Program Specialization for Efficient Monitored Execution

Peter Thiemann
Universität Freiburg
thiemann@informatik.uni-freiburg.de

February 20, 2001

Abstract

Monitored execution is a proven tool to enforce safety properties on applets and mobile code. Due to the inefficiency of monitoring by interpretation, many tools transform incoming code by inserting safety checks before executing it. For efficiency reasons, they attempt to insert as few checks as possible using either ad-hoc techniques or elaborate theoretical frameworks.

Partial evaluation is a powerful tool for specifying and implementing program transformations. The present work demonstrates that standard partial evaluation techniques are sufficient to transform a monitoring interpreter into a non-standard compiler. This compiler is directly applicable to incoming code and implements a transformation that guarantees safety. The transformation can elide many safety checks if the monitoring interpreter is properly staged.

The transformation runs in linear time and is guaranteed not to duplicate incoming code.

1 Introduction

Mobile code is invading our everyday lives. Many applications load modules and third-party extensions from the network and run it on the local computer. Web browsers are the most prominent of these applications, but many others (*e.g.*, mobile agents that relocate themselves proactively from one computer to the other) are gaining importance quickly.

The advent of these applications and related incidents has brought an increasing awareness of the problems involved in executing foreign and potentially hostile programs. Clearly, foreign code should not compromise the hosting computer, by crashing the computer (data integrity), by accessing/modifying data that it is not supposed to access (memory integrity), or —more generally— by using resources that it is not supposed to use. A generally accepted way of giving such guarantees is to execute the code in a *sand box*. A sand box tracks the execution of foreign code and stops it if it attempts a malicious action. This kind of *monitored execution* can enforce *safety properties*, which state that nothing bad will ever happen. Liveness properties, like fair scheduling of resources, are not in the scope of this method. However, each liveness property can be conservatively approximated by a safety property that encodes timeouts.

1.1 Related work

Monitoring sand box environments have been conceived and implemented with varying degrees of sophistication. The obvious approach to monitored execution is to run the code on an interpreter that checks each action before actually performing it. This approach is highly flexible but it involves a large overhead. Another approach, taken by the JDK [15], is to equip strategic functions in a library with calls to a security manager. A user-provided instantiation of the security manager is then responsible to keep track of the actions and to stop the code, if necessary. The latter approach is less flexible, but more efficient. Java solves the problem of data and memory integrity statically by subjecting all programs to a bytecode verification process [22].

The Omniware approach [1,23,36] guarantees memory integrity by imposing a simple program transformation on programs in assembly language. The transformation confines a foreign module to its own private data and code segment. The approach is very efficient, but of limited expressiveness.

Schneider [30] shows that all and only safety properties can be decided by keeping track of the execution history. The history is abstracted into a (not necessarily finite) state automaton. The SASI project implemented this idea [21] for x86-assembly language and for JVM bytecode. Both allow for a separate specification of a state automaton and rely on code transformation to integrate the propagation of the state with the execution of the program.

Evans and Twyman [7] have constructed a system that takes a specification of a safety policy and generates a transformed version of the Java run-time classes. Any program that uses the transformed classes is guaranteed to obey the specified safety policy.

Necula and Lee [26, 27] have developed a framework in which compiled machine programs can be combined with an encoding of a proof that the program obeys certain properties (for example, a safety policy). The resulting proof-carrying code is sent to a remote machine, which can check the proof locally against the code, to make sure that it obeys the safety policy. This has been pursued further by Appel and others [2, 24].

Kozen [18] has developed a very light-weight version of proof-carrying code. He has built a compiler that includes hints to the structure of the compiled program in the code. A receiver of such instrumented code can verify the structural hints and thus obtain confidence that the program preserves memory integrity.

Typed assembly language (TAL) [25] provides another avenue to generating high-level invariants for low-level code. Using TAL can guarantee type safety and memory integrity. TAL programs include extensive type annotations that enable the receiver to perform type checking effectively.

Colcombet and Fradet [3] propose to transform code received from a foreign principal, guided by a safety policy. The transformed code propagates a run-time encoding of a security state which is checked at run-time to avoid illegal actions. They present an elaborate framework built around a new kind of automaton to optimize the management of these states.

Walker [37] presents a sophisticated type system that can encode the passing of the security state on the type-level. The type system enables powerful optimizations. However, a separate transformation system must be implemented and lemmas about the security policy must be proved separately and fed into the system to enable optimizing transformations.

Implementing program transformations by program specialization has been proposed by Turchin and Glück [9,35] and put into practice by Glück, Jørgensen, and others [10, 11].

Since the outcome of our translation (Fig. 3) is explicitly staged (the annotations need not be analysed separately), a metaprogramming language like Meta-ML [31] could be used to implement it just as well. However, there is a problem. Our specialization approach exploits the fact that the specializer preserves the semantics to argue for the correctness of the transformation. Unfortunately, there is no such semantics-preservation result for Meta-ML.

Thiemann [34] has investigated the use of type specialization to remove (in some cases) all run-time safety checks from programs transformed for monitored execution. This approach relies on a translation to continuation-passing style that exposes information to the specializer as much as possible. The approach is appealing and works well in many cases, but it is prone to code duplication. Also, due to the sophisticated specializer, the transformation phase may take exponential time. This behavior might be exploited by a malicious attacker to stage a denial of service attack. In contrast, the techniques presented in the present work are guaranteed to never duplicate downloaded code and to perform the transformation in linear time. This is due to the reliance on a less powerful but more efficient specialization technique, which is less effective in removing run-time safety checks but has guaranteed runtimes.

1.2 Overview

The present work demonstrates that standard partial evaluation techniques can implement program transformations that guarantee safety. This is advantageous because it avoids ad-hoc solutions by reusing partial evaluation practice (like efficient code generation) and theory. For example, the correctness of a transformation is guaranteed through the correctness of the partial evaluator. This allows for modular correctness proofs of the trusted computing base and enables the user of monitored execution to concentrate on the specification of the safety policy and on its correctness.

We start from a naive translation (in Sec. 3) and modify it in Sec. 4 to yield two-level terms (in the sense of [17]) which explicitly manage the computation of a security state. Traditional partial evaluation of these terms implements a homogeneous translation that never duplicates incoming code. It yields satisfactory compiled programs. We consider a variant of the translation that is more general and yields better results (Sec. 4.2), comparable to the results of Colcombet and Fradet's method [3].

The purpose of this paper is to demonstrate the concepts involved in a manner as simple as possible. Elsewhere [34], we demonstrate the use of an advanced specialization method with a much more complicated translation.

