mapsto\ &R_{\mathsf{A}}[R_{\mathsf{S}}[R_{\mathsf{A}}[R_{\mathsf{S}}[\mathcal{AR}_{\mathsf{A}}[\![\text{UserFn}]\!]\ \mathcal{AR}_{\mathsf{S}}[\![\text{SystemFn}]\!]]]]] \\
&\ldots
\end{aligned}$$

Reduction (w/o Annotations)

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

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 tail-call optimization. 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 illustrates the steps taken by λ_{sec} in evaluating the given program, where the two procedures are annotated with their permissions. In this example, the context quickly grows without bound. 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 which is obtained by evaluating the program's pure source.

As Fournet and Gordon point out in their paper, all is not lost. They introduce an additional reduction into their abstract machine that explicitly removes a frame before performing a call. Unfortunately, as they point out, indiscriminate application of this rule changes the semantics. Thus, they impose strict conditions that the machine must check before it can apply the rule. The rule and its side conditions clarify that an improved compiler can turn *some* tail calls into jumps, but Fournet and Gordon state that many tail calls cannot be optimized.

4 An Abstract Machine for $\lambda_{\mathbf{sec}}$

Following Clinger's work on defining tail-optimized languages via space complexity classes [4], we reformulate the λ_{sec} semantics as a CESK machine [5]. 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 machine has 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 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 [5]. 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 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.

The new Eval function is abstracted over the machine under consideration. In particular, the definition of Eval_x for a machine x depends both on the transition function, $\mapsto_x$, and on the empty context, empty_x.

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

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

THE FG MACHINE

$$
\begin{aligned}
C_{\mathsf{fg}} &= \langle M, \rho, \sigma, \kappa\rangle \mid \langle V, \rho, \sigma, \kappa\rangle \mid \langle V, \sigma\rangle \mid \mathsf{fail}\\
\kappa &= \langle\rangle \mid \langle \mathrm{push} : M, \rho, \kappa\rangle \mid \langle \mathrm{call} : V, \kappa\rangle \mid \langle \mathrm{frame} : R, \kappa\rangle \mid \langle \mathrm{grant} : R, \kappa\rangle\\
V \in \mathrm{Values} &= \langle \mathrm{closure} : M, \rho\rangle\\
\rho &\in \mathrm{Identifiers} \to_f \mathrm{Locations}\\
\alpha, \beta &\in \mathrm{Locations}\\
\sigma &\in \mathrm{Locations} \to_f \mathrm{Values}\\
\mathrm{empty}_{\mathsf{fg}} &= \langle\rangle
\end{aligned}
$$

$$
\begin{aligned}
\langle \lambda_f x.M, \rho, \sigma, \kappa\rangle &\mapsto_{\mathsf{fg}} \langle\langle \mathrm{closure} : \lambda_f x.M, \rho\rangle, \rho, \sigma, \kappa\rangle\\
\langle x, \rho, \sigma, \kappa\rangle &\mapsto_{\mathsf{fg}} \langle \sigma(\rho(x)), \rho, \sigma, \kappa\rangle\\
\langle M\ N, \rho, \sigma, \kappa\rangle &\mapsto_{\mathsf{fg}} \langle M, \rho, \sigma, \langle \mathrm{push} : N, \rho, \kappa\rangle\rangle\\
\langle R[M], \rho, \sigma, \kappa\rangle &\mapsto_{\mathsf{fg}} \langle M, \rho, \sigma, \langle \mathrm{frame} : R, \kappa\rangle\rangle\\
\langle \mathsf{grant}\ R\ \mathsf{in}\ M, \rho, \sigma, \kappa\rangle &\mapsto_{\mathsf{fg}} \langle M, \rho, \sigma, \langle \mathrm{grant} : R, \kappa\rangle\rangle\\
\langle \mathsf{test}\ R\ \mathsf{then}\ M\ \mathsf{else}\ N, \rho, \sigma, \kappa\rangle &\mapsto_{\mathsf{fg}} \begin{cases} \langle M, \rho, \sigma, \kappa\rangle \text{ if } \mathcal{OK}_{\mathsf{fg}}[\![R]\!][\![\kappa]\!] \\ \langle N, \rho, \sigma, \kappa\rangle \text{ otherwise} \end{cases}\\
\langle \mathsf{fail}, \rho, \sigma, \kappa\rangle &\mapsto_{\mathsf{fg}} \mathsf{fail}
\end{aligned}
$$

