A Systematic Approach to Static Access Control

FRANÇOIS POTTIER
INRIA Rocquencourt
CHRISTIAN SKALKA
The University of Vermont
and
SCOTT SMITH
The Johns Hopkins University

The Java Security Architecture includes a dynamic mechanism for enforcing access control checks, the so-called *stack inspection* process. While the architecture has several appealing features, access control checks are all implemented via dynamic method calls. This is a highly nondeclarative form of specification that is hard to read, and that leads to additional run-time overhead. This article develops type systems that can statically guarantee the success of these checks. Our systems allow security properties of programs to be clearly expressed within the types themselves, which thus serve as static declarations of the security policy. We develop these systems using a systematic methodology: we show that the security-passing style translation, proposed by Wallach et al. [2000] as a *dynamic* implementation technique, also gives rise to *static* security-aware type systems, by composition with conventional type systems. To define the latter, we use the general HM(X) framework, and easily construct several constraint- and unification-based type systems.

Categories and Subject Descriptors: D.3.3 [**Programming Languages**]: Language Constructs and Features—*Control structures*; *polymorphism*; F.3.3 [**Logics and Meanings of Programs**]: Studies of Program Constructs—*Type structure*

General Terms: Languages, Reliability, Security, Theory

Additional Key Words and Phrases: Type systems, stack inspection, access control

1. INTRODUCTION

The Java Security Architecture [Gong and Schemers 1998; Gong 1998], found in the Java JDK 1.2 and later, includes mechanisms to protect systems from operations performed by untrusted code. These access control decisions are enforced

Corresponding authors' address: C. Skalka, Department of Computer Science, University of Vermont, Votey 351, 33 Colchester Ave., Burlington, VT 05405; email: skalka@cs.uvm.edu.

by *dynamic* checks. Our goal is to make some or all of these decisions *statically*, by extensions to the type system. Thus, access control violations will be caught at compile-time rather than run-time. Furthermore, these type extensions constitute a statically-specified security policy, which is much preferred to a dynamic one.

1.1 The Java Security Architecture

We now briefly review the Java security Architecture [Gong and Schemers 1998; Gong 1998; Wallach 1999]. The stack inspection algorithm underlying the architecture is primarily concerned with code-based access control: in a single JVM can be found code loaded from different *codebases*, and code from each codebase may have different access rights. For instance, applets should not be allowed to read and write arbitrary files, but applets may be allowed to read and write files in `/tmp/*`. Thus, applets may have a `FilePermission` for read/write to `/tmp/*`, but no permissions to read or write any other files.

The stack inspection system is used in two different modes; these two different modes are not stated very clearly in the literature so we review them now. In the first mode, a `checkPermission()` command is executed before a critical operation, such as a system library about to do a low-level file write; if this command does not raise an exception, execution continues and the file is written. For the applet example, if the applet tries to write `/tmp/scratch2232`, the `checkPermission()` will succeed since the applet has this privilege (we will describe the checking process in more detail below). In the second mode, there may be a need to *temporarily raise* privileges to allow the system to perform a privileged operation for untrusted code. An example is the system may need to read a font file, `/usr/java/fonts/helvetica.fnt`, so the applet can use this font, but this would otherwise cause an exception since the applet cannot read that file: the `checkPermission()` for read of `/usr/java/fonts/helvetica.fnt` would fail. The `doPrivileged()` command is designed to solve this problem: the system can execute `doPrivileged(readFontCode)` where `readFontCode` reads the font and is executed with system, not applet, privileges; and, the `checkPermission()` will succeed since it was executed as a system-privileged operation.

Access control decisions of `checkPermission()` are made using a stack inspection algorithm. The original requestor of an action such as a file read may be far back on the call stack: the applet invoked some system file method which in turn invoked other system methods ... which finally invoked a low-level system method to read the file which invoked `checkPermission()`. So, back on the call stack is a frame owned by the applet codebase. The `checkPermission()` thus searches back the stack, making sure *every* frame's codebase has the permission needed. This covers the first case of usage above. For the second case, where a temporary raising of privileges is needed to for example, read a font file, the `doPrivileged()` command adds a flagged stack frame to the stack which performs the privileged operation; when a privilege is checked via the `checkPermission()` command, the stack frames are searched most to least recent. If a `doPrivileged` frame for the relevant permission is encountered, and the codebase of every frame up to and including that one is authorized for the permission, the check terminates successfully: even though applet stack frames may be further up the stack because applet code induced the font load, its privileges are not queried.