The main technical results are the correctness proofs of the translations and the non-standard compilers generated by partial evaluation. They guarantee the safety of the translated code and of the compiled code.

2 Prerequisites

This section introduces the source language, the concept of a security automaton to encode safety properties, and traditional partial evaluation.

2.1 Source language

The source language is a simply-typed call-by-value lambda calculus with constants, conditionals, tuples, and primitive operations on (tuples of) base type values (see Figure 1). The value of $\mathtt{fix}\ x_0(x_1, \ldots, x_n)e$ is a recursively defined function. We write $\lambda(x_1, \ldots, x_n)e$ if x_0 does not appear in e, and $\mathtt{let}\ (x_1, \ldots, x_n) = e_1\ \mathtt{in}\ e_2$ for $(\lambda(x_1, \ldots, x_n)e_2)@e_1$. The latter is the only way to inspect the components of a tuple. The standard typing rules defining the judgement $\Gamma \vdash e : \tau$ are omitted.

The semantics is defined by a labeled transition relation $\xrightarrow{t}$, where each label, t, is a sequence of primitive operations and their arguments. The label, ε, denotes the empty sequence. Each primitive operation, $\mathtt{O} : \mathrm{BaseType}^n \to \mathrm{BaseType}$, is defined through an associated partial semantic function $[\![\mathtt{O}]\!] \in \mathrm{BaseType}^n \hookrightarrow \mathrm{BaseType}$.

Each reduction sequence $e_0 \xrightarrow{t_1} e_1 \xrightarrow{t_2} \ldots$ gives rise to a potentially infinite sequence $\vec{t} = (t_1, t_2, \ldots)$ of primitive operations (an execution *trace*). Write $e \downarrow^t v$ if there is a finite sequence of reductions, $e \xrightarrow{t}{}^{*} v$.

2.2 Security automata

A *trace policy* is a set of finite and infinite traces. A reduction sequence is acceptable if its execution trace is contained in the policy. Safety policies are particular trace policies which place certain restrictions on the set of traces. Schneider [30] has shown that all safety policies can be modeled by a state automaton with a distinguished sink state *bad*.

A security automaton is a tuple $\mathcal{S} = (\Sigma, \mathit{Op}, \mathit{Value}, \delta, \sigma_0, \mathit{bad})$ [37] where

- Σ is a countable set of states;
- Op is a finite set of operation symbols;

Syntax

$$
\begin{array}{llcl}
\textit{Exp} \ni & e & ::= & v \mid (\texttt{if}\ e\ e\ e) \mid \texttt{O}(e) \mid e@e \mid (e,\ldots,e) \\
\textit{Value} \ni & v & ::= & x \mid a \mid \texttt{fix}\ f(x,\ldots,x)e \mid (v,\ldots,v) \\
\text{evaluation contexts} & C & ::= & (\texttt{if}\ [\,]\ e\ e) \mid \texttt{O}([\,]) \mid [\,]@e \mid v@[\,] \mid \\
 & & & (v,\ldots,[\,],e,\ldots) \\
\text{security states} & \sigma & \in & \Sigma \\
\text{base-type constants} & a & \in & \textit{Base} \\
\text{primitive operators} & \texttt{O} & \in & \textit{Op} \\
\text{types} & \tau & ::= & \text{BaseType} \mid \tau \to \tau \mid (\tau,\ldots,\tau) \\
\text{traces} & t & ::= & \varepsilon \mid \texttt{O}(a,\ldots,a) \mid t\,t
\end{array}
$$

Labeled reduction rules

$$
\begin{array}{lcll}
(\texttt{if true}\ e_1\ e_2) & \xrightarrow{\varepsilon} & e_1 & \\
(\texttt{if false}\ e_1\ e_2) & \xrightarrow{\varepsilon} & e_2 & \\
\texttt{O}(a_1 \ldots a_n) & \xrightarrow{\texttt{O}(a_1 \ldots a_n)} & a & \text{if } a = [\![\texttt{O}]\!](a_1,\ldots,a_n) \text{ is defined}
\end{array}
$$

$$
(\texttt{fix}\ x_0(x_1,\ldots,x_n)e)@(v_1,\ldots,v_n) \xrightarrow{\varepsilon} e[x_0 \mapsto \texttt{fix}\ x_0(x_1,\ldots,x_n)e, x_1 \mapsto v_1,\ldots,x_n \mapsto v_n]
$$

Contextual rules

$$
\frac{e \xrightarrow{t} e'}{C[e] \xrightarrow{t} C[e']} \qquad e \xrightarrow{\varepsilon}{}^{*}\ e \qquad \frac{e \xrightarrow{t} e' \quad e' \xrightarrow{t'}{}^{*}\ e''}{e \xrightarrow{tt'}{}^{*}\ e''}
$$

Figure 1: The source language

- *Value* is a countable set of values;
- $\delta : \textit{Op} \times \textit{Value}^{*} \times \Sigma \to \Sigma$ is a total function with $\delta(\texttt{O}, x_1 \ldots x_n, \textit{bad}) = \textit{bad}$ (state transition function);
- $\sigma_0 \in \Sigma$ is the initial state; and
- $\textit{bad} \in \Sigma$ is the sink state with $\sigma_0 \neq \textit{bad}$.

For a trace, t, define the iterated transition function $\delta^{*}(t,\sigma)$ by

- $\delta^{*}(\varepsilon,\sigma) = \sigma$;
- $\delta^{*}(\texttt{O}(a_1 \ldots a_n),\sigma) = \delta(\texttt{O}, a_1 \ldots a_n, \sigma)$; and
- $\delta^{*}(t_1 t_2,\sigma) = \delta^{*}(t_2, \delta^{*}(t_1,\sigma))$.

A closed term e is *safe* with respect to $\mathcal{S}$ and some $\sigma \in \Sigma \setminus \{bad\}$ if for all traces t and expressions e' where $e \xrightarrow{t}{}^{*} e'$, it holds that $\delta^*(t, \sigma) \neq bad$. It is safe with respect to $\mathcal{S}$ if it is safe with respect to the initial state σ_0.

