Enforcing Safety Properties Using Type Specialization

Peter Thiemann

Universität Freiburg
thiemann@informatik.uni-freiburg.de

Abstract. Type specialization can serve as a powerful tool in enforcing safety properties on foreign code. Using the specification of a monitoring interpreter, polyvariant type specialization can produce compiled code that is guaranteed to obey a specified safety policy. It propagates a security state at compile-time and generates code for each different security state. The resulting code contains virtually no run-time operations on the security state, at the price of some code duplication. A novel extension of type specialization by intersection types limits the amount of code duplication considerably, thus making the approach practical.

A few years back, mobile code was merely an exciting research subject. Meanwhile, the situation has changed dramatically and mobile code is about to invade our everyday lives. Many applications load parts of their code —or even third-party extension modules— 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) 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, it should be guaranteed that foreign code does 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 this guarantee is to execute the code in a *sand box*. Conceptually, a sand box performs monitored execution. It tracks the execution of foreign code and stops it if it attempts an illegal sequence of actions. A property that can be enforced in this way is called a *safety property*.

Such sand box environments have been conceived and implemented with widely different degrees of sophistication. The obvious approach to such a sand box is to perform monitoring by interpreting the code. However, while the approach is highly flexible it involves a large interpretation 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 prevent unwanted actions. 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 [19].

Related work

The Omniware approach [2, 20, 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 [31] 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 [18] for x86-assembly language and for JVM bytecode. Both allow for a separate specification of a state automaton and rely on an ad-hoc code transformation to integrate the propagation of the state with the execution of the program.

Evans and Twyman [9] have constructed an impressive 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 [23–26] 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 [3, 21].

Kozen [17] 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) [22] 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 [5] 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 [10,35] and put into practice by Glück, Jørgensen, and others [11,12,32,34]. The present work can also be phrased in the framework of semantic modifiers [1].

Contributions

The present work demonstrates that previous ad-hoc approaches to enforcing safety properties by program transformation can be expressed uniformly using partial evaluation. This simplifies their theoretical development and their implementation considerably since partial evaluation technology is reused.

After introducing the source language, security automata, and type specialization in Sec. 1, we start with a naive implementation of monitored execution using an instrumented interpreter for a simply-typed call-by-value lambda calculus (§ 2).

In Section 3, we define a translation into a two-level lambda calculus. Type specialization [13] of the resulting two-level terms can remove (in certain cases) all run-time operations on the security state. Specialization creates variants of user code tailored to particular security states. They must be drawn from a finite set for our approach to work.

In Section 4, we introduce a novel extension of type specialization by intersection types and subtyping. It avoids unnecessary code duplication, thus making our approach practical. Our prototype implementation automatically performs all example optimizations from Walker's paper [37].

The main technical results are the correctness proofs of the translation and the non-standard compilation performed by type specialization. They guarantee the safety of the translated code and of the compiled code. In addition, we have proved correct our extension of type specialization, which amounts to proving subject reduction [14].

1 Prerequisites

1.1 Source language

The source language is a simply-typed call-by-value lambda calculus with constants, conditionals, and primitive operations on base types (see Fig. 1). Each primitive operation, `O`, can change the current security state. The value of $\mathtt{fix}\ x_0(x_1 \dots x_n)e$ is a recursively defined function. Write $\lambda(x_1 \dots x_n)e$ if x_0 does not appear in e, and $\mathtt{let}\ x = e_1\ \mathtt{in}\ e_2$ for $(\lambda(x)e_2)@(e_1)$. The typing rules defining the judgement $\Gamma \vdash e : \tau$ are standard.

Each primitive operation, $\mathtt{O} : \mathrm{BaseType}^n \to \mathrm{BaseType}$, comes with a partial semantic function $[\![\mathtt{O}]\!] \in \mathrm{BaseType}^n \hookrightarrow \mathrm{BaseType}$ and a total state transition function, $\delta : \mathit{Op} \to \Sigma \times \mathrm{BaseType}^n \to \Sigma$, which models the change of the (security-) state on application of the operation. The semantics of the language is given in structural operational style. It maps a pair of a (security-) state, σ, and a closed term to a new state and closed term.

Each reduction sequence $\sigma_0, e_0 \to \sigma_1, e_1, \to \dots$ gives rise to a potentially infinite sequence $\boldsymbol{\sigma} = (\sigma_0, \sigma_1, \dots)$ of states (a trace). Write $\sigma_0, e_0 \downarrow \sigma', v$ if there is a finite sequence of reductions, $\sigma_0, e_0 \to \sigma_1, e_1 \to \dots \to \sigma', v$.

Eta-value conversion is the reflexive, transitive, symmetric, and compatible closure of eta-value reduction: $\mathtt{fix}\ x_0(x_1, \dots, x_n)v@(x_1, \dots, x_n) \to_{\eta v} v$ where $x_0, x_1, \dots, x_n$ are distinct variables not occurring free in v.

Syntax

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

Operational semantics

$$
\begin{array}{ll}
\sigma, (\texttt{if true}\ e_1\ e_2) & \to \sigma, e_1 \\
\sigma, (\texttt{if false}\ e_1\ e_2) & \to \sigma, e_2 \\
\sigma, \texttt{O}(a_1 \ldots a_n) & \to \delta(\texttt{O})(\sigma, a_1 \ldots a_n), v \\
& \quad \text{if } v = [\![\texttt{O}]\!](a_1, \ldots, a_n) \text{ is defined} \\
\sigma, (\texttt{fix}\ x_0(x_1 \ldots x_n)e)@(v_1 \ldots v_n) \to & \\
\qquad \sigma, e[x_0 \mapsto \texttt{fix}\ x_0(x_1 \ldots x_n)e, x_1 \mapsto v_1, \ldots, x_n \mapsto v_n] &
\end{array}
$$

$$
\frac{\sigma, e \to \sigma', e'}{\sigma, C[e] \to \sigma', C[e']}
$$

Fig. 1. The source language

1.2 Security automata

A safety policy is a set of finite and infinite traces that obeys certain restrictions [31]. A reduction sequence is acceptable if its associated trace is contained in the policy. Schneider [31] 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 $\mathcal{S} = (\Sigma, Op, Value, \delta, \sigma_0, bad)$ [37] where

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

A closed term e_0 is *safe* with respect to $\mathcal{S}$ and some $\sigma_0 \in \Sigma \setminus \{bad\}$ if either there exist $\sigma' \in \Sigma$ and $v \in Value$ such that $\sigma_0, e_0 \downarrow \sigma', v$ and $\sigma' \neq bad$ or the trace of σ_0, e_0 is infinite. 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 (Fig. 2) that no network `send` operation happens after a `read` operation from a local file. The transition functions are the identity functions for all primitive operations except `send` and `read`.