1.1.1 *Java's Lack of Full Declarativity.* The Java Security Architecture is popular in practice and embodies several useful principles, but it also has some weaknesses. There is a performance penalty to pay due to the need for run-time stack inspection. The architecture also is not as declarative as it could be, but for security policies it is important to be maximally declarative: fixed, immutable policies have fixed meaning.

The Java policy file is a fixed declaration of privilege authorizations for codebases, so this aspect of the architecture is sufficiently declarative. The problem is how this policy is enforced in the code: for example, is code from `foo.com` indeed restricted at runtime from writing to `"/tmp"`, if this is declared in the policy file? In fact, implementation of this policy requires that there be appropriate insertions of `checkPermissions` which guard all low-level file accesses, which are checked dynamically. Thus, a programmer must have a perfect understanding of the control flow of the underlying program to guarantee that proper checks are in place. This obviously makes it difficult to see whether the code is implementing the correct policy; in large programs, tens of thousands of lines long, how can programmers have such a perfect understanding?

This article explores solutions to these problems through the use of static type systems. If types can declare precisely the privileges needed for an invocation of a method to avoid run-time security exceptions, these types could give a top-level declaration of the permissions needed by each chunk of code, and programmers could verify that the correct policies are implemented without having to understand the complete codebase.

1.2 Our Framework

We define a security typing system which statically typechecks, and thus statically verifies success of, the run-time access control checks. This obviates the need for stack inspection at run-time, since all the checks have been proven to succeed at compile-time. In this article, a foundational framework is developed; there still are several important issues to be addressed before it could be applied to a real language such as Java.

We employ several technical tools to streamline the results. We reduce the security typing problem to a conventional typing problem using a translation-based method inspired by Pottier and Conchon [2000]. We use a standard language of row types [Rémy 1992b] to describe sets of privileges. We also re-use the HM(X) framework [Odersky et al. 1999; Sulzmann 2000], which allows a wide variety of type systems to be defined in a single stroke, saves some proof effort, and (most importantly) shows that our custom type systems arise naturally out of a standard one. Some technical results about HM(X) are drawn from [Skalka and Pottier 2002]. We develop several different type systems, including both constraint-based and unification-based systems.

We begin by defining a simplified model of the Java Security Architecture, λ_{sec}. This calculus is equipped with a nonstandard operational semantics that includes a specification of stack inspection. In order to construct a static type system for λ_{sec}, we translate it into a standard λ-calculus, called λ_{set}. The translation is a security-passing style transformation [Wallach 1999; Wallach et al. 2000]: it implements stack inspection by passing around sets of privileges at run-time. For this purpose, λ_{set} is equipped with built-in notions of set and set operations. The translation is proven to be correct, in that program semantics are preserved in translation.

Then, we define a type system for λ_{set}. Because λ_{set} is a standard λ-calculus, we are able to define our type system as a simple instance of the HM(X) framework [Odersky et al. 1999]. In fact, by using this framework, a whole family of type systems may be succinctly defined, each with different costs and benefits. In order to give precise types to λ_{set}'s built-in set operations, our instance uses set types, defined as a simplification of Rémy's record types [Rémy 1992b].

Due to correctness of the λ_{sec}-to-λ_{set} translation, and type safety within the λ_{set} type framework, an *indirect* type analysis for λ_{sec} is immediately obtained. That is, a sound typing for any λ_{sec} expression is the type of its encoding in λ_{set}. However, a *direct* type system that treats λ_{sec} expressions themselves is still desirable, for various reasons (e.g., efficiency, error reporting). Thus, we lastly define direct type systems for λ_{sec}, which are based on, or "derived" from, analogous λ_{set} type systems. As an appealing consequence of our technical approach, a direct type safety result follows easily from indirect type safety, correctness of the λ_{sec}-to-λ_{set} translation, and a straighforward syntactic correspondence between the direct and indirect type systems.