A typical example is the policy that no network `send` operation happens after a `read` operation from a local file. The set of states has three elements

$$\Sigma = \{\mathit{before\text{-}read}, \mathit{after\text{-}read}, \mathit{bad}\}$$

with initial state $\sigma_0 = \mathit{before\text{-}read}$. The transition functions are the identity functions for all primitive operations except `send` and `read`:

σ	$\delta(\mathtt{read}, \mathit{file}, \sigma)$	$\delta(\mathtt{send}, \mathit{data}, \sigma)$	$\delta(\mathtt{0}, y_1 \ldots y_n, \sigma)$
before-read	*after-read*	*before-read*	*before-read*
after-read	*after-read*	*bad*	*after-read*
bad	*bad*	*bad*	*bad*

The program $(\lambda(x)\mathtt{read}(\mathit{file}))@(\mathtt{send}(\mathit{data}))$ is safe (with respect to σ_0) since its trace is $t = (\mathtt{send}(\mathit{data}), \mathtt{read}(\mathit{file}))$ and $\delta^*(t, \sigma_0) = \mathit{after\text{-}read}$. It is not safe with respect to *after-read* since $\delta^*(t, \mathit{after\text{-}read}) = \mathit{bad}$.

The program $(\lambda(x)\mathtt{send}(\mathit{data}))@(\mathtt{read}(\mathit{file}))$ is not safe with respect to any state: its trace $t' = (\mathtt{read}(\mathit{file}), \mathtt{send}(\mathit{data}))$ yields $\delta^*(t', \mathit{before\text{-}read}) = \mathit{bad}$ as well as $\delta^*(t', \mathit{after\text{-}read}) = \mathit{bad}$.

2.3 Partial evaluation

Traditional partial evaluation techniques [4, 17] rely on non-standard interpretation or evaluation of a source program to perform as many operations on compile-time data as possible. In *offline partial evaluation*, a binding-time analysis determines the compile-time operations before actually specializing the code. The analysis communicates with the specializer using two-level terms, which indicate code generation by underlining and compile-time operations by overlining.

Although offline partial evaluation misses some specialization opportunities, it demonstrates the concepts involved in compiling monitoring execution well.

- It is simple to understand as non-standard interpretation.
- It is easy to predict the form and the size of the compiled code as well as the compilation speed by inspecting the two-level term.
- It gives satisfactory results for this application.
- It is possible to circumvent the analysis and to directly generate two-level terms if it can be proven in advance that they are well-annotated with respect to the analysis.
- Specialization is very fast.

We rely on the PGG system [33], which specializes extremely fast because it relies on the *cogen approach* to program specialization [19]. In this approach, there is no interpretation of two-level terms. Instead, two-level terms are executed directly using a fast implementation of the code generating constructs [32].

Proving the correctness of a partial evaluator amounts to proving the MIX-equation [17]. Suppose `spec` is the program text of a specializer, p is a two-level term denoting a program with inputs $\vec{x}$ and $\vec{y}$. Let further $[\![\ ]\!]$ map a program text to a function from inputs to output. Considering the inputs $\vec{x}$ as early inputs and the inputs $\vec{y}$ as late inputs, the MIX-equation reads

$$[\![[\![\texttt{spec}]\!]\ p\ \vec{x}]\!]\ \vec{y} = [\![\mathit{erase}(p)]\!]\ \vec{x}\ \vec{y}. \tag{1}$$

The function *erase*() maps a two-level term to a standard term by erasing all overlining and underlining annotations as well as $\mathit{erase}(\texttt{lift}\ e) = \mathit{erase}(e)$ (The construct $\texttt{lift}\ y$ converts a compile-time value, y, into a literal with the same run-time value.). In reading this equation, it is assumed that $p' = [\![\texttt{spec}]\!]\ p\ \vec{x}$ always yields a result. Furthermore, the specialized program, p', is assumed to terminate on input $\vec{y}$ if and only if the original program, p, terminates on input $\vec{x}\ \vec{y}$.

The text of the specialized (compiled) program is $[\![\texttt{spec}]\!]\ p\ \vec{x}$. The cogen approach mentioned above produces a program, which is equivalent to $\lambda(\vec{x})[\![\texttt{spec}]\!]\ p\ \vec{x}$, but is much more efficient.

The correctness of offline partial evaluation has been considered in a number of places [5, 12, 13, 20]. In this work, we take the correctness of the underlying specializer for granted and make use of (1) where required.

3 Monitoring execution

A simple way to enforce safe execution is to incorporate a security automaton into an interpreter and run it in lockstep with the primitive operations. Before attempting a primitive operation, the interpreter steps the security state according to the operation and its arguments and checks whether the result is *bad*. If that is the case, the program is stopped. Otherwise, the interpreter performs the primitive operation and continues using the new security state.

We often consider an interpreter and the translation achieved by unfolding the interpreter's defining equations interchangeably. This view is quite natural because partially evaluating an interpreter implements exactly this translation.

Figure 2 shows a translation that makes the sequencing of the operations and the passing of the security state explicit. The output of the translation is in A-normal form [8] and thus particularly suited to compiling it. In particular, each `let` on the right side of the translation corresponds to a few machine instructions (often just one). Thus, it is a translation into a low-level language.

The translation inserts the stepping and testing of the security state as mentioned above. It amounts to a standard monadic translation to state-passing style (state transformations), extended by the handling of primitive operations.

The translation acts on types as follows.

Types

$$
\begin{array}{lcl}
||\mathrm{BaseType}|| & = & \mathrm{BaseType} \\
||\tau_1 \to \tau_2|| & = & ||\tau_1|| \to |\tau_2| \\
||(\tau_1, \ldots, \tau_n)|| & = & (||\tau_1||, \ldots, ||\tau_n||) \\
|\tau| & = & \Sigma \to (\Sigma, ||\tau||)
\end{array}
$$

Type environments

$$||\emptyset|| = \emptyset \qquad ||\Gamma, x : \tau|| = ||\Gamma||, x : ||\tau||$$

Values

$$
\begin{array}{lcl}
||x|| & = & x \\
||a|| & = & a \\
||\mathtt{fix}\ x_0(x_1 \ldots x_n)e|| & = & \mathtt{fix}\ x_0(x_1, \ldots, x_n)\lambda(\sigma)|e|\sigma \\
||(v_1, \ldots, v_n)|| & = & (||v_1||, \ldots, ||v_n||)
\end{array}
$$