Set of states: $\Sigma = \{$*before-read*, *after-read*, *bad*$\}$
Initial state: $\sigma_0 =$ *before-read*

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

Fig. 2. Transition functions

The program $(\lambda(x)\texttt{read}(\mathit{file}))@(\texttt{send}(\mathit{data}))$ is safe (with respect to σ_0) due to the trace (*before-read*, *before-read*, *after-read*). It is not safe with respect to *after-read*: the corresponding trace is (*after-read*, *bad*, *bad*).

The program $(\lambda(x)\texttt{send}(\mathit{data}))@(\texttt{read}(\mathit{file}))$ is not safe with respect to any state: it generates the unacceptable traces (*before-read*, *after-read*, *bad*) and (*after-read*, *after-read*, *bad*).

1.3 Type specialization

Traditional partial evaluation techniques [16] rely on non-standard interpretation or evaluation of a source program to perform as many operations on compile-time data as possible. They incur certain limitations because they propagate compile-time data using compile-time values. Once a traditional specializer generates a specialized expression, it loses all further information about it. This leads to the *well-formedness restriction* in binding-time analysis: if a structured value is run-time then all its components are run-time, too. For example, if a function is classified as a run-time value, then so are its arguments and results.

Type specialization [13] is fundamentally different because it propagates compile-time data using types. A type specializer transforms a source expression into a specialized expression *and* its specialized type. The type contains all the compile-time information. If all information from a source term can be processed at compile time then the specialized expression becomes trivial, indicated by $\bullet$, and can be discarded. In particular, there is no well-formedness restriction: since specialization is performed using type inference, compile-time and run-time data may be arbitrarily mixed.

Figure 3 defines type specialization as a judgement $\Gamma \vdash e \rightsquigarrow e' : \tau'$, that is, in typing context Γ the two-level term e specializes to specialized term e' with specialized type τ'. In a two-level term, constants are always compile-time values, variables may be bound to compile-time or run-time values, `lift` converts a compile-time constant into a run-time constant, and `poly` and `spec` control polyvariance (see below). The operation $e_1 \texttt{+} e_2$ is an example primitive operation. For simplicity, we formalize only single-argument functions.

Syntax of two-level language

$$
\begin{array}{lll}
\text{Terms} & e & ::= x \mid n \mid e\overline{+}e \mid \overline{\texttt{if}}\ e\ \overline{\texttt{then}}\ e\ \overline{\texttt{else}}\ e \mid \overline{\texttt{fix}}\ x(x)e \mid e\overline{@}e \mid \texttt{poly}\ \ e \mid \\
 & & \quad\ \texttt{lift}\ e \mid e\underline{+}e \mid \underline{\texttt{if}}\ e\ \underline{\texttt{then}}\ e\ \underline{\texttt{else}}\ e \mid \underline{\texttt{fix}}\ x(x)e \mid e\underline{@}e \mid \texttt{spec}\ \ e \\
\text{Specialized terms} & e' & ::= \bullet \mid x \mid n \mid e'\texttt{+}e' \mid \texttt{if}\ \ e'\ \ \texttt{then}\ \ e'\ \ \texttt{else}\ \ e' \mid \\
 & & \quad\ e'@e' \mid \texttt{fix}\ x(x)e' \mid (e',\ldots,e') \mid \pi_i(e') \\
\text{Specialized types} & \tau' & ::= S\{n\} \mid \mathit{Int} \mid \tau' \to \tau' \mid \tau'\texttt{+}\tau' \mid \tau' \times \ldots \times \tau' \\
\text{Typing contexts} & \Gamma & ::= \emptyset \mid \Gamma, x \rightsquigarrow e' : \tau'
\end{array}
$$

Equality on specialized types

$$
\tau' = \tau' \qquad \frac{\tau_1' = \tau_2' \quad \tau_2' = \tau_3'}{\tau_1' = \tau_3'} \qquad \frac{\tau_1' = \tau_2'}{\tau_2' = \tau_1'} \qquad \frac{\tau_1' = \tau_2' \quad \tau_3' = \tau_4'}{\tau_1' \to \tau_3' = \tau_2' \to \tau_4'}
$$

$$
\frac{\tau_1' = \tau_2' \quad \tau_3' = \tau_4'}{\tau_1'\texttt{+}\tau_3' = \tau_2'\texttt{+}\tau_4'} \qquad S\{n_1\}\texttt{+}S\{n_2\} = S\{n_1 + n_2\}
$$

Inference rules of type specialization

$$
\Gamma, x \rightsquigarrow e' : \tau', \Gamma' \vdash x \rightsquigarrow e' : \tau' \qquad \Gamma \vdash n \rightsquigarrow \bullet : S\{n\} \qquad \frac{\Gamma \vdash e \rightsquigarrow e' : S\{n\}}{\Gamma \vdash \texttt{lift}\ e \rightsquigarrow n : \mathit{Int}}
$$

$$
\frac{\begin{array}{c}\Gamma \vdash e_1 \rightsquigarrow e_1' : \tau_1' \\ \Gamma \vdash e_2 \rightsquigarrow e_2' : \tau_2'\end{array}}{\Gamma \vdash e_1\overline{+}e_2 \rightsquigarrow \bullet : \tau_1'\texttt{+}\tau_2'} \qquad \frac{\begin{array}{c}\Gamma \vdash e_1 \rightsquigarrow e_1' : \mathit{Int} \\ \Gamma \vdash e_2 \rightsquigarrow e_2' : \mathit{Int}\end{array}}{\Gamma \vdash e_1\underline{+}e_2 \rightsquigarrow e_1'\texttt{+}e_2' : \mathit{Int}}
$$

$$
\frac{\Gamma \vdash e_0 \rightsquigarrow e_0' : S\{0\} \quad \Gamma \vdash e_1 \rightsquigarrow e_1' : \tau'}{\Gamma \vdash \overline{\texttt{if}}\ e_0\ \overline{\texttt{then}}\ e_1\ \overline{\texttt{else}}\ e_2 \rightsquigarrow e_1' : \tau'} \qquad \frac{\Gamma \vdash e_0 \rightsquigarrow e_0' : S\{1\} \quad \Gamma \vdash e_2 \rightsquigarrow e_2' : \tau'}{\Gamma \vdash \overline{\texttt{if}}\ e_0\ \overline{\texttt{then}}\ e_1\ \overline{\texttt{else}}\ e_2 \rightsquigarrow e_2' : \tau'}
$$