$$
\begin{aligned}
\langle V, \rho, \sigma, \langle\rangle\rangle &\mapsto_{\mathsf{fg}} \langle V, \sigma\rangle\\
\langle V, \rho, \sigma, \langle \mathrm{push} : M, \rho', \kappa\rangle\rangle &\mapsto_{\mathsf{fg}} \langle M, \rho', \sigma, \langle \mathrm{call} : V, \kappa\rangle\rangle\\
\langle V, \rho, \sigma, \langle \mathrm{call} : V', \kappa\rangle\rangle &\mapsto_{\mathsf{fg}} \langle M, \rho'[f \mapsto \beta][x \mapsto \alpha], \sigma[\alpha \mapsto V][\beta \mapsto V'], \kappa\rangle\\
&\qquad \text{if } V' = \langle \mathrm{closure} : \lambda_f x.M, \rho'\rangle \text{ and } \alpha, \beta \notin \mathrm{dom}(\sigma)\\
\langle V, \rho, \sigma, \langle \mathrm{frame} : R, \kappa\rangle\rangle &\mapsto_{\mathsf{fg}} \langle V, \rho, \sigma, \kappa\rangle\\
\langle V, \rho, \sigma, \langle \mathrm{grant} : R, \kappa\rangle\rangle &\mapsto_{\mathsf{fg}} \langle V, \rho, \sigma, \kappa\rangle
\end{aligned}
$$

$$
\begin{aligned}
\langle V, \rho, \sigma[\beta, \ldots \mapsto V', \ldots], \kappa\rangle &\mapsto_{\mathsf{fg}} \langle V, \rho, \sigma, \kappa\rangle\\
&\qquad \text{if } \{\beta, \ldots\} \text{ is nonempty and}\\
&\qquad \beta, \ldots \text{ do not occur in } V,\ \rho,\ \sigma, \text{ or } \kappa
\end{aligned}
$$

where

$$
\begin{aligned}
\mathcal{OK}_{\mathsf{fg}}[\![\emptyset]\!][\![\kappa]\!] &= \mathrm{true}\\
\mathcal{OK}_{\mathsf{fg}}[\![R]\!][\![\langle\rangle]\!] &= \mathrm{true}\\
\mathcal{OK}_{\mathsf{fg}}[\![R]\!][\![\langle \mathrm{push} : M, \rho, \kappa\rangle]\!] &= \mathcal{OK}_{\mathsf{fg}}[\![R]\!][\![\kappa]\!]\\
\mathcal{OK}_{\mathsf{fg}}[\![R]\!][\![\langle \mathrm{call} : V, \kappa\rangle]\!] &= \mathcal{OK}_{\mathsf{fg}}[\![R]\!][\![\kappa]\!]\\
\mathcal{OK}_{\mathsf{fg}}[\![R]\!][\![\langle \mathrm{frame} : R', \kappa\rangle]\!] &= \begin{cases} \mathcal{OK}_{\mathsf{fg}}[\![R]\!][\![\kappa]\!] \text{ if } R \subseteq R' \\ \mathrm{false\ otherwise} \end{cases}\\
\mathcal{OK}_{\mathsf{fg}}[\![R]\!][\![\langle \mathrm{grant} : R', \kappa\rangle]\!] &= \mathcal{OK}_{\mathsf{fg}}[\![R - R']\!][\![\kappa]\!]
\end{aligned}
$$

Fig. 1.

Definition 3 (Eval_x).

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

where

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

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 1 (Machine Fidelity). *For all* $(\langle M_0, R_0\rangle, \ldots)$,

$$\mathrm{Eval}_{\mathsf{fg}}(\langle M_0, R_0\rangle, \ldots) = V \text{ iff } \mathrm{Eval}(\langle M_0, R_0\rangle, \ldots) = V$$

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

4.2 The fg Machine Is Not Tail-Call-Optimizing

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.