Terms

$$
\begin{array}{lcl}
|v|\sigma & = & (\sigma, ||v||) \\
|(\mathtt{if}\ e_1\ \ e_2\ \ e_3)|\sigma & = & \mathtt{let}\ (\sigma_1, y_1) = |e_1|\sigma\ \mathtt{in} \\
 & & \mathtt{if}\ y_1\ \mathtt{then}\ |e_2|\sigma_1\ \mathtt{else}\ |e_3|\sigma_1 \\
|(e_1, \ldots, e_n)|\sigma & = & \mathtt{let}\ (\sigma_1, y_1) = |e_1|\sigma\ \mathtt{in}\ \ldots \\
 & & \mathtt{let}\ (\sigma_n, y_n) = |e_n|\sigma_{n-1}\ \mathtt{in} \\
 & & (\sigma_n, (y_1, \ldots, y_n)) \\
|\mathtt{O}(e)|\sigma & = & \mathtt{let}\ (\sigma', y) = |e|\sigma\ \mathtt{in} \\
 & & \mathtt{let}\ \sigma'' = \delta(\mathtt{O}, y, \sigma')\ \mathtt{in} \\
 & & \mathtt{if}\ \sigma'' = \mathit{bad}\ \mathtt{then}\ \mathtt{halt}()\ \mathtt{else}\ (\sigma'', \mathtt{O}(y)) \\
|e_0@e_1|\sigma & = & \mathtt{let}\ (\sigma_0, y_0) = |e_0|\sigma\ \mathtt{in} \\
 & & \mathtt{let}\ (\sigma_1, y_1) = |e_1|\sigma_0\ \mathtt{in} \\
 & & y_0@\sigma_1@y_1
\end{array}
$$

Figure 2: Monitoring interpreter

Proposition 1 *1. If $\Gamma \vdash e : \tau$ then $||\Gamma|| \vdash |e| : |\tau|$.*

2. If $\Gamma \vdash v : \tau$ then $||\Gamma|| \vdash ||v|| : ||\tau||$.

A translated program never violates the security policy since the primitive operation δ on the explicit security state does not affect the security automaton. Formally, let $\mathcal{S}' = (\Sigma, Op', Value, \delta', \sigma_0, bad)$ with $Op' = Op \cup \{\delta, \mathtt{halt}\}$ where δ' is specified by $\delta'(\mathtt{O}, y_1 \ldots y_n, \sigma) = \delta(\mathtt{O}, y_1 \ldots y_n, \sigma)$, for all $\mathtt{O} \in Op$, $\delta'(\delta, y_0 y_1 \ldots y_n, \sigma) = \sigma$, and $\delta'(\mathtt{halt}, y_0, \sigma) = \sigma$. Let $[\![\mathtt{halt}]\!]() = a$, an arbitrary constant signalling an error.

An expression translated with state σ is safe with respect to $\mathcal{S}'$ and σ.

Proposition 2 *Let $\sigma \neq bad$. If $|e|\sigma \downarrow^{t'} (\sigma', v')$ then $\delta'^*(t', \sigma) = \sigma'$ where $\sigma' \neq bad$ and $v' = ||v||$, for some v.*

If the original term delivers a result without entering a bad state then so does the translated term. To state this, it is necessary to relate execution traces for $\mathcal{S}$ with those of $\mathcal{S}'$. If t is a trace for $\mathcal{S}$ and σ is a security state then define $|t|\sigma$ as follows:

- $|\varepsilon|\sigma = \varepsilon$;
- $|\mathtt{O}(a_1 \dots a_n)|\sigma = \delta(\mathtt{O}, a_1 \dots a_n, \sigma)\ \mathtt{O}(a_1 \dots a_n)$; and
- $|t_1 t_2|\sigma = |t_1|\sigma\ |t_2|(\delta^*(t_1, \sigma))$.

Hence, each operation in a trace is transformed into its checking operation $\delta(\mathtt{O}, \dots)$ followed by the actual operation.

Proposition 3 *Let $\sigma \neq bad$. Suppose $e \downarrow^t v$ and $\sigma' = \delta^*(t, \sigma)$. If $\sigma' \neq bad$ then $|e|\sigma \downarrow^{t'} (\sigma', ||v||)$ where $t' = |t|\sigma$.*

If evaluation of the translated term leads to non-termination or to an undefined primitive operation then so does evaluation of the source term.

Proposition 4 *If there exist no σ', v', and t' such that $|e|\sigma \downarrow^{t'} (\sigma', v')$ then there exist no v and t such that $e \downarrow v$.*

4 Compiling policies by partial evaluation

The naive translation (Fig. 2) is already amenable to partial evaluation, by considering the functions $|\cdot|$ and $||\cdot||$ as compile-time functions and unfolding their invocations. The resulting programs are safe, but inefficient because they test the security state at every primitive operation.

To achieve more interesting results, we must transform the naive translation to (1) expose compile-time information about the security state and to (2) stage all operations to make sure that compile-time operations do not depend on run-time values. The first task requires some creativity and insight in the particular problem. The second task is performed with the help of a type system or a binding-time analysis.

The revised translation makes the staging explicit using a two-level language. In a two-level program, overlining indicates compile-time operations whereas underlinging indicates run-time operations. The same markup applies to types, too. Running a two-level program executes the compile-time operations and generates code for the run-time operations. Invocations of the translation are considered as compile-time operations. Specializing the translation with respect to an incoming program results in an optimized, safe program.

4.1 Staged translation

The first idea is to make the security state a compile-time value. However, this approach quickly leads to problems. In a traditional partial evaluator, run-time data structures cannot contain compile-time values. In particular, run-time

Types

$$
\begin{array}{lcl}
||\mathrm{BaseType}||_o & = & \underline{\mathrm{BaseType}} \\
||\tau_1 \to \tau_2||_o & = & (||\tau_1||_o)\underline{\to}|\tau_2|_d \\
||(\tau_1, \ldots, \tau_n)||_o & = & (||\tau_1||_o, \ldots, ||\tau_n||_o) \\
|\tau|_d & = & (\underline{\Sigma})\underline{\to}(\underline{\Sigma}, ||\tau||_o) \\
|\tau|_o & = & ((\overline{\Sigma})\overline{\to}\overline{\Sigma}, \underline{\Sigma})\overline{\to}((\overline{\Sigma})\overline{\to}\overline{\Sigma}, \underline{\Sigma}, ||\tau||_o)
\end{array}
$$

Type environments

$$
||\emptyset||_o = \emptyset \qquad\qquad ||\Gamma, x : \tau||_o = ||\Gamma||_o, x : ||\tau||_o
$$