This article expands on the conference paper [Pottier et al. 2001], which was itself a refiguration of the ideas first presented in Skalka and Smith [2000]. The latter paper defined the first static type analysis for stack inspection. There, function types are of the form $\tau_1 \xrightarrow{\Pi} \tau_2$, where τ_1 and τ_2 are "ordinary" types, and Π represents a family of sets containing at least the permissions necessary to use the function. An inference technique based on a set constraint solution algorithm was defined to implement the system. However, the system is non-standard and monomorphic; these shortcomings are addressed in Pottier et al. [2001] and the current article, which extend the type analysis to a polymorphic setting, using standard type logics with well-studied and efficient inference methods.

2. THE SOURCE LANGUAGE λ_{sec}

This section defines λ_{sec}, a simplified model of the security architecture of the JDK 1.2 and later. It is a λ-calculus equipped with a notion of code ownership and constructs for enabling or checking privileges. For the sake of formal simplicity, we do not define stacks explicitly; rather, stacks are implicit in λ_{sec} evaluation contexts, and can be gleaned from them. This is in contrast to a version of the calculus presented in Skalka [2002] with explicit stacks, inspection thereon, and a dopriv construct, called $\lambda_{\text{sec}}^{\mathsf{S}}$, that clearly reflects the JDK implementation details. However, $\lambda_{\text{sec}}^{\mathsf{S}}$ is shown to be embeddable in λ_{sec} in Skalka [2002], ensuring confidence in the correctness of λ_{sec} as a model of the Java JDK architecture.

$r \in \mathcal{R}, R \subseteq \mathcal{R}$	*resources*
$p \in \mathcal{P}, P \subseteq \mathcal{P}$, *where* $\mathcal{P} = 2^{\mathcal{R}}$	*principals*
$v ::= \mathsf{fix}\, z.\lambda x.f$	*values*
$e ::= x \mid \mathsf{fix}\, z.\lambda x.f \mid e\, e \mid \mathsf{let}\, x = e\, \mathsf{in}\, e \mid \mathsf{enable}\, r\, \mathsf{in}\, e \mid \mathsf{check}\, r\, \mathsf{then}\, e \mid \mathsf{test}\, r\, \mathsf{then}\, e\, \mathsf{else}\, e \mid f$	*expressions*
$f ::= p.e$	*signed expressions*
$E ::= [] \mid E\, e \mid v\, E \mid \mathsf{let}\, x = E\, \mathsf{in}\, e \mid \mathsf{enable}\, r\, \mathsf{in}\, E \mid p.E$	*evaluation contexts*

Fig. 1. Grammar for λ_{sec}.

We assume given an arbitrary set $\mathcal{R}$ of *resources* (also known as *privileges*). We use r and R to range over resources and over sets thereof, respectively. Following Fournet and Gordon [2002], we define the set of *principals* $\mathcal{P}$ as the powerset of $\mathcal{R}$, that is, we identify a principal with the set of resources to which it has access. We use p and P to range over principals and over sets thereof, respectively. We write *nobody* for the empty privilege set, that is, for the principal with no access rights. For typing purposes, we shall require every set of resources to be either finite or cofinite (Section 5.3).

The reader may be somewhat puzzled by the fact that both p and R range over sets of resources. The choice of notation is intended to reflect the manner in which a set of resources is obtained. On the one hand, the notation p represents the set of resources associated (via an implicit access rights matrix) with some principal name, found in the code. On the other hand, the notation R represents an arbitrary set of resources and may be the result of a computation involving union and intersection operations. In other words, p represents what Fournet and Gordon refer to as a "static" set of privileges, while R represents a "dynamic" set of privileges.