$$
\frac{\Gamma \vdash e_0 \rightsquigarrow e_0' : \mathit{Int} \quad \Gamma \vdash e_1 \rightsquigarrow e_1' : \tau' \quad \Gamma \vdash e_2 \rightsquigarrow e_2' : \tau'}{\Gamma \vdash \underline{\texttt{if}}\ e_0\ \underline{\texttt{then}}\ e_1\ \underline{\texttt{else}}\ e_2 \rightsquigarrow \texttt{if}\ \ e_0'\ \ \texttt{then}\ \ e_1'\ \ \texttt{else}\ \ e_2' : \tau'}
$$

$$
\frac{\Gamma, x_0 \rightsquigarrow x_0' : \tau_2' \to \tau_1', x_1 \rightsquigarrow x_1' : \tau_2' \vdash e \rightsquigarrow e' : \tau_1'}{\Gamma \vdash \underline{\texttt{fix}}\ x_0(x_1)e \rightsquigarrow \texttt{fix}\ x_0'(x_1')e' : \tau_2' \to \tau_1'}
$$

$$
\frac{\Gamma \vdash e_1 \rightsquigarrow e_1' : \tau_2' \to \tau_1' \quad \Gamma \vdash e_2 \rightsquigarrow e_2' : \tau_2'}{\Gamma \vdash e_1\underline{@}e_2 \rightsquigarrow e_1'@e_2' : \tau_1'}
$$

$$
\frac{\Gamma \vdash e \rightsquigarrow e' : \tau_1' \quad \tau_1' = \tau_2'}{\Gamma \vdash e \rightsquigarrow e' : \tau_2'}
$$

Fig. 3. Standard type specialization

Here is an example specialization of the term $(\underline{\lambda} x.\texttt{lift}\ x)\underline{@}4$:

$$
\frac{\dfrac{\dfrac{x \rightsquigarrow x' : S\{4\} \vdash x \rightsquigarrow x' : S\{4\}}{x \rightsquigarrow x' : S\{4\} \vdash \texttt{lift}\ x \rightsquigarrow 4 : \mathit{Int}}}{\emptyset \vdash \underline{\lambda} x.\texttt{lift}\ x \rightsquigarrow \lambda x'.4 : S\{4\} \to \mathit{Int}} \qquad \emptyset \vdash 4 \rightsquigarrow \bullet : S\{4\}}{\emptyset \vdash (\underline{\lambda} x.\texttt{lift}\ x)\underline{@}4 \rightsquigarrow (\lambda x'.4)@\bullet : \mathit{Int}}
$$

The typing expresses the compile-time value 4 as a singleton type, $S\{4\}$.

There are two significant changes with respect to Hughes's presentation [13]. First, Hughes's two-level terms obey a simple type discipline. It ensures that

$$\frac{(\forall 1 \leq i \leq n)\Gamma \vdash e \leadsto e'_i : \tau'_i}{\Gamma \vdash \texttt{poly}\ e \leadsto (e'_1, \ldots, e'_n) : \tau'_1 \times \ldots \times \tau'_n} \qquad \frac{\Gamma \vdash e \leadsto e' : \tau'_1 \times \ldots \times \tau'_n}{\Gamma \vdash \texttt{spec}\ e \leadsto \pi_i(e') : \tau'_i}$$

Fig. 4. Type specialization rules for `poly` and `spec`

Reduction for two-level terms

$$\begin{array}{ll}
n_1 \overline{+} n_2 & \rightarrow_{tt} (n_1 + n_2) \\
\overline{\texttt{if}}\ 0\ \overline{\texttt{then}}\ e_1\ \overline{\texttt{else}}\ e_2 & \rightarrow_{tt} e_1 \\
\overline{\texttt{if}}\ 1\ \overline{\texttt{then}}\ e_1\ \overline{\texttt{else}}\ e_2 & \rightarrow_{tt} e_2 \\
(\overline{\texttt{fix}}\ f(x)e_1)\overline{@}e_2 & \rightarrow_{tt} e_1[f \mapsto \overline{\texttt{fix}}\ f(x)e_1, x \mapsto e_2] \\
\texttt{spec}\ (\texttt{poly}\ e) & \rightarrow_{tt} e \\
\texttt{lift}\ n_1 \underline{+} \texttt{lift}\ n_2 & \rightarrow_{tt} \texttt{lift}\ (n_1 + n_2) \\
\underline{\texttt{if}}\ 0\ \underline{\texttt{then}}\ e_1\ \underline{\texttt{else}}\ e_2 & \rightarrow_{tt} e_1 \\
\underline{\texttt{if}}\ 1\ \underline{\texttt{then}}\ e_1\ \underline{\texttt{else}}\ e_2 & \rightarrow_{tt} e_2 \\
(\underline{\texttt{fix}}\ x_0(x_1)e_1)\underline{@}e_2 & \rightarrow_{tt} e_1[x_0 \mapsto \underline{\texttt{fix}}\ x_0(x_1)e_1, x_1 \mapsto e_2]
\end{array}$$

Reduction for specialized terms

$$\begin{array}{ll}
n_1 \texttt{+} n_2 & \rightarrow_{sp} (n_1 + n_2) \\
\texttt{if}\ 0\ \texttt{then}\ e'_1\ \texttt{else}\ e'_2 & \rightarrow_{sp} e'_1 \\
\texttt{if}\ 1\ \texttt{then}\ e'_1\ \texttt{else}\ e'_2 & \rightarrow_{sp} e'_2 \\
(\texttt{fix}\ x_0(x_1)e'_1)@e'_2 & \rightarrow_{sp} e'_1[x_0 \mapsto \texttt{fix}\ x_0(x_1)e'_1, x_1 \mapsto e'_2] \\
\pi_i(e'_1, \ldots, e'_n) & \rightarrow_{sp} e'_i
\end{array}$$

Fig. 5. Notions of reduction

the specializer never confuses compile-time and run-time values. However, it does not guarantee that the two-level term specializes successfully. Moreover, the specializer discovers errors of this kind anyway while inferring specialized types. Therefore, we have dropped this set of typing rules.

Second, Hughes's presentation hardwires the processing of singleton types into the rule for compile-time addition. Instead, we have formalized compile-time addition through conversion rules for singleton types. This choice simplifies the specification of extensions considerably, as demonstrated in Sec. 4.2.

For brevity, our formalization does not include compile-time functions, which are expanded at compile-time before their specialized type is inferred. Their addition is exactly as in Hughes's work [13,14] and is orthogonal to the problems discussed in the present paper.