Values

$$
\begin{array}{lcl}
||x||_o & = & x \\
||a||_o & = & a \\
||\mathtt{fix}\ x_0(x_1 \ldots x_n)e||_o & = & \underline{\mathtt{fix}}\ x_0(x_1, \ldots, x_n)\underline{\lambda}\sigma. \\
 & & \qquad \overline{\mathtt{let}}\ (\overline{\delta}, \sigma', y) = |e|_o\overline{@}(\overline{\lambda}\overline{\sigma}.\overline{\sigma}, \sigma)\ \overline{\mathtt{in}}\ (\overline{\delta}(\sigma'), y) \\
||(v_1, \ldots, v_n)||_o & = & (||v_1||_o, \ldots, ||v_n||_o)
\end{array}
$$

Terms

$$
\begin{array}{ll}
\multicolumn{2}{l}{|v|_o(\overline{\delta}, \sigma)} \\
= & (\overline{\delta}, \sigma, ||v||_o) \\
\multicolumn{2}{l}{|(\mathtt{if}\ e_1\ e_2\ e_3)|_o(\overline{\delta}, \sigma)} \\
= & \overline{\mathtt{let}}\ (\overline{\delta}_1, \sigma_1, y_1) = |e_1|_o\overline{@}(\overline{\delta}, \sigma)\ \overline{\mathtt{in}} \\
 & \underline{\mathtt{let}}\ (\sigma', y') = \underline{\mathtt{if}}\ y_1 \\
 & \qquad\qquad \underline{\mathtt{then}}\ \overline{\mathtt{let}}\ (\overline{\delta}_2, \sigma_2, y_2) = |e_2|_o\overline{@}(\overline{\delta}_1, \sigma_1)\ \overline{\mathtt{in}}\ (\overline{\delta}_2(\sigma_2), y_2) \\
 & \qquad\qquad \underline{\mathtt{else}}\ \overline{\mathtt{let}}\ (\overline{\delta}_3, \sigma_3, y_3) = |e_3|_o\overline{@}(\overline{\delta}_1, \sigma_1)\ \overline{\mathtt{in}}\ (\overline{\delta}_3(\sigma_3), y_3) \\
 & \underline{\mathtt{in}}\ (\overline{\lambda}\overline{\sigma}.\overline{\sigma}, \sigma', y') \\
\multicolumn{2}{l}{|(e_1 \ldots e_n)|_o(\overline{\delta}, \sigma)} \\
= & \overline{\mathtt{let}}\ (\overline{\delta}_1, \sigma_1, y_1) = |e_1|_o\overline{@}(\overline{\delta}, \sigma)\ \overline{\mathtt{in}}\ \ldots \\
 & \overline{\mathtt{let}}\ (\overline{\delta}_n, \sigma_n, y_n) = |e_n|_o\overline{@}(\overline{\delta}_{n-1}, \sigma_{n-1})\ \overline{\mathtt{in}} \\
 & (\overline{\delta}_n, \sigma_n, (y_1, \ldots, y_n)) \\
\multicolumn{2}{l}{|\mathtt{O}(e)|_o(\overline{\delta}, \sigma)} \\
= & \overline{\mathtt{let}}\ (\overline{\delta}', \sigma', y) = |e|_o\overline{@}(\overline{\delta}, \sigma)\ \overline{\mathtt{in}} \\
 & \overline{\mathtt{let}}\ \overline{\delta}'' = \overline{\delta}_O \circ \overline{\delta}'\ \overline{\mathtt{in}} \\
 & \overline{\mathtt{let}}\ B = \overline{\delta}''^{-1}(bad)\ \overline{\mathtt{in}} \\
 & \overline{\mathtt{if}}\ B = \emptyset\ \overline{\mathtt{then}}\ \underline{\mathtt{let}}\ y' = \underline{O}(y)\ \underline{\mathtt{in}}\ (\overline{\delta}'', \sigma', y')\ \overline{\mathtt{else}} \\
 & \underline{\mathtt{let}}\ \sigma'' = \underline{\delta}(\mathtt{O}, y, \overline{\delta}'(\sigma'))\ \underline{\mathtt{in}} \\
 & \underline{\mathtt{if}}\ \sigma'' = bad\ \underline{\mathtt{then}}\ \underline{\mathtt{halt}()}\ \underline{\mathtt{else}} \\
 & \underline{\mathtt{let}}\ y' = \underline{O}(y)\ \underline{\mathtt{in}}\ (\overline{\lambda}\overline{\sigma}.\overline{\sigma}, \sigma'', y') \\
\multicolumn{2}{l}{|e_0 @ e_1|_o(\overline{\delta}, \sigma)} \\
= & \overline{\mathtt{let}}\ (\overline{\delta}_0, \sigma_0, y_0) = |e_0|_o\overline{@}(\overline{\delta}, \sigma)\ \overline{\mathtt{in}} \\
 & \overline{\mathtt{let}}\ (\overline{\delta}_1, \sigma_1, y_1) = |e_1|_o\overline{@}(\overline{\delta}_0, \sigma_0)\ \overline{\mathtt{in}} \\
 & \underline{\mathtt{let}}\ (\sigma', y') = y_0\underline{@}(\overline{\delta}_1(\sigma_1))\underline{@}(y_1)\ \underline{\mathtt{in}} \\
 & (\overline{\lambda}\overline{\sigma}.\overline{\sigma}, \sigma', y')
\end{array}
$$

Figure 3: Staged translation

functions neither take compile-time parameters nor deliver compile-time results. Since the translation must treat all functions from the incoming code as run-time functions, it is not possible to thread a compile-time state through them (at least not without code duplication, which we wanted to avoid).

But perhaps we can represent the changes to the security state by a compile-time value. Unfortunately, even that is not possible, in general, because δ can depend on the actual (run-time) arguments to a primitive operation.

However, we can consider the compile-time component, $\overline{\delta}(\mathtt{O}) : \overline{\mathtt{State}} \to \overline{\mathtt{State}}$, of the transition function, δ. It is defined by

$$\overline{\delta}(\mathtt{O})(\overline{\sigma}) := \begin{cases} \overline{\sigma}' & \forall y_1 \dots y_n . \delta(\mathtt{O}, y_1, \dots, y_n, \sigma) = \sigma' \\ bad & \forall \sigma' . \exists y_1 \dots y_n . \delta(\mathtt{O}, y_1, \dots, y_n, \sigma) \neq \sigma' \end{cases}$$

In the first case, the next state is independent of the actual arguments to $\mathtt{O}$, so it can be computed at compile-time. In the second case, the next state can only be computed at run-time. The latter is signaled by returning *bad* at compile-time.