The grammar of λ_{sec} is given in Figure 1. An abstraction $\mathsf{fix}\, z.\lambda x.f$ may recursively refer to itself through the program variable z. (This conflation of the fix and λ binders simplifies the treatment of recursion.) We write $\lambda x.f$ when z does not appear free in f. The let form does not make the untyped calculus more expressive; instead, as in ML, it is used by the type system to determine where polymorphism may be introduced. A *signed expression* $p.e$ behaves as the expression e endowed with the authority of principal p. The body of every λ-abstraction is required to be a signed expression—thus, every piece of code must be vouched for by some principal. The construct $\mathsf{enable}\, r\, \mathsf{in}\, e$ allows an authorized principal to enable the use of a resource r within the expression e. The construct $\mathsf{check}\, r\, \mathsf{then}\, e$ asserts that the use of r is currently enabled. If r is indeed enabled, e is evaluated; otherwise, execution fails. The construct $\mathsf{test}\, r\, \mathsf{then}\, e_1\, \mathsf{else}\, e_2$ dynamically tests whether r is enabled, branching to e_1 or e_2 if this holds or fails, respectively. Versions of enable, check, and test that bear on a set of resources R, as opposed to a single resource r, may be later introduced as syntactic sugar.

2.1 Stack Inspection

The JDK determines whether a resource is enabled by literally examining the runtime stack, hence the name *stack inspection*. We give a simple specification

$$\frac{r \in p \qquad S \vdash r}{S.p \vdash r} \qquad \frac{S \vdash r}{S.r' \vdash r} \qquad \frac{S \vdash_{\bullet} r}{S.r \vdash r} \qquad \frac{S \vdash_{\bullet} r}{S.r' \vdash_{\bullet} r} \qquad \frac{r \in p}{S.p \vdash_{\bullet} r}$$

Fig. 2. Backward stack inspection algorithm.

$$\frac{\begin{array}{c} \mathit{nobody}, \varnothing, S \vdash R \\ r \in R \end{array}}{S \vdash r} \qquad p, R, \epsilon \vdash R \qquad \frac{p', R \cap p', S \vdash R'}{p, R, p'.S \vdash R'} \qquad \frac{p, R \cup (\{r\} \cap p), S \vdash R'}{p, R, r.S \vdash R'}$$

Fig. 3. Forward stack inspection algorithm.

of this process by noticing that stacks are implicitly contained in *evaluation contexts*, whose grammar is defined in Figure 1. Indeed, a context defines a path from the term's root down to its active redex, along which one finds exactly the security annotations which the JDK would maintain on the stack, that is, code owners p and enabled resources r.

To formalize this idea, we associate to every evaluation context E a finite string $|E|$ of principals and resources, called a *stack*. The rightmost letters in the string correspond to the most recent stack frames. We write ϵ for the empty stack and $S_1.S_2$ for the concatenation of the stacks S_1 and S_2.

$$\begin{array}{rlrl} |[]| &= \epsilon & |E\,e| &= |E| \\ |v\,E| &= |E| & |\mathsf{let}\,x = E\,\mathsf{in}\,e| &= |E| \\ |\mathsf{enable}\,r\,\mathsf{in}\,E| &= r.|E| & |p.E| &= p.|E|\,\cdot \end{array}$$

We can now define a "stack inspection" algorithm. We give two variants of it, a backward (Figure 2) and a forward one (Figure 3). Both are defined in terms of a judgement of the form $S \vdash r$, which may be read: *inspecting the stack S to check privilege r succeeds*. The former algorithm scans the stack, starting with the most recent frames, then moving towards their ancestors. The latter, on the other hand, scans the stack in the order it was built. Furthermore, its formulation is altered so that it internally computes not only whether access to a given resource r is legal, but also the set of all resources that may be legally accessed given the current stack. These algorithms are referred to as *lazy* and *eager*, respectively, by Gong [Gong and Schemers 1998; Gong 1998]. While the former is employed by most current JVM implementations, the latter forms the basis of the security-passing style [Wallach 1999] translation which we will introduce in Section 4.

The following theorem states that forward and backward stack inspection are in fact equivalent. This initial result is later used to establish the correctness of *security-passing style* (Theorem 2). Subsequently, we will write $S \vdash r$ without specifying which of the two algorithms is being used. We will also write $E \vdash r$ for $|E| \vdash r$.