The `poly` and `spec` constructs [13] introduce and eliminate polyvariant values. A polyvariant value is a set of specialized terms indexed by their specialized types. The type specializer employs a numeric encoding of the index in its output. It implements the rules using backtracking.

$$
\begin{aligned}
||\mathrm{BaseType}|| &= \mathrm{BaseType} \\
||(\tau_1, \ldots, \tau_n) \to \tau|| &= (\Sigma, ||\tau_1||, \ldots, ||\tau_n||, (\Sigma, ||\tau||) \to \mathtt{Ans}) \to \mathtt{Ans} \\
|\tau| &= (\Sigma, (\Sigma, ||\tau||) \to \mathtt{Ans}) \to \mathtt{Ans} \\
||\emptyset|| &= \emptyset \\
||\Gamma, x : \tau|| &= ||\Gamma||, x : ||\tau||
\end{aligned}
$$

$$
\begin{aligned}
||x|| &= x \\
||a|| &= a \\
||\mathtt{fix}\ x_0(x_1 \ldots x_n)e|| &= \mathtt{fix}\ x_0(\sigma, x_1, \ldots, x_n, x_{n+1})|e|(\sigma, x_{n+1})
\end{aligned}
$$

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

Fig. 6. Translation that enforces a security policy

Hughes [14] has proved the correctness of type specialization (including Fig. 4) by specifying two reduction relations, one for two-level terms, $\to_{tt}$, and one for specialized terms, $\to_{sp}$, (see Fig. 5) and then proving a result like this:

Proposition 1 (Simulation). *If* $\Gamma \vdash e_1 \rightsquigarrow e_1' : \tau'$ *and* $e_1 \to_{tt} e_2$ *then there exists* e_2' *such that* $\Gamma \vdash e_2 \rightsquigarrow e_2' : \tau'$ *and* $e_1' \overset{*}{\to}_{sp} e_2'$.

As in Hughes's paper [14], the proof relies on a number of substitution lemmas (see Section 4.2), which are all easy to prove.

2 Enforcing a policy by interpretation

A simple way to enforce safe execution is to incorporate a security automaton into an interpreter. An equivalent point of view is a translation that makes the threading of the security state explicit. Before attempting a primitive operation, a translated program steps the security state and checks whether the result is *bad*.

Figure 6 shows a translation to continuation-passing and state-passing style [30], augmented by stepping and testing of the security state. The translation makes explicit the flow of control and of the current security state. Using `Ans` as the type of answers that can be extracted from a program, the translation acts on types as follows.

Proposition 2. *If* $\Gamma \vdash e : \tau$ *then* $||\Gamma|| \vdash |e| : |\tau|$.

The translated program never violates the security policy if the operations $\delta(\mathtt{O})$ on the explicit state do not affect the state component in the operational semantics. Formally, let $\mathcal{S}' = (\Sigma, Op', Value, \delta', \sigma_0, bad)$ with $Op' = Op \cup \{\delta(\mathtt{O}) \mid \mathtt{O} \in Op\} \cup \{\mathtt{halt}\}$ (regarding $\delta(\mathtt{O})$ as the name of a new primitive) and, for all $\mathtt{O} \in Op$, $\delta'(\mathtt{O}) = \delta(\mathtt{O})$ and $\delta'(\delta(\mathtt{O}))(v_\sigma, v_1 \ldots v_n) = v_\sigma$. Let $[\![\mathtt{halt}]\!]() = a$, a fixed constant signalling an error.

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

Proposition 3. *If* $\sigma, |e|(\sigma, \lambda(\sigma, y)y) \downarrow \sigma', v'$ *then* $\sigma' \neq bad$.

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

Proposition 4. *Suppose* $\sigma, e \downarrow \sigma', v$. *If* $\sigma' \neq bad$ *then* $\sigma, |e|(\sigma, \lambda(\sigma, y)y) \downarrow \sigma', ||v||$.

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 5. *If there exist no* σ' *and* v' *such that* $\sigma, |e|(\sigma, \lambda(\sigma, y)y) \downarrow \sigma', v'$ *then there exist no* σ' *and* v' *such that* $\sigma, e \downarrow \sigma', v'$.

Using this naive translation yields inefficient programs because every use of a primitive operation is preceded by a run-time check of the security state.

3 Compiling policies by type specialization

To submit the translation to a specializer, we retarget it to a two-level language, indicating compile-time by overlining and run-time by underlining. Type specialization [13] of the translated terms can remove the state component, σ, and the corresponding run-time checks completely, in certain cases.

We consider the two-level translation as an interpreter and specialize it with respect to a source program. The specialized program can be shown to be safe in two steps: Prove that translated programs are safe, and appeal to the correctness of the specializer (Prop. 1) to see that the specialized programs are safe.

We conclude with a discussion of alternative specialization methods.

3.1 First steps

Specialization potentially generates code variants for each different security state. Hence, it is only applicable if the set of states is finite. For further simplification, we initially assume that the transition function *does not* depend on the arguments but only on the name of the primitives. Hence, the compile-time transition function, $\overline{\delta}$, is well-defined and gives the full information:

- $\overline{\delta}(\mathtt{O})(\overline{\sigma}) := \overline{\sigma}'$ if $\forall y_1 \ldots y_n . \delta(\mathtt{O})(\sigma, y_1, \ldots, y_n) = \sigma'$,
- $\overline{\delta}(\mathtt{O})(\overline{\sigma}) := bad$ if $\forall \sigma' . \exists y_1 \ldots y_n . \delta(\mathtt{O})(\sigma, y_1, \ldots, y_n) \neq \sigma'$.

$$
\begin{aligned}
||\mathrm{BaseType}||_e &= \underline{\mathrm{BaseType}} \\
||(\tau_1, \ldots, \tau_n) \rightarrow \tau||_e &= \underline{\mathtt{poly}}\ (\overline{\Sigma}, ||\tau_1||_e, \ldots, ||\tau_n||_e, \mathtt{poly}\ (\overline{\Sigma}, ||\tau||_e)\underline{\rightarrow \mathtt{Ans}})\underline{\rightarrow \mathtt{Ans}} \\
|\tau|_e &= (\overline{\Sigma}, (\overline{\Sigma}, ||\tau||_e)\overline{\rightarrow}\underline{\mathtt{Ans}})\overline{\rightarrow}\underline{\mathtt{Ans}} \\
||\emptyset||_e &= \emptyset \\
||\Gamma, x : \tau||_e &= ||\Gamma||_e, x : ||\tau||_e
\end{aligned}
$$