Since the initial security state, σ, is not available, a direct test like $\overline{\delta}(\sigma) = bad$ is not possible. Instead, the translation in Figure 3 uses the preimage map, $\overline{\delta}^{-1}$. Each translated term receives a pair of a compile-time transition function, $\overline{\delta}$, and a run-time state, σ. The current state is always represented by $\overline{\delta}(\sigma)$. The translated term only updates the run-time component, σ, lazily.

Suppose the translated term is about to execute a primitive operation, $\mathtt{O}$. Let $\overline{\delta}'$ and σ' be the transition function and the state after evaluating the argument of $\mathtt{O}$. First, the translated term forms the function $\overline{\delta}'' = \overline{\delta}_O \circ \overline{\delta}'$. To test whether $\overline{\delta}''(\sigma')$ (the state after the operation) is not *bad*, it is sufficient to test the preimage of *bad* for $\overline{\delta}''^{-1}(bad) = \emptyset$ at compile-time. If this test succeeds, the operation is performed and the compile-time transition is updated to $\overline{\delta}''$ in the rest of the computation. Otherwise, the run-time state is bumped to $\sigma'' = \overline{\delta}'(\sigma')$, a run-time test is performed using $\underline{\delta}$, and the compile-time transition is reset to the identity function.

In the case of `fix`, there is no compile-time information on the state, so the compile-time transition is set to the identity. At the end of the body, the accumulated compile-time transition is applied to the run-time state to return a run-time value to the caller. We call this *flushing* the compile-time information. Dually, in the case of function application, the current state is passed to the function as a run-time value and the return value is paired up with a compile-time identity as transition function.

This explains the two different translations for the type of an expressions, $|\cdot|_d$ and $|\cdot|_o$. The translation $|\cdot|_d$ applies to the result of a run-time function and thus returns a run-time type, where the compile-time parts have been flushed as just explained. The translation $|\cdot|_o$ gives a genuine two-level type with compile-time and run-time parts to expressions that do not immediately crop up as the body of a function.

Similar reasoning applies in the case of `if`. The translation processes both branches of the `if` and obtains two different compile-time functions, $\overline{\delta}_2$ and $\overline{\delta}_3$, as well as two different run-time states, σ_2 and σ_3. The translation synchronizes them by turning them into a run-time state that is returned along with a compile-time identity as before. It is possible to improve on this behavior by starting from a translation to continuation-passing style, but this improvement can lead to code duplication.

Another alternative would be to statically compute $\overline{\delta}_2$ and $\overline{\delta}_3$. If they are equal and if they still refer to the same run-time state, then the conditional may return $(\overline{\delta}_2, \sigma_1, y')$. This amounts roughly to renumbering states as suggested by Colcombet and Fradet [3].

The preimage can be easily computed if $\overline{\texttt{State}}$ is finite. Otherwise, the preimage functions must be part of the specification. The implementation encodes $\overline{\delta}$ by a list of primitive operators in reverse order of execution. The meaning of the list $\texttt{O}_n \dots \texttt{O}_1$ is the composition of the associated state transition functions $\delta(\texttt{O}_n) \circ \dots \circ \delta(\texttt{O}_1)$ and the empty list stands for $\overline{\lambda}\sigma.\sigma$. This representation also enables simple computation of $\overline{\delta}(\sigma)$ regardless of σ's binding time.

4.2 Refined translation

An annoying problem with the above definition of $\overline{\delta}$ becomes apparent when using a refined `read` primitive, `rread`, in the running example. Suppose that reading before sending is disallowed only if sensitive files have been read, that is,

$$\delta(\texttt{rread}, \textit{file}, \textit{before-read}) := \begin{cases} \textit{after-read} & \text{if } \textit{sensitive (file)} \\ \textit{before-read} & \text{otherwise.} \end{cases}$$

In this case, $\overline{\delta}_{\texttt{rread}}(\textit{before-read}) = \textit{bad}$, so that every `rread` operation in the translated term comes with a run-time check. However, these run-time checks are superfluous because `rread` never leads to state *bad*.

To address this problem, we redefine $\overline{\delta}$ so that it is no longer a function but rather a binary relation on Σ:

$$(\sigma, \sigma') \in \overline{\delta}(\texttt{O}) \quad \Leftrightarrow \quad \exists y_1 \dots y_n . \delta(\texttt{O}, y_1, \dots, y_n, \sigma) = \sigma' \qquad (2)$$

In addition, the translated term keeps track of a set, S, of possible current states. This set S is a compile-time value, too. The interpretation of $\overline{\delta}$ is a function $\mathcal{P}(\Sigma) \to \mathcal{P}(\Sigma)$ where $\overline{\delta}(\texttt{O})(S) = \{\sigma' \mid \sigma \in S, (\sigma, \sigma') \in \overline{\delta}(\texttt{O})\}$. Figure 4 shows the revised translation for primitives. The remaining parts do not change conceptually.

The value of S is initialized at the usual places (`fix` and application) to $\Sigma \setminus \{\textit{bad}\}$[1]. As a further advantage of this representation, it is possible to extract information out of a conditional more often. After processing the branches of a conditional, the resulting sets can be merged, thus giving a conservative estimate of the actual run-time security state. The other compile-time information needs to be flushed as explained before.

The function, d, represents the lag between the security state maintained by the translated program, σ, and the actual security state. It is used to bump σ

[1]Better approximations are possible here by using control flow analysis [28].

$$
\begin{aligned}
&|\mathtt{O}(e)|'_o(S,d,\sigma)\\
&\quad= \overline{\mathtt{let}}\ (S',d',\sigma',y) = |e|'_o(S,d,\sigma)\ \overline{\mathtt{in}}\\
&\qquad\ \overline{\mathtt{let}}\ S'' = \overline{\delta}(\mathtt{O})(S')\ \overline{\mathtt{in}}\\
&\qquad\ \overline{\mathtt{let}}\ d'' = (\lambda(\sigma)\delta(\mathtt{O},y,\sigma)) \circ d'\ \overline{\mathtt{in}}\\
&\qquad\ \overline{\mathtt{if}}\ bad \notin S''\ \overline{\mathtt{then}}\ \underline{\mathtt{let}}\ y' = \underline{O}(y)\ \underline{\mathtt{in}}\ (S'',d'',\sigma',y')\ \overline{\mathtt{else}}\\
&\qquad\ \underline{\mathtt{let}}\ \sigma'' = d''(\sigma')\ \underline{\mathtt{in}}\\
&\qquad\ \underline{\mathtt{if}}\ \sigma''\underline{=}\mathtt{lift}\ bad\ \underline{\mathtt{then}}\ \underline{\mathtt{halt}()}\ \underline{\mathtt{else}}\\
&\qquad\ \underline{\mathtt{let}}\ y' = \underline{O}(y)\ \underline{\mathtt{in}}\\
&\qquad\ (S''\setminus\{bad\},\lambda(\sigma)\sigma,\sigma'',y')
\end{aligned}
$$

Figure 4: Refined treatment of primitive operators

to the actual state. It is a composition of run-time functions. Depending on the representation and on the cardinality of Σ, most of the compositions can be performed at compile-time. Memoization can be employed to ensure that each composition is computed at most once. The security state, σ, is a run-time value.