THEOREM 1. *Assume given a stack S and a resource r. Let P stand for the set of all principals that contain r. Then, the following three statements are equivalent:*

(1) *$S \vdash r$ holds according to the rules of Figure 2;*
(2) *$S \vdash r$ holds according to the rules of Figure 3;*
(3) *some suffix of S belongs to the regular language $P\mathcal{R}^\star r(P \mid \mathcal{R})^\star$.*

Proof. We begin by proving that the first statement is equivalent to the third one. First, check that the auxiliary judgment $S \vdash_\bullet r$ holds if and only if some suffix of S belongs to $P\mathcal{R}^\star$. Then, check that $S \vdash r$ holds, according to the rules of Figure 2, if and only if some suffix of S belongs to the regular language $P\mathcal{R}^\star r(P \mid \mathcal{R})^\star$. Each of these checks is immediate.

We now prove that the second statement is equivalent to the third one. Let A (respectively, B; respectively, C) be the set of stacks S such that $\exists R' \ni r$ $p, R, S \vdash R'$ for some (or, equivalently, for all) p, R such that $p \not\ni r \wedge R \not\ni r$ (respectively, $p \ni r \wedge R \not\ni r$, respectively, $p \ni r \wedge R \ni r$). It is straightforward to check that, according to the last three rules in Figure 3, A, B and C are the least solutions to the following recursive equations:

$$\begin{aligned} A &::= P.B \mid (\mathcal{P} \setminus P).A \mid \mathcal{R}.A \\ B &::= P.B \mid (\mathcal{P} \setminus P).A \mid r.C \mid (\mathcal{R} \setminus \{r\}).B \\ C &::= \epsilon \mid (\mathcal{P} \setminus P).A \mid (P \mid \mathcal{R}).C. \end{aligned}$$

An inductive argument shows that $A \subseteq B \subseteq C$ holds. Then, through a few rewriting steps, one can bring the equations into a form where it is evident that A is exactly $(\mathcal{P} \mid \mathcal{R})^\star P\mathcal{R}^\star r(P \mid \mathcal{R})^\star$. We do not give the details. In principle, the check can be mechanized by verifying that the minimal deterministic finite automaton (over the 4-symbol alphabet $\{r\}$, $\mathcal{R} \setminus \{r\}$, P and $\mathcal{P} \setminus P$) associated with this regular expression is exactly the one described by the above equations. There remains to conclude by noticing that, according to the first rule in Figure 3, $S \vdash r$ holds if and only if $S \in A$. □

2.2 Operational Semantics for λ_{sec}

The operational semantics of λ_{sec} is defined by the following reduction rules:

$$\begin{array}{rcll} E[(\mathsf{fix}\, z.\lambda x.f)\,v] & \rightarrow & E[f[v/x][\mathsf{fix}\, z.\lambda x.f/z]] & \\ E[\mathsf{let}\, x = v\, \mathsf{in}\, e] & \rightarrow & E[e[v/x]] & \\ E[\mathsf{check}\, r\, \mathsf{then}\, e] & \rightarrow & E[e] & \text{if } E \vdash r \\ E[\mathsf{test}\, r\, \mathsf{then}\, e_1\, \mathsf{else}\, e_2] & \rightarrow & E[e_1] & \text{if } E \vdash r \\ E[\mathsf{test}\, r\, \mathsf{then}\, e_1\, \mathsf{else}\, e_2] & \rightarrow & E[e_2] & \text{if } \neg(E \vdash r) \\ E[\mathsf{enable}\, r\, \mathsf{in}\, v] & \rightarrow & E[v] & \\ E[p.v] & \rightarrow & E[v]. & \end{array}$$

The evaluation context E is made explicit in every rule, which allows looking it up when needing to perform security checks. Note that it is *not* the case that $e \rightarrow e'$ implies $E[e] \rightarrow E[e']$. Indeed, enclosing e within a new evaluation context E enables more privileges, possibly causing tests of the form $\mathsf{test}\, r\, \mathsf{then}\, e_1\, \mathsf{else}\, e_2$ to be resolved differently.