$$
\begin{aligned}
||x||_e &= x \\
||a||_e &= a \\
||\mathtt{fix}\ x_0(x_1 \ldots x_n)e||_e &= \mathtt{poly}\ \underline{\mathtt{fix}}\ x_0(\overline{\sigma}, x_1, \ldots, x_n, x_{n+1}) \\
&\quad\ |e|_e\overline{@}(\overline{\sigma}, \overline{\lambda}(\overline{\sigma}, y).\mathtt{spec}\ \ x_{n+1}\underline{@}(\overline{\sigma}, y)) \\
|v|_e(\overline{\sigma}, c) &= c\overline{@}(\overline{\sigma}, ||v||_e) \\
|(\mathtt{if}\ \ e_1\ \ e_2\ \ e_3)|_e(\overline{\sigma}, c) &= |e_1|_e\overline{@}(\overline{\sigma}, \overline{\lambda}(\overline{\sigma}_1, y_1). \\
&\quad\ \underline{\mathtt{if}}\ y_1\ \underline{\mathtt{then}}\ |e_2|_e\overline{@}(\overline{\sigma}_1, c)\ \underline{\mathtt{else}}\ |e_3|_e\overline{@}(\overline{\sigma}_1, c)) \\
|\mathtt{O}(e_1 \ldots e_n)|_e(\overline{\sigma}_0, c) &= |e_1|_e(\overline{\sigma}_0, \overline{\lambda}(\overline{\sigma}_1, y_1). \ldots |e_n|_e(\overline{\sigma}_{n-1}, \overline{\lambda}(\overline{\sigma}_n, y_n). \\
&\quad\ \overline{\mathtt{let}}\ \overline{\sigma}' = \overline{\delta}(\mathtt{O})(\overline{\sigma}_n)\ \overline{\mathtt{in}} \\
&\quad\ \overline{\mathtt{if}}\ \overline{\sigma}' = \mathit{bad}\ \overline{\mathtt{then}}\ \underline{\mathtt{halt}()}\ \overline{\mathtt{else}} \\
&\quad\ \underline{\mathtt{let}}\ y = \underline{O}(y_1, \ldots, y_n)\ \underline{\mathtt{in}}\ c\overline{@}(\overline{\sigma}', y)) \ldots) \\
|e_0@(e_1 \ldots e_n)|_e(\overline{\sigma}, c) &= |e_0|_e\overline{@}(\overline{\sigma}, \overline{\lambda}(\overline{\sigma}_0, y_0). \\
&\quad\ |e_1|_e\overline{@}(\overline{\sigma}_0, \overline{\lambda}(\overline{\sigma}_1, y_1). \ldots |e_n|_e\overline{@}(\overline{\sigma}_{n-1}, \overline{\lambda}(\overline{\sigma}_n, y_n). \\
&\quad\ \mathtt{spec}\ \ y_0\underline{@}(\overline{\sigma}_n, y_1, \ldots, y_n, \mathtt{poly}\ \underline{\lambda}(\overline{\sigma}, y).c\overline{@}(\overline{\sigma}, y))) \ldots))
\end{aligned}
$$

Fig. 7. Two-level translation

Hence, the state becomes a compile-time value and all operations thereon can be computed at compile-time. Figure 7 defines the translation. It follows the basic strategy of Danvy and Filinski's one-pass translation to continuation-passing style [7, 8]. It avoids introducing administrative redexes by keeping the continuation a compile-time object. This involves converting compile-time continuations to run-time ones, and vice versa, using eta-value expansion. The relevant terms are in the translation of `fix` and application: $\underline{\lambda}(\overline{\sigma}, y).c\overline{@}(\overline{\sigma}, y)$ converts the compile-time continuation c to a run-time value and $\overline{\lambda}(\overline{\sigma}, y).\mathtt{spec}\ \ x_{n+1}\underline{@}(\overline{\sigma}, y)$ converts the run-time continuation x_{n+1} into a compile-time one.

Both the terms for `fix` and for application contain subterms of the form $\underline{\lambda}(\overline{\sigma}, x, \ldots). \ldots$ where a run-time function has a compile-time parameter, $\overline{\sigma}$. This violates the well-formedness restriction of traditional partial evaluation [16] and is the motivation for using type specialization altogether.

3.2 Polyvariance matters

To see, why the `poly` and `spec` annotations in the translation are required, consider a simple example term

$$\lambda(\mathit{file})\mathtt{read}(\mathit{file}) \tag{3}$$

$$\begin{array}{ll}
\underline{\lambda}(\sigma, \mathit{file}, c). & \\
\quad \overline{\mathtt{let}}\ \sigma' = \overline{\delta}(\mathtt{read})(\sigma)\ \overline{\mathtt{in}} & \\
\quad \overline{\mathtt{if}}\ \overline{=}(\sigma', \mathit{bad})\ \overline{\mathtt{then}}\ \underline{\mathtt{HALT}}()\ \overline{\mathtt{else}} & (1)\\
\quad \underline{\mathtt{let}}\ y_1 = \underline{\mathtt{read}}(\mathit{file})\ \underline{\mathtt{in}} & \\
\quad c\underline{@}(\sigma', y_1) &
\end{array}
\qquad
\begin{array}{ll}
\mathtt{poly}\ \underline{\lambda}(\sigma, \mathit{file}, c). & \\
\quad \overline{\mathtt{let}}\ \sigma' = \overline{\delta}(\mathtt{read})(\sigma)\ \overline{\mathtt{in}} & \\
\quad \overline{\mathtt{if}}\ \overline{=}(\sigma', \mathit{bad})\ \overline{\mathtt{then}}\ \underline{\mathtt{HALT}}()\ \overline{\mathtt{else}} & (2)\\
\quad \underline{\mathtt{let}}\ y_1 = \underline{\mathtt{read}}(\mathit{file})\ \underline{\mathtt{in}} & \\
\quad \underline{\mathtt{spec}}\ (c)\underline{@}(\sigma', y_1) &
\end{array}$$

Fig. 8. Translated example

and its translation (1) in Fig. 8. It has specialized type

$$(S\{\mathit{before\text{-}read}\}, \text{BaseType}, (S\{\mathit{after\text{-}read}\}, \text{BaseType})\underline{\rightarrow}\mathtt{Ans})\underline{\rightarrow}\mathtt{Ans}$$

when called in state *before-read* and type

$$(S\{\mathit{after\text{-}read}\}, \text{BaseType}, (S\{\mathit{after\text{-}read}\}, \text{BaseType})\underline{\rightarrow}\mathtt{Ans})\underline{\rightarrow}\mathtt{Ans}$$

when called in state *after-read.* Since $S\{\mathit{before\text{-}read}\} \neq S\{\mathit{after\text{-}read}\}$ and the type specializer is based on simple types, the function cannot be used at both types at once.