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

$$\begin{aligned}
&\langle \mathcal{A}R_{\mathsf{A}}[\![\mathrm{UserFn}]\!]\ \mathcal{A}R_{\mathsf{S}}[\![\mathrm{SystemFn}]\!], \emptyset, \emptyset, \langle\rangle\rangle \\
&\mapsto_{\mathsf{fg}} \langle \mathcal{A}R_{\mathsf{A}}[\![\mathrm{UserFn}]\!], \emptyset, \emptyset, \langle \mathrm{push} : \mathcal{A}R_{\mathsf{S}}[\![\mathrm{SystemFn}]\!], \emptyset, \langle\rangle\rangle\rangle \\
&\mapsto_{\mathsf{fg}} \langle \mathrm{UserClo}, \emptyset, \emptyset, \langle \mathrm{push} : \mathcal{A}R_{\mathsf{S}}[\![\mathrm{SystemFn}]\!], \emptyset, \langle\rangle\rangle\rangle \\
&\mapsto_{\mathsf{fg}} \langle \mathcal{A}R_{\mathsf{S}}[\![\mathrm{SystemFn}]\!], \emptyset, \emptyset, \langle \mathrm{call} : \mathrm{UserClo}, \langle\rangle\rangle\rangle \\
&\mapsto_{\mathsf{fg}} \langle \mathrm{SystemClo}, \emptyset, \emptyset, \langle \mathrm{call} : \mathrm{UserClo}, \langle\rangle\rangle\rangle \\
&\mapsto_{\mathsf{fg}} \langle R_{\mathsf{A}}[sys\ \ user], \rho_0, \sigma_0, \langle\rangle\rangle \\
&\mapsto_{\mathsf{fg}} \langle sys\ \ user, \rho_0, \sigma_0, \langle \mathrm{frame} : R_{\mathsf{A}}, \langle\rangle\rangle\rangle \\
&\mapsto_{\mathsf{fg}} \langle sys, \rho_0, \sigma_0, \langle \mathrm{push} : user, \rho_0, \langle \mathrm{frame} : R_{\mathsf{A}}, \langle\rangle\rangle\rangle\rangle \\
&\mapsto_{\mathsf{fg}} \langle \mathrm{SystemClo}, \rho_0, \sigma_0, \langle \mathrm{push} : user, \rho_0, \langle \mathrm{frame} : R_{\mathsf{A}}, \langle\rangle\rangle\rangle\rangle \\
&\mapsto_{\mathsf{fg}} \langle user, \rho_0, \sigma_0, \langle \mathrm{call} : \mathrm{SystemClo}, \langle \mathrm{frame} : R_{\mathsf{A}}, \langle\rangle\rangle\rangle\rangle \\
&\mapsto_{\mathsf{fg}} \langle \mathrm{UserClo}, \rho_0, \sigma_0, \langle \mathrm{call} : \mathrm{SystemClo}, \langle \mathrm{frame} : R_{\mathsf{A}}, \langle\rangle\rangle\rangle\rangle \\
&\overset{2}{\mapsto}_{\mathsf{fg}} \langle R_{\mathsf{S}}[user\ \ sys], \rho_0, \sigma_0, \langle \mathrm{frame} : R_{\mathsf{A}}, \langle\rangle\rangle\rangle \\
&\mapsto_{\mathsf{fg}} \langle user\ \ sys, \rho_0, \sigma_0, \langle \mathrm{frame} : R_{\mathsf{S}}, \langle \mathrm{frame} : R_{\mathsf{A}}, \langle\rangle\rangle\rangle\rangle \\
&\mapsto_{\mathsf{fg}} \langle user, \rho_0, \sigma_0, \langle \mathrm{push} : sys, \rho_0, \langle \mathrm{frame} : R_{\mathsf{S}}, \langle \mathrm{frame} : R_{\mathsf{A}}, \langle\rangle\rangle\rangle\rangle\rangle \\
&\mapsto_{\mathsf{fg}} \langle \mathrm{UserClo}, \rho_0, \sigma_0, \langle \mathrm{push} : sys, \rho_0, \langle \mathrm{frame} : R_{\mathsf{S}}, \langle \mathrm{frame} : R_{\mathsf{A}}, \langle\rangle\rangle\rangle\rangle\rangle \\
&\mapsto_{\mathsf{fg}} \langle sys, \rho_0, \sigma_0, \langle \mathrm{call} : \mathrm{UserClo}, \langle \mathrm{frame} : R_{\mathsf{S}}, \langle \mathrm{frame} : R_{\mathsf{A}}, \langle\rangle\rangle\rangle\rangle\rangle \\
&\mapsto_{\mathsf{fg}} \langle \mathrm{SystemClo}, \rho_0, \sigma_0, \langle \mathrm{call} : \mathrm{UserClo}, \langle \mathrm{frame} : R_{\mathsf{S}}, \langle \mathrm{frame} : R_{\mathsf{A}}, \langle\rangle\rangle\rangle\rangle\rangle \\
&\overset{7}{\mapsto}_{\mathsf{fg}} \langle \mathrm{UserClo}, \rho_0, \sigma_0, \langle \mathrm{call} : \mathrm{SystemClo}, \langle \mathrm{frame} : R_{\mathsf{A}}, \langle \mathrm{frame} : R_{\mathsf{S}}, \langle \mathrm{frame} : R_{\mathsf{A}}, \langle\rangle\rangle\rangle\rangle\rangle\rangle \\
&\overset{7}{\mapsto}_{\mathsf{fg}} \langle \mathrm{SystemClo}, \rho_0, \sigma_0, \\
&\qquad \langle \mathrm{call} : \mathrm{UserClo}, \langle \mathrm{frame} : R_{\mathsf{S}}, \langle \mathrm{frame} : R_{\mathsf{A}}, \langle \mathrm{frame} : R_{\mathsf{S}}, \langle \mathrm{frame} : R_{\mathsf{A}}, \langle\rangle\rangle\rangle\rangle\rangle\rangle\rangle
\end{aligned}$$