The translation maintains the following invariant. If S, d, σ are the current values in the translated term and σ' is the state obtained from the trace of the semantics, then $\sigma' = d(\sigma)$ and $\sigma' \in S$ and $bad \notin S$.

4.3 Properties of the staged translation

This section only states the properties of the translation in Figure 3. The properties of the refined translation (Fig. 4) are similar.

The translation is type preserving and fulfills the well-formedness restriction of partial evaluation. The judgement $\Gamma \vdash e : \tau$ states the latter fact [17].

Proposition 5 *1. If* $\Gamma \vdash e : \tau$ *then* $||\Gamma||_o \vdash |e|_o : |\tau|_o$.

2. If $\Gamma \vdash v : \tau$ *then* $||\Gamma||_o \vdash ||v||_o : ||\tau||_o$.

Next, we prove a sequence of results analogous to the ones in Sec. 3. The only difference is that we first have to erase the newly introduced two-level annotations.

A translated expression is safe with respect to $\mathcal{S}'$ (see Sec. 3) and σ.

Proposition 6 *Let* $\sigma \neq bad$. *If* $erase(|e|_o(\overline{\lambda}\overline{\sigma}.\overline{\sigma},\sigma)) \downarrow^{t'} (\overline{\delta}', \sigma', v')$ *then* $\delta'^{*}(t',\sigma) = \overline{\delta}'(\sigma')$ *where* $\overline{\delta}'(\sigma') \neq bad$ *and* $v' = erase(||v||)$, *for some* v.

If the original term delivers a result without entering a bad state then so does the translated term.

Proposition 7 *Let $\sigma \neq bad$ and $\sigma' \neq bad$. Suppose $e \downarrow^t v$ and $\sigma' = \delta^*(t, \sigma)$.*

Then $erase(|e|_o(\overline{\lambda}\overline{\sigma}.\overline{\sigma}, \sigma)) \downarrow^{t'} (\overline{\delta}', \sigma'', erase(||v||_o))$, so that $\overline{\delta}'(\sigma'') = \sigma'$ and $t' = |t|\sigma$.

If evaluation of the translated term leads to non-termination or to an undefined primitive operation then so does evaluation of the source term.

Proposition 8 *If there exists no σ', v', and t' such that $erase(|e|_o(\overline{\lambda}\overline{\sigma}.\overline{\sigma}, \sigma)) \downarrow^{t'} (\sigma', v')$ then there exists no v and t such that $e \downarrow^t v$.*

The safety of the translated program specialized with respect to an initial state σ_0 follows directly from the MIX-equation (1). In our application, there are two compile-time inputs, the incoming program and the initial value for $\overline{\delta}$, and one run-time input, the initial value for σ.

Proposition 9 *Let* `trans` *be the two-level term so that $[\![\texttt{trans}]\!]e = |e|_o$.*

$$[\![[\![\texttt{spec}]\!]\ \texttt{trans}\ e\ (\lambda(\sigma)\sigma, \sigma_0)]\!]\ \vec{y} = erase(|e|_o(\overline{\lambda}\overline{\sigma}.\overline{\sigma}, \sigma_0))\vec{y}$$

The safety of the thus compiled program follows.

Proposition 10 *The compiled program $[\![\texttt{spec}]\!]$ `trans` e $(\lambda(\sigma)\sigma, \sigma_0)$ is safe with respect to $\mathcal{S}'$ (as defined in Sec. 3).*

5 Conclusions

Offline partial evaluation is well-suited to translate programs into safe programs that observe safety policies specified by security automata. We have demonstrated this using a very simple homogeneous approach to have a clear presentation of the fundamental concepts. More run-time checks could be eliminated by considering a number of avenues, for example, by using stronger specialization techniques [34], by allowing for code duplication, or by flow analysis [16].

The techniques shown here are designed to never duplicate code, run efficiently and predictably in linear time, and be applicable with all specialization techniques, *e.g.*, online partial evaluation [38], offline partial evaluation [17], type specialization [14], type-directed partial evaluation [6]. The translations presented in this work have been designed and tested using the PGG system [33], an offline partial evaluator for Scheme.

The results of the translation are comparable to those of Colcombet and Fradet [3]. The results are similar for straight-line code. For programs that involve function calls, their technique should yield better results because our translation does not transfer information across function calls. Basing the translation on the results of a flow analysis or just performing a prepass of function inlining before the translation will improve on that.

We plan to integrate the translation with run-time code generation. The resulting framework will provide just-in-time enforcing compilation and it will serve as a vehicle for experiments with mobile code.

References

[1] A.-R. Adl-Tabatabai, G. Langdale, S. Lucco, and R. Wahbe. Efficient and language-independent mobile programs. In *Proceedings of the ACM SIGPLAN '96 Conference on Programming Language Design and Implementation (PLDI)*, pages 127–136, Philadelphia, Pa., May 1996.

[2] A. W. Appel and A. P. Felty. A semantics model of types and machine instructions for proof-carrying code. In Reps [29], pages 243–253.

[3] T. Colcombet and P. Fradet. Enforcing trace properties by program transformation. In Reps [29], pages 54–66.

[4] C. Consel and O. Danvy. Tutorial notes on partial evaluation. In *Proc. 20th Annual ACM Symposium on Principles of Programming Languages*, pages 493–501, Charleston, South Carolina, Jan. 1993. ACM Press.

[5] C. Consel and S.-C. Khoo. On-line & off-line partial evaluation: Semantic specifications and correctness proofs. Technical report, Yale University, Dept. of Computer Science, Nov. 1993.

[6] O. Danvy. Type-directed partial evaluation. In *Proc. 23rd Annual ACM Symposium on Principles of Programming Languages*, pages 242–257, St. Petersburg, Fla., Jan. 1996. ACM Press.