To overcome this restriction, Hughes introduced polyvariance. A polyvariant expression gives rise to a tuple of specializations, one for every different type of use. Hence the translation uses $\mathtt{poly}\ \underline{\lambda}(\overline{\sigma}, x, \ldots). \ldots$ which has specialized type $((S\{\sigma_1\}, \text{BaseType}, \ldots) \rightarrow \tau_1') \times \ldots \times ((S\{\sigma_n\}, \text{BaseType}, \ldots) \rightarrow \tau_n')$, for distinct $\sigma_1, \ldots, \sigma_n$. The set $\{\sigma_1, \ldots, \sigma_n\}$ contains only those states that actually reach a use of the polyvariant type. The specializer determines this set dynamically during specialization. A term `spec` ... indicates an elimination point for a tuple introduced by `poly` . It selects a component of the tuple, based on the required type (*i.e.*, the state at the elimination point).

Using `poly` in the translation of term (3) yields term (2) in Fig. 8 with specialized type

$$\begin{array}{l}
((S\{\mathit{before\text{-}read}\}, \text{BaseType}, (S\{\mathit{after\text{-}read}\}, \text{BaseType})\underline{\rightarrow}\mathtt{Ans})\underline{\rightarrow}\mathtt{Ans})\\
\times\ ((S\{\mathit{after\text{-}read}\}, \text{BaseType}, (S\{\mathit{after\text{-}read}\}, \text{BaseType})\underline{\rightarrow}\mathtt{Ans})\underline{\rightarrow}\mathtt{Ans})
\end{array} \qquad (4)$$

and specialized code typing

$$\begin{array}{l}
(\lambda(\mathit{file}, c)\mathtt{let}\ y_1 = \mathtt{read}(\mathit{file})\ \mathtt{in}\ c@(y_1)\\
,\lambda((\mathit{file}, c))\mathtt{let}\ y_1 = \mathtt{read}(\mathit{file})\ \mathtt{in}\ c@((y_1))).
\end{array} \qquad (5)$$

3.3 Properties of the translation

The translation preserves typing.

Proposition 6. *If* $\Gamma \vdash e : \tau$ *then* $||\Gamma||_e \vdash |e|_e : |\tau|_e$.

We state the relation to the naive translation (Fig. 6) using the function *erase*(). It maps a two-level term to a standard term by erasing all overlining and underlining annotations as well as *erase*(`lift` e) = *erase*(e), *erase*(`poly` e) = *erase*(e), and *erase*(`spec` e) = *erase*(e).

$$
\begin{aligned}
&|\mathsf{O}(e_1 \ldots e_n)|'_1(\overline{\sigma}_0, c)\\
&= |e_1|'_1(\overline{\sigma}_0, \overline{\lambda}(\overline{\sigma}_1, y_1). \ldots |e_n|'_1(\overline{\sigma}_{n-1}, \overline{\lambda}(\overline{\sigma}_n, y_n).\\
&\quad \overline{\texttt{let}}\ \overline{\sigma}' = \overline{\delta}(\mathsf{O})(\overline{\sigma}_n)\ \overline{\texttt{in}}\\
&\quad \overline{\texttt{if}}\ \overline{\sigma}' \neq bad\ \overline{\texttt{then}}\ c\overline{@}(\overline{\sigma}', \underline{O}(y_1, \ldots, y_n))\ \overline{\texttt{else}}\\
&\quad \underline{\texttt{let}}\ \underline{\sigma} = \underline{\delta}(\mathsf{O})(\texttt{lift}\ \overline{\sigma}_n, y_1, \ldots, y_n)\ \underline{\texttt{in}}\\
&\quad \underline{\texttt{if}}\ \underline{\sigma = }\texttt{lift}\ bad\ \underline{\texttt{then}}\ \underline{\texttt{halt}}()\ \underline{\texttt{else}}\\
&\quad \overline{\texttt{let}}\ \{\overline{\sigma}'_1, \ldots, \overline{\sigma}'_r\} = \overline{\Delta(\mathsf{O})(\overline{\sigma}')}\ \overline{\texttt{in}}\\
&\quad \underline{\texttt{let}}\ y = \underline{O}(y_1, \ldots, y_n)\ \underline{\texttt{in}}\\
&\quad \underline{\texttt{if}}\ \underline{\sigma = }\texttt{lift}\ \overline{\sigma}'_1\ \underline{\texttt{then}}\ c\overline{@}(\overline{\sigma}'_1, y)\ \underline{\texttt{else}}\\
&\quad \underline{\texttt{if}}\ \underline{\sigma = }\texttt{lift}\ \overline{\sigma}'_2\ \underline{\texttt{then}}\ c\overline{@}(\overline{\sigma}'_2, y)\ \underline{\texttt{else}}\\
&\quad \ldots c\overline{@}(\overline{\sigma}'_r, y)) \ldots)
\end{aligned}
$$

Fig. 9. Revised heterogeneous treatment of primitive operators

Proposition 7. *$\sigma, |e|(\sigma, \lambda(\sigma, y)y) \downarrow \sigma', ||v||$ if and only if σ, $erase(|e|_e)(\sigma, \lambda(\sigma, y)y) \downarrow \sigma'$, $erase(||v||_e)$.*

To relate to the compiled/specialized program, we invoke the correctness of the underlying specializer and conclude the safety of the compiled program.

Proposition 8. *Suppose $\emptyset \vdash$ `trans` e $(\overline{\sigma}_0, \overline{\lambda}(\overline{\sigma}, y).y) \rightsquigarrow e' : \tau'$ where `trans` is the program text defining $|\ |_e$. The compiled program e' is safe with respect to $\sigma_0 \in \Sigma$.*

Technically, Hughes's proof only applies to a call-by-name lambda calculus. This does not pose problems in our case, because we are only specializing programs in continuation-passing style.

3.4 Achieving generality

Up to now, our treatment is limited to policies where the state transition function does not depend on the arguments to the primitives. This restriction is not essential. The revised treatment of primitive operators in Figure 9 may be used with arbitrary state transition functions.

The code first evaluates and checks the arguments of the operation. If it can predict a potential security violation from the pre-computed security state, $\overline{\sigma'}$, then it generates a run-time test using an implementation, $\underline{\delta}$, of the state transition function applied to the run-time constant, `lift` $\overline{\sigma}_n$, and the actual arguments. The resulting run-time security state, $\underline{\sigma}$, is tested against *bad* at run-time.