5 An Alternative Implementation

5.1 How Security Inspections Really Work

A close look at λ_{sec} shows that frame 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 $\Sigma^* g y^*$; 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 non-accepting. A g always transitions to the accepting state, and a n always transitions to the non-accepting state. A y causes a (trivial) transition to the current state.

This last observation leads us to a further simplification. 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 $\Sigma^* g$.

Now consider the reduction semantics again. Although a context represents a long string, we cannot reduce all permission information in a context to a single state in our machine, because the context also contains expressions waiting to be evaluated. In other words, there are many prefixes of this "permission word" that evaluation depends on. Whenever a sequence of frame and grant expressions occurs without interruption, however, it is safe to collapse it, and it is easy to see how to do so. A substring ending in a g results in an accepting state, a substring ending in an n results in a non-accepting 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.

This reasoning also provides an intuitive understanding for the componential nature of our annotation scheme. Consider the evaluation of a program containing both annotated and unannotated components. Since this computation ignores security frames indicating the presence of a given permission, code that has not been annotated at all is equivalent to code that has been granted all permissions. This means that system libraries need not be recompiled to take advantage of such a scheme.

THE CM MACHINE

$$
\begin{aligned}
m &\in \mathcal{P} \to_f \{\text{grant}, \text{no}\} \\
\text{configurations} : C_{\mathsf{cm}} &= \langle M, \rho, \sigma, \kappa\rangle \mid \langle V, \rho, \sigma, \kappa\rangle \mid \langle V, \sigma\rangle \mid \mathsf{fail} \\
\kappa &= \langle \text{empty} : m\rangle \mid \langle \text{push} : M, \rho, \kappa, m\rangle \mid \langle \text{call} : V, \kappa, m\rangle \\
V \in \text{Values} &= \langle \text{closure} : M, \rho\rangle \\
\rho &\in \text{Identifiers} \to_f \text{Locations} \\
\alpha, \beta &\in \text{Locations} \\
\sigma &\in \text{Locations} \to_f \text{Values} \\
\text{empty}_{\mathsf{cm}} &= \langle \text{empty} : \emptyset\rangle
\end{aligned}
$$