[7] D. Evans and A. Twyman. Flexible policy-directed code safety. In *IEEE Symposium on Security and Privacy*, Oakland, CA, May 1999.

[8] C. Flanagan, A. Sabry, B. F. Duba, and M. Felleisen. The essence of compiling with continuations. In *Proc. of the ACM SIGPLAN '93 Conference on Programming Language Design and Implementation*, pages 237–247, Albuquerque, New Mexico, June 1993.

[9] R. Glück. On the generation of specializers. *Journal of Functional Programming*, 4(4):499–514, Oct. 1994.

[10] R. Glück and J. Jørgensen. Generating optimizing specializers. In *IEEE International Conference on Computer Languages*, pages 183–194. IEEE Computer Society Press, 1994.

[11] R. Glück and J. Jørgensen. Generating transformers for deforestation and supercompilation. In B. Le Charlier, editor, *Static Analysis*, volume 864 of *Lecture Notes in Computer Science*, pages 432–448. Springer-Verlag, 1994.

[12] J. Hatcliff. Mechanically verifying the correctness of an offline partial evaluator. In D. Swierstra and M. Hermenegildo, editors, *International Symposium on Programming Languages, Implementations, Logics and Programs (PLILP '95)*, volume 982 of *Lecture Notes in Computer Science*, pages 279–298, Utrecht, The Netherlands, Sept. 1995. Springer-Verlag.

[13] J. Hatcliff and O. Danvy. A computational formalization for partial evaluation. *Mathematical Structures in Computer Science*, 7(5):507–542, 1997.

[14] J. Hughes. Type specialisation for the λ-calculus; or, a new paradigm for partial evaluation based on type inference. In O. Danvy, R. Glück, and P. Thiemann, editors, *Partial Evaluation*, volume 1110 of *Lecture Notes in Computer Science*, pages 183–215, Schloß Dagstuhl, Germany, Feb. 1996. Springer-Verlag.

[15] Java2 platform. http://www.javasoft.com/products/, 2000.

[16] S. Jagannathan and A. Wright. Polymorphic splitting: An effective polyvariant flow analysis. *ACM Transactions on Programming Languages and Systems*, 20(1):166–207, Jan. 1998.

[17] N. D. Jones, C. K. Gomard, and P. Sestoft. *Partial Evaluation and Automatic Program Generation*. Prentice-Hall, 1993.

[18] D. Kozen. Language-based security. Technical Report TR99-1751, Cornell University, Computer Science, June 15, 1999.

[19] J. Launchbury and C. K. Holst. Handwriting cogen to avoid problems with static typing. In *Draft Proceedings, Fourth Annual Glasgow Workshop on Functional Programming*, pages 210–218, Skye, Scotland, 1991. Glasgow University.

[20] J. L. Lawall and P. Thiemann. Sound specialization in the presence of computational effects. In *Proc. Theoretical Aspects of Computer Software*, volume 1281 of *Lecture Notes in Computer Science*, pages 165–190, Sendai, Japan, Sept. 1997. Springer-Verlag.

[21] Úlfar Erlingsson and F. B. Schneider. SASI enforcement of security policies: A retrospective. In *Proceedings of the 1999 New Security Paradigms Workshop*, Caledon Hills, Ontario, Canada, Sept. 1999.

[22] T. Lindholm and F. Yellin. *The Java Virtual Machine Specification*. Addison-Wesley, 1996.

[23] S. Lucco, O. Sharp, and R. Wahbe. Omniware: A universal substrate for web programming. *WorldWideWeb Journal*, 1(1), Dec. 1995.

[24] N. G. Michael and A. W. Appel. Machine instruction syntax and semantics in higher order logic. In *17th International Conference on Automated Deduction (CADE-17)*, June 2000.

[25] G. Morrisett, D. Walker, K. Crary, and N. Glew. From system F to typed assembly language. In L. Cardelli, editor, *Proc. 25th Annual ACM Symposium on Principles of Programming Languages*, San Diego, CA, USA, Jan. 1998. ACM Press.

[26] G. C. Necula. Proof-carrying code. In *Proceedings of the 24th ACM Symposium on Principles of Programming Languages*, Paris, France, Jan. 1997.

[27] G. C. Necula and P. Lee. Safe, Untrusted Agents Using Proof-Carrying Code. In G. Vigna, editor, *Mobile Agent Security*, Lecture Notes in Computer Science No. 1419, pages 61–91. Springer-Verlag: Heidelberg, Germany, 1998.

[28] F. Nielson, H. R. Nielson, and C. Hankin. *Principles of Program Analysis.* Springer Verlag, 1999.

[29] T. Reps, editor. *Proc. 27th Annual ACM Symposium on Principles of Programming Languages*, Boston, MA, USA, Jan. 2000. ACM Press.

[30] F. B. Schneider. Enforceable security policies. Technical Report TR99-1759, Cornell University, Ithaca, NY, USA, July 1999.

[31] W. Taha and T. Sheard. Multi-stage programming with explicit annotations. In C. Consel, editor, *Proc. ACM SIGPLAN Symposium on Partial Evaluation and Semantics-Based Program Manipulation PEPM '97*, pages 203–217, Amsterdam, The Netherlands, June 1997. ACM Press.

[32] P. Thiemann. Combinators for program generation. *Journal of Functional Programming*, 9(5):483–525, Sept. 1999.

[33] P. Thiemann. *The PGG System—User Manual.* Universität Freiburg, Freiburg, Germany, Mar. 2000. Available from `http://www.informatik.uni-freiburg.de/proglang/software/pgg/`.

[34] P. Thiemann. Enforcing safety properties using type specialization. In D. Sands, editor, *Proc. 10th European Symposium on Programming*, Lecture Notes in Computer Science, Genova, Italy, Apr. 2001. Springer-Verlag.

[35] V. F. Turchin. Program tranformation with metasystem transitions. *Journal of Functional Programming*, 3(3):283–313, July 1993.

[36] R. Wahbe, S. Lucco, T. E. Anderson, and S. L. Graham. Efficient software-based fault isolation. In *Proceedings of the 14th ACM Symposium on Operating Systems Principles*, pages 203–216, 1993.

[37] D. Walker. A type system for expressive security policies. In Reps [29], pages 254–267.

[38] D. Weise, R. Conybeare, E. Ruf, and S. Seligman. Automatic online partial evaluation. In J. Hughes, editor, *Proc. Functional Programming Languages and Computer Architecture 1991*, volume 523 of *Lecture Notes in Computer Science*, pages 165–191, Cambridge, MA, 1991. Springer-Verlag.