Finally, it extracts a compile-time state from $\underline{\sigma}$ using

$$\Delta(\mathsf{O})(\sigma) = \{\delta(\mathsf{O})(\sigma, y_1, \ldots, y_n) \mid y_1, \ldots, y_n \in Base\} \setminus \{bad\}$$

to estimate the set of possible non-*bad* outcomes of the run-time state transition $\delta(\mathsf{O})$ on the compile-time state $\overline{\sigma}$. Using this set, the code recovers the compile-time value from the run-time outcome of the state transition by testing the latter against all possible values and using the compile-time value in the continuation. This is essentially "The Trick" [16], a standard binding-time improving transformation. It is a further source of code duplication because the continuation c is processed for each possible outcome.

3.5 Discussion

The reader might be tempted to think that traditional partial evaluation tricks (like lifting, polyvariant binding times, etc) might be applicable to the problem. Before we examine this question, let us recall that we are *not* interested in optimizing the user's code. Hence, the translated user code is fixed to be a run-time value.

By examining the translation, the security state is almost always a compile-time value. Lifting it to a run-time value is almost certainly a bad idea because it causes run-time checks to appear in the compiled code. Checking that they do not have to appear amounts to performing the optimization in the next section.

The security state must become a run-time value only when there is a primitive operation whose state transition function requires its run-time arguments. This binding time is a semantic necessity and cannot be amended by binding-time variation or lifting.

The underlying problem is really that the traditional well-formedness restriction must be overcome because the (compile-time) security state must be passed to a (run-time) user function.

4 Compiling policies using intersection types

While type specialization of terms translated as in Fig. 7 does not retain safety checks at run-time, the generated code can be redundant. Many specialized versions are identical and the code grows considerably. In consequence, this approach is not really feasible for all but trivial examples.

4.1 A refined approach

To find a remedy against code duplication, let's look again at the translation of $\lambda(\mathit{file})\mathtt{read}(\mathit{file})$ in Fig. 8, (1), its specialized types in (4) and terms in (5). Despite the difference in the specialization types, the code is identical. However, the function is *not* polymorphic with respect to the compile-time state because it maps *before-read* to *after-read* and is undefined on state *bad*.

However, the translated term (1) has an *intersection type* [4,6,27,28]:

$$\bigwedge \begin{bmatrix} (S\{\mathit{before\text{-}read}\}, \mathrm{BaseType}, (S\{\mathit{after\text{-}read}\}, \mathrm{BaseType}) \to \mathtt{Ans}) \to \mathtt{Ans}, \\ (S\{\mathit{after\text{-}read}\}, \mathrm{BaseType}, (S\{\mathit{after\text{-}read}\}, \mathrm{BaseType}) \to \mathtt{Ans}) \to \mathtt{Ans} \end{bmatrix} \tag{6}$$

This observation has lead us to extend type specialization with a restricted notion of intersection types and subtyping. The restriction is that intersection types can

Specialized types (revised)

$$\tau' ::= \ldots \mid \bigwedge [\tau' \ldots \tau']$$

Kinding

$$S\{n\} : \mathit{INT} \qquad \frac{\tau_1' : \mathit{INT} \quad \tau_2' : \mathit{INT}}{\tau_1' + \tau_2' : \mathit{INT}}$$

$$\frac{\tau' : \mathit{INT} \quad \tau'' : \mathit{INT}}{\tau' \sim \tau'' : *} \qquad \mathit{Int} \sim \mathit{Int} : * \qquad \frac{\tau_1' \sim \tau_1'' : * \quad \tau_2' \sim \tau_2'' : *}{\tau_1' \to \tau_2' \sim \tau_1'' \to \tau_2'' : *}$$

$$\frac{(\forall 1 \leq i, j \leq n) \tau_i' \sim \tau_j' : *}{\bigwedge [\tau_1' \ldots \tau_n'] : *} \qquad \frac{\tau' \sim \tau'' : *}{\tau' : *}$$

Equality relation (additional rules)

$$\frac{\tau_1' = \tau_1'' \quad \ldots \quad \tau_n' = \tau_n''}{\bigwedge [\tau_1', \ldots, \tau_n'] = \bigwedge [\tau_1'', \ldots, \tau_n'']}$$

Subtyping relation (extending equality)

$$\frac{i \in \{1, \ldots, n\}}{\bigwedge [\tau_1', \ldots, \tau_n'] \leq \tau_i'} \qquad \frac{(\forall 1 \leq i \leq n) \tau' \leq \tau_i'}{\tau' \leq \bigwedge [\tau_1', \ldots, \tau_n']} \qquad \frac{\tau_2'' \leq \tau_2' \quad \tau_1' \leq \tau_1''}{\tau_2' \to \tau_1' \leq \tau_2'' \to \tau_1''}$$

Additional specialization rules

$$\frac{\Gamma \vdash e \leadsto e' : \tau' \quad \tau' \leq \tau''}{\Gamma \vdash e \leadsto e' : \tau''} \qquad \frac{\begin{array}{c}(\forall 1 \leq i \leq n) \Gamma \vdash e_1 \leadsto e_1' : \tau_i' \\ \Gamma, x \leadsto x' : \bigwedge [\tau_i' \mid 1 \leq i \leq n] \vdash e_2 \leadsto e_2' : \tau'\end{array}}{\Gamma \vdash \underline{\mathtt{let}}\ x = e_1\ \underline{\mathtt{in}}\ e_2 \leadsto \mathtt{let}\ x' = e_1'\ \mathtt{in}\ e_2' : \tau'}$$

Fig. 10. Type specialization with intersections and subtyping

only be formed from structurally isomorphic types that differ in singleton types, as formalized in Fig. 10 using kinds.

In the running example, specialization with intersection types generates the same term $\lambda(\mathit{file}, c)\mathtt{let}\ y_1 = \mathtt{read}(\mathit{file})\ \mathtt{in}\ c@(y_1)$ with type (6).

4.2 Type specialization with intersection types

The extended syntax of specialized types contains finite intersections of types. The kinding rules ensure that intersection types can only be formed from structurally isomorphic types, as formalized with the judgement $\tau' \sim \tau'' : *$. The rules defining $\tau' = \tau''$ make equality compatible with intersection. Subtyping extends conversion with the usual rules for intersection and function subtyping [27].

The additional specialization rules include the standard subsumption rule, which eliminates intersection types where required. The introduction rule for intersection types requires a special $\underline{\mathtt{let}}\ x = e_1\ \underline{\mathtt{in}}\ e_2$ construct because its implementation incurs considerable expense. The type specializer processes the term e_1 for each demanded type τ_i' and checks that the resulting specialized term e_1' is *identical* for each τ_i'. If that is not possible, we must revert to polyvariance and generate a new variant. Quite often, functions are polymorphic with respect to the security state. In this case, the intersection typing generates exactly one variant.