$$
\begin{aligned}
\langle \lambda_f x.M, \rho, \sigma, \kappa\rangle &\mapsto_{\mathsf{cm}} \langle\langle \text{closure} : \lambda_f x.M, \rho\rangle, \rho, \sigma, \kappa\rangle \\
\langle x, \rho, \sigma, \kappa\rangle &\mapsto_{\mathsf{cm}} \langle \sigma(\rho(x)), \rho, \sigma, \kappa\rangle \\
\langle M\ N, \rho, \sigma, \kappa\rangle &\mapsto_{\mathsf{cm}} \langle M, \rho, \sigma, \langle \text{push} : N, \rho, \kappa, \emptyset\rangle\rangle \\
\langle R[M], \rho, \sigma, \kappa\rangle &\mapsto_{\mathsf{cm}} \langle M, \rho, \sigma, \kappa[\overline{R} \mapsto \text{no}]\rangle \\
\langle \mathsf{grant}\ R\ \mathsf{in}\ M, \rho, \sigma, \kappa\rangle &\mapsto_{\mathsf{cm}} \langle M, \rho, \sigma, \kappa[R \mapsto \text{grant}]\rangle \\
\langle \mathsf{test}\ R\ \mathsf{then}\ M\ \mathsf{else}\ N, \rho, \sigma, \kappa\rangle &\mapsto_{\mathsf{cm}} \begin{cases} \langle M, \rho, \sigma, \kappa\rangle \text{ if } \mathcal{OK}_{\mathsf{cm}}[\![R]\!][\![\kappa]\!] \\ \langle N, \rho, \sigma, \kappa\rangle \text{ otherwise} \end{cases} \\
\langle \mathsf{fail}, \rho, \sigma, \kappa\rangle &\mapsto_{\mathsf{cm}} \mathsf{fail}
\end{aligned}
$$

$$
\begin{aligned}
\langle V, \rho, \sigma, \langle \text{empty} : m\rangle\rangle &\mapsto_{\mathsf{cm}} \langle V, \sigma\rangle \\
\langle V, \rho, \sigma, \langle \text{push} : M, \rho', \kappa, m\rangle\rangle &\mapsto_{\mathsf{cm}} \langle M, \rho', \sigma, \langle \text{call} : V, \kappa, \emptyset\rangle\rangle \\
\langle V, \rho, \sigma, \langle \text{call} : V', \kappa, m\rangle\rangle &\mapsto_{\mathsf{cm}} \langle M, \rho'[f \mapsto \beta][x \mapsto \alpha], \sigma[\alpha \mapsto V][\beta \mapsto V'], \kappa\rangle \\
&\qquad \text{if } V' = \langle \text{closure} : \lambda_f x.M, \rho'\rangle \text{ and } \alpha, \beta \notin \text{dom}(\sigma)
\end{aligned}
$$

$$
\begin{aligned}
\langle V, \rho, \sigma[\beta, \ldots \mapsto V, \ldots], \kappa\rangle &\mapsto_{\mathsf{cm}} \langle V, \rho, \sigma, \kappa\rangle \\
&\qquad \text{if } \{\beta, \ldots\} \text{ is nonempty and} \\
&\qquad \beta, \ldots \text{ do not occur in } V,\ \rho,\ \sigma, \text{ or } \kappa
\end{aligned}
$$

where

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

and

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

Fig. 2.

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).

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

The two machines produce the same results.

Theorem 2 (Machine Equivalence). *For all* $(\langle M_0, R_0\rangle, \ldots)$*,*

$$\text{Eval}_{\mathsf{fg}}(\langle M_0, R_0\rangle, \ldots) = V \text{ iff } \text{Eval}_{\mathsf{cm}}(\langle M_0, R_0\rangle, \ldots) = V$$

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 will assume that no garbage collection steps are taken, because garbage collection cannot affect the result of the evaluation.

Lemma 1 (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 $\mathcal{T}$.

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

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 2 (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.