Finally, we have to extend the simulation result (Prop. 1) to the enriched language. Since we are not adding new reductions to the language, it is sufficient to extend the proofs of the substitution lemmas [14]:

Lemma 1 (Source substitution). *If* $\Gamma \vdash e_1 \rightsquigarrow e_1' : \tau_1'$ *and* $\Gamma, x_1 \rightsquigarrow e_1' : \tau_1' \vdash e_2 \rightsquigarrow e_2' : \tau_2'$ *then* $\Gamma \vdash e_2[x_1 \mapsto e_1] \rightsquigarrow e_2' : \tau_2'$.

Lemma 2 (Specialized substitution). *Let* θ *be a substitution that maps variables in specialized terms to specialized terms. If* $\Gamma \vdash e \rightsquigarrow e' : \tau'$ *then* $\theta(\Gamma) \vdash e \rightsquigarrow \theta(e') : \tau'$.

5 Conclusions

We have shown that partial evaluation techniques are well-suited to translate programs into safe programs that observe security policies specified by security automata. We have exhibited a heterogeneous approach that eliminates most run-time security checks, but can result in code duplication.

We have extended the type specializer by intersection types to avoid excessive code duplication in this approach. This refined approach automatically achieves all optimizations mentioned in Walker's work [37]. There is a prototype implementation, which has been used to validate the examples in this paper. It can be obtained from `http://www.informatik.uni-freiburg.de/~thiemann/scheme/ts-ius.scm`.

As future work, we plan to address the restriction to finite sets of security states by splitting them into compile-time and run-time components and to integrate the translation with our earlier work on run-time code generation [33]. The resulting framework will provide just-in-time enforcing compilation and it will serve for experiments with mobile code.

References

1. S. M. Abramov and R. Glück. Semantics modifiers: an approach to non-standard semantics of programming languages. In M. Sato and Y. Toyama, editors, *Third Fuji International Symposium on Functional and Logic Programming*, pages 247–270. World Scientific, 1998.
2. 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.
3. 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.

4. F. Barbanera and M. Dezani-Ciancaglini. Intersection and union types. In T. Ito and A. Meyer, editors, *Proc. Theoretical Aspects of Computer Software*, volume 526 of *Lecture Notes in Computer Science*, Sendai, Japan, 1991. Springer-Verlag.
5. T. Colcombet and P. Fradet. Enforcing trace properties by program transformation. In Reps [29], pages 54–66.
6. M. Coppo and M. Dezani-Ciancaglini. A new type-assignment for λ-terms. *Archiv. Math. Logik*, 19(139-156), 1978.
7. O. Danvy and A. Filinski. Abstracting control. In *Proc. 1990 ACM Conference on Lisp and Functional Programming*, pages 151–160, Nice, France, 1990. ACM Press.
8. O. Danvy and A. Filinski. Representing control: A study of the CPS transformation. *Mathematical Structures in Computer Science*, 2:361–391, 1992.
9. D. Evans and A. Twyman. Flexible policy-directed code safety. In *IEEE Symposium on Security and Privacy*, Oakland, CA, May 1999.
10. R. Glück. On the generation of specializers. *Journal of Functional Programming*, 4(4):499–514, Oct. 1994.
11. R. Glück and J. Jørgensen. Generating optimizing specializers. In *IEEE International Conference on Computer Languages*, pages 183–194. IEEE Computer Society Press, 1994.
12. 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.
13. 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.
14. J. Hughes. The correctness of type specialisation. In G. Smolka, editor, *Proc. 9th European Symposium on Programming*, volume 1782 of *Lecture Notes in Computer Science*, pages 215–229, Berlin, Germany, Mar. 2000. Springer-Verlag.
15. Java2 platform. http://www.javasoft.com/products/, 2000.
16. N. D. Jones, C. K. Gomard, and P. Sestoft. *Partial Evaluation and Automatic Program Generation*. Prentice-Hall, 1993.
17. D. Kozen. Language-based security. Technical Report TR99-1751, Cornell University, Computer Science, June 15, 1999.
18. 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.
19. T. Lindholm and F. Yellin. *The Java Virtual Machine Specification*. Addison-Wesley, 1996.
20. S. Lucco, O. Sharp, and R. Wahbe. Omniware: A universal substrate for web programming. *WorldWideWeb Journal*, 1(1), Dec. 1995.
21. 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.
22. 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.
23. G. C. Necula. Proof-carrying code. In *Proceedings of the 24th ACM Symposium on Principles of Programming Languages*, Paris, France, Jan. 1997.
24. G. C. Necula and P. Lee. Safe kernel extensions without run-time checking. In *Proceedings of the Second Symposium on Operating System Design and Implementation*, Seattle, Wa., Oct. 1996.

25. G. C. Necula and P. Lee. The design and implementation of a certifying compiler. In K. D. Cooper, editor, *Proceedings of the 1998 ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*, pages 333–344, Montreal, Canada, June 1998. ACM. Volume 33(5) of SIGPLAN Notices.
26. 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.
27. B. Pierce. Programming with intersection types, union types, and polymorphism. Technical Report CMU-CS-91-106, Carnegie Mellon University, Feb. 1991.
28. B. C. Pierce. Intersection types and bounded polymorphism. *Mathematical Structures in Computer Science*, 11, 1996.
29. T. Reps, editor. *Proc. 27th Annual ACM Symposium on Principles of Programming Languages*, Boston, MA, USA, Jan. 2000. ACM Press.
30. J. C. Reynolds. Definitional interpreters for higher-order programming languages. In *ACM Annual Conference*, pages 717–740, July 1972.
31. F. B. Schneider. Enforceable security policies. Technical Report TR99-1759, Cornell University, Ithaca, NY, USA, July 1999.
32. M. Sperber, R. Glück, and P. Thiemann. Bootstrapping higher-order program transformers from interpreters. In *Proc. 11th Annual Symposium on Applied Computing, SAC (SAC '96)*, pages 408–413, Philadelphia, PA, Feb. 1996. ACM.
33. M. Sperber and P. Thiemann. Two for the price of one: Composing partial evaluation and compilation. In *Proc. of the ACM SIGPLAN '97 Conference on Programming Language Design and Implementation*, pages 215–225, Las Vegas, NV, USA, June 1997. ACM Press.
34. P. Thiemann and R. Glück. The generation of a higher-order online partial evaluator. In M. Takeichi and T. Ida, editors, *Fuji International Workshop on Functional and Logic Programming*, pages 239–253. World Scientific, Nov. 1995.
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.