$$R_{\mathsf{S}} \triangleq \{b, c\}$$
$$R_{\mathsf{A}} \triangleq \{a, b\}$$

$\langle \mathcal{A}R_{\mathsf{A}}[\![\text{UserFn}]\!]\ \mathcal{A}R_{\mathsf{S}}[\![\text{SystemFn}]\!], \emptyset, \emptyset, \langle \text{empty} : \emptyset \rangle \rangle$
$\mapsto_{\mathsf{cm}} \langle \mathcal{A}R_{\mathsf{A}}[\![\text{UserFn}]\!], \emptyset, \emptyset, \langle \text{push} : \mathcal{A}R_{\mathsf{S}}[\![\text{SystemFn}]\!], \emptyset, \langle \text{empty} : \emptyset \rangle, \emptyset \rangle \rangle$
$\mapsto_{\mathsf{cm}} \langle \text{UserClo}, \emptyset, \emptyset, \langle \text{push} : \mathcal{A}R_{\mathsf{S}}[\![\text{SystemFn}]\!], \emptyset, \langle \text{empty} : \emptyset \rangle, \emptyset \rangle \rangle$
$\mapsto_{\mathsf{cm}} \langle \mathcal{A}R_{\mathsf{S}}[\![\text{SystemFn}]\!], \emptyset, \emptyset, \langle \text{call} : \text{UserClo}, \langle \text{empty} : \emptyset \rangle, \emptyset \rangle \rangle$
$\mapsto_{\mathsf{cm}} \langle \text{SystemClo}, \emptyset, \emptyset, \langle \text{call} : \text{UserClo}, \langle \text{empty} : \emptyset \rangle, \emptyset \rangle \rangle$
$\mapsto_{\mathsf{cm}} \langle R_{\mathsf{A}}[\mathit{sys}\ \mathit{user}], \rho_0, \sigma_0, \langle \text{empty} : \emptyset \rangle \rangle$
$\mapsto_{\mathsf{cm}} \langle \mathit{sys}\ \mathit{user}, \rho_0, \sigma_0, \langle \text{empty} : [\{c\} \mapsto \text{no}] \rangle \rangle$
$\mapsto_{\mathsf{cm}} \langle \mathit{sys}, \rho_0, \sigma_0, \langle \text{push} : \mathit{user}, \rho_0, \langle \text{empty} : [\{c\} \mapsto \text{no}] \rangle \rangle \rangle$
$\mapsto_{\mathsf{cm}} \langle \text{SystemClo}, \rho_0, \sigma_0, \langle \text{push} : \mathit{user}, \rho_0, \langle \text{empty} : [\{c\} \mapsto \text{no}] \rangle, \emptyset \rangle \rangle$
$\mapsto_{\mathsf{cm}} \langle \mathit{user}, \rho_0, \sigma_0, \langle \text{call} : \text{SystemClo}, \langle \text{empty} : [\{c\} \mapsto \text{no}] \rangle, \emptyset \rangle \rangle$
$\mapsto_{\mathsf{cm}} \langle \text{UserClo}, \rho_0, \sigma_0, \langle \text{call} : \text{SystemClo}, \langle \text{empty} : [\{c\} \mapsto \text{no}] \rangle, \emptyset \rangle \rangle$
$\overset{2}{\mapsto}_{\mathsf{cm}} \langle R_{\mathsf{S}}[\mathit{user}\ \mathit{sys}], \rho_0, \sigma_0, \langle \text{empty} : [\{c\} \mapsto \text{no}] \rangle \rangle$
$\mapsto_{\mathsf{cm}} \langle \mathit{user}\ \mathit{sys}, \rho_0, \sigma_0, \langle \text{empty} : [\{a, c\} \mapsto \text{no}] \rangle \rangle$
$\mapsto_{\mathsf{cm}} \langle \mathit{user}, \rho_0, \sigma_0, \langle \text{push} : \mathit{sys}, \rho_0, \langle \text{empty} : [\{a, c\} \mapsto \text{no}] \rangle \rangle \rangle$
$\mapsto_{\mathsf{cm}} \langle \text{UserClo}, \rho_0, \sigma_0, \langle \text{push} : \mathit{sys}, \rho_0, \langle \text{empty} : [\{a, c\} \mapsto \text{no}] \rangle, \emptyset \rangle \rangle$
$\mapsto_{\mathsf{cm}} \langle \mathit{sys}, \rho_0, \sigma_0, \langle \text{call} : \text{UserClo}, \langle \text{empty} : [\{a, c\} \mapsto \text{no}] \rangle, \emptyset \rangle \rangle$
$\mapsto_{\mathsf{cm}} \langle \text{SystemClo}, \rho_0, \sigma_0, \langle \text{call} : \text{UserClo}, \langle \text{empty} : [\{a, c\} \mapsto \text{no}] \rangle, \emptyset \rangle \rangle$
$\overset{7}{\mapsto}_{\mathsf{cm}} \langle \text{UserClo}, \rho_0, \sigma_0, \langle \text{call} : \text{SystemClo}, \langle \text{empty} : [\{a, c\} \mapsto \text{no}] \rangle, \emptyset \rangle \rangle$
$\overset{7}{\mapsto}_{\mathsf{cm}} \langle \text{SystemClo}, \rho_0, \sigma_0, \langle \text{call} : \text{UserClo}, \langle \text{empty} : [\{a, c\} \mapsto \text{no}] \rangle, \emptyset \rangle \rangle$

6 Space Consumption

In order to apply Clinger's analytic framework of TCO [4], we must extend his configuration-measuring function to handle security frames (in the case of the fg machine) and marks (in the case of the cm machine). Fortunately, we can use the same function for configurations of both machines. Applying the function to the configurations assumed by the fg and cm machines during the evaluation of a program yields space functions S_{fg} and S_{cm}, mapping programs to the maximum space consumed during the evaluations on their respective machines.

With this extension, we can define space complexity classes $O(S_{\mathsf{fg}})$ and $O(S_{\mathsf{cm}})$ as the sets of space functions that are asymptotically similar to S_{fg} and S_{cm}. We can demonstrate the inclusion of $O(S_{\mathsf{cm}})$ in $O(S_{\mathsf{fg}})$ by mapping configurations of the cm machine onto configurations of the fg machine and showing a worst-case growth of no more than the number of permissions $|\mathcal{P}|$, and the non-inclusion of $O(S_{\mathsf{fg}})$ in $O(S_{\mathsf{cm}})$ by choosing a program (like the example shown earlier) that grows without bound in the fg machine but has a finite bound in the cm machine.

To directly show that the cm machine is TCO, we must define TCO for this language. We define an oracular machine that makes the right security decisions

with no information whatsoever, and then show that the cm machine's space use is asymptotically bounded by the complexity class $O(S_o)$ induced by the oracle's space function S_o.

Theorem 3 (Space Complexity). $O(S_o) = O(S_{\mathsf{cm}}) \subset O(S_{\mathsf{fg}})$

7 A Note on TCO in Fournet and Gordon

Our reduction semantics differs from that presented by Fournet & Gordon [7]. In particular, our semantics omits runtime checks for grant expressions against their source permissions. While we have justified this omission with a static check (section 5.2), it is important to understand that our evaluator differs from Fournet & Gordon's on programs that do not satisfy this predicate.

The difference in the evaluators induces a further difference in the respective contextual equivalence theories. In Fournet & Gordon's theory, the equation

$$\emptyset[\ \mathsf{grant}\ \emptyset\ \mathsf{in}\ \mathsf{test}\ R\ \mathsf{then}\ e\ \mathsf{else}\ f] \equiv \emptyset[\ \mathsf{grant}\ R\ \mathsf{in}\ \mathsf{test}\ R\ \mathsf{then}\ e\ \mathsf{else}\ f]$$

holds. The two expressions are contextually equivalent because the permissions enabled by the grant are dynamically reduced to the empty set at runtime. In our system, though, this runtime check is omitted and the two expressions therefore produce different results.

Although this difference might suggest that the results of this paper do not apply to the semantics of Fournet & Gordon, this is not the case. To make this point, we sketch an optimization path using their theory of contextual equivalence that reduces any program to one that contains at most two frame expressions and one grant expression for each ordinary expression. This guarantees that the amount of security information in the program is linear in the size of the ordinary program.

Consider an expression containing an arbitrarily long (nested) sequence of frame and grant expressions wrapped around a single ordinary expression e. Using Fournet & Gordon's contextual equivalence theory, it can be reduced to at most two frame expressions wrapped around at most one grant expression wrapped around e. Informally, this optimization path consists of three specific optimizations, using four laws from the theory [7, pp. 311–312].

Selected Equations

(Frame Frame Frame) : $R_1[R_2[R_3[e]]] = (R_1 \cap R_2)[R_3[e]]$

(Grant Grant) : $\mathsf{grant}\ R_1\ \mathsf{in}\ \mathsf{grant}\ R_2\ \mathsf{in}\ e = \mathsf{grant}\ R_1 \cup R_2\ \mathsf{in}\ e$

(Frame Grant) : $R_1[\mathsf{grant}\ R_2\ \mathsf{in}\ e] = R_1[\mathsf{grant}\ R_1 \cap R_2\ \mathsf{in}\ e]$

(Frame Grant Frame) : $R_1 \supseteq R_2 \Rightarrow R_1[\mathsf{grant}\ R_2\ \mathsf{in}\ R_3[e]] = R_1[R_3[\mathsf{grant}\ R_2\ \mathsf{in}\ e]]$

The first reduces a sequence of three or more frame expressions to two frame expressions. The second reduces two or more grant expressions to a single grant expression. The third moves a frame outward past a grant. We conjecture that these optimizations yield a provably TCO machine semantics that is a direct modification of Fournet & Gordon's reduction semantics.

8 Related Work

This paper is directly inspired by the POPL presentation of a semantics for stack inspection by Fournet & Gordon [7], and by our earlier research on an algebraic stepper for DrScheme [3]. In this work, 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 run-time 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 paper 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.

The initial presentation of stack inspection is due to Wallach et al. [17,18]. They provide informal specifications and multiple implementations for this security architecture. Our paper aims to bridge the gap between this implementation work and the equational reasoning of Fournet & Gordon.

Several others [1,15] 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 user and library code.

Karjoth [10] presents 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 includes rules for matching 'call' and 'return' expressions, his system cannot be the foundation for a TCO implementation.

9 Conclusions

Our paper 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 develop an alternative implementation of stack inspection; we prove that it preserves the observable behavior of all programs; and we show that its notion of tail call is consistent with Clinger's 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. Finally, we expect that such an implementation will perform as well or better than a conventional stack inspection implementation.

Acknowledgments. We are grateful to C. Fournet and J. Marshall for their comments, and to M. Flatt for the design and implementation of continuation marks.

References

[1] Nick Benton, Andrew Kennedy, and George Russell. Compiling standard ML to Java bytecodes. In *ACM SIGPLAN International Conference on Functional Programming*, pages 129–140, 1998.

[2] Don Box. *Essential .NET, Volume I: The Common Language Runtime*. Addison-Wesley, To Appear.

[3] John Clements, Matthew Flatt, and Matthias Felleisen. Modeling an algebraic stepper. *Lecture Notes in Computer Science*, 2028:320–334, 2001.

[4] William D. Clinger. Proper tail recursion and space efficiency. In *ACM SIGPLAN Conference on Programming Language Design and Implementation*, pages 174–185, 1998.

[5] Matthias Felleisen and Matthew Flatt. Programming languages and their calculi. Unpublished Manuscript. Online at `<http://www.ccs.neu.edu/home/matthias/3810-w02/mono.ps.gz>`, 1989–2002.

[6] Matthew Flatt. PLT MzScheme: Language manual. Online at `<http://www.plt-scheme.org>`, 1995–2002.

[7] Cedric Fournet and Andrew D. Gordon. Stack inspection: theory and variants. In *Symposium on Principles of Programming Languages*, pages 307–318, 2002.

[8] Erich Gamma, Richard Helm, Ralph Johnson, and John Vlissides. *Design Patterns*. Addison-Wesley, 1995.

[9] Guy Lewis Steele Jr. Debunking the "expensive procedure call" myth. In *ACM Conference*, pages 153–162, 1977.

[10] Günter Karjoth. An operational semantics of Java 2 access control. In *The Computer Security Foundations Workshop*, pages 224–232, 2000.

[11] Richard Kelsey, William D. Clinger, and Jonathan Rees. Revised^5 report on the algorithmic language scheme. *SIGPLAN Notices*, 33(9):26–76, 1998.

[12] Microsoft. Common language runtime SDK documentation. Online at `http://www.microsoft.com`. Part of .NET SDK documentation, 2002.

[13] Gordon D. Plotkin. Call-by-name, call-by-value and the λ-calculus. *Theoretical Computer Science*, pages 125–159, 1975.

[14] F. Pottier, Christian Skalka, and Scott Smith. A systematic approach to static access control. *Lecture Notes in Computer Science*, 2028:30–45, 2001.

[15] Michel Schinz and Martin Odersky. Tail call elimination on the Java virtual machine. In *SIGPLAN BABEL Workshop on Multi-Language Infrastructure and Interoperability*, pages 155–168, 2001.

[16] Christian Skalka and Scott Smith. Static enforcement of security with types. *ACM SIGPLAN Notices*, 35(9):34–45, 2000.

[17] Dan Wallach, Dirk Balfanz, Drew Dean, and Ed Felten. Extensible security architectures for Java. In *The 16th Symposium on Operating Systems Principles*, pages 116–128, october 1997.

[18] Dan Wallach, Edward Felten, and Andrew Appel. The security architecture formerly known as stack inspection: A security mechanism for language-based systems. *ACM Transactions on Software Engineering and Methodology*, 9(4):341–378, October 2000.