The first two rules are standard. The next rule allows $\mathsf{check}\, r\, \mathsf{then}\, e$ to reduce into e only if stack inspection succeeds (as expressed by the side condition $E \vdash r$); otherwise, execution is blocked. The following two rules use stack inspection in a similar way to determine how to reduce $\mathsf{test}\, r\, \mathsf{then}\, e_1\, \mathsf{else}\, e_2$;

$$
\begin{array}{ll}
e ::= x \mid v \mid e\,e \mid \mathsf{let}\,x = e\,\mathsf{in}\,e & \textit{expressions} \\
v ::= \mathsf{fix}\,z.\lambda x.e \mid R \mid ._r \mid ?_r \mid \vee_R \mid \wedge_R & \textit{values} \\
E ::= [] \mid E\,e \mid v\,E \mid \mathsf{let}\,x = E\,\mathsf{in}\,e & \textit{evaluation contexts}
\end{array}
$$

Fig. 4. Grammar for λ_{set}.

however, they never cause execution to fail. The last two rules state that security annotations become unnecessary once the expression they enclose has been reduced to a value. In a Java virtual machine, these rules would be implemented simply by popping stack frames (and the security annotations they contain) after executing a method.

This operational semantics constitutes a concise, formal description of Java stack inspection in a higher-order setting. It is easy to check that every closed term either is a value, or is reducible, or is of the form $E[\mathsf{check}\,r\,\mathsf{then}\,e]$ where $\neg(E \vdash r)$. Terms of the third category are *stuck*; they represent access control violations. An expression e is said to *go wrong* if and only if $e \to^\star e'$, where e' is a stuck expression, holds.

3. THE TARGET CALCULUS λ_{set}

We now define a standard calculus, λ_{set}, to be used as the target of our translation. It is a λ-calculus equipped with a number of constants which provide set operations, and is given in Figure 4. We will use $e.r$, $e?r$, $e \vee R$ and $e \wedge R$ as syntactic sugar for $(._r\, e)$, $(?_r\, e)$, $(\vee_R\, e)$ and $(\wedge_R\, e)$, respectively.

The constant R represents a constant privilege set. The construct $e.r$ asserts that r is an element of the set denoted by e; its execution fails if that is not the case. The construct $e \vee R$ (respectively, $e \wedge R$) allows computing the union (respectively, intersection) of the set denoted by e with a constant set R. Lastly, the expression $e?r\ x\ y$ dynamically tests whether r belongs to the set R denoted by e, and accordingly invokes x or y, passing R to it. The operational semantics for λ_{set} is as follows:

$$
\begin{array}{rcll}
(\mathsf{fix}\,z.\lambda x.e)\,v & \to & e[v/x][\mathsf{fix}\,z.\lambda x.e/z] & \\
\mathsf{let}\,x = v\,\mathsf{in}\,e & \to & e[v/x] & \\
R.r & \to & R & \text{if } r \in R \\
R?r & \to & \lambda x.\lambda y.(x\ R) & \text{if } r \in R \\
R?r & \to & \lambda x.\lambda y.(y\ R) & \text{if } r \notin R \\
R_1 \vee R_2 & \to & R_1 \cup R_2 & \\
R_1 \wedge R_2 & \to & R_1 \cap R_2 & \\
E[e] & \to & E[e'] & \text{if } e \to e'.
\end{array}
$$

Again, an expression e is said to *go wrong* if and only if $e \to^\star e'$, where e' is a stuck expression, holds.

4. SOURCE-TO-TARGET TRANSLATION

4.1 Definition

A translation of λ_{sec} into λ_{set} is defined in Figure 5. The distinguished identifiers s and $\_$ are assumed not to appear in source expressions. Notice that s may

$$
\begin{array}{rcl}
[\![x]\!]_p & = & x \\
[\![\mathsf{fix}\, z.\lambda x.f]\!]_p & = & \mathsf{fix}\, z.\lambda x.\lambda s.[\![f]\!] \\
[\![e_1\, e_2]\!]_p & = & [\![e_1]\!]_p\, [\![e_2]\!]_p\, s \\
[\![\mathsf{let}\, x = e_1\, \mathsf{in}\, e_2]\!]_p & = & \mathsf{let}\, x = [\![e_1]\!]_p\, \mathsf{in}\, [\![e_2]\!]_p \\
[\![\mathsf{enable}\, r\, \mathsf{in}\, e]\!]_p & = & \mathsf{let}\, s = s \vee (\{r\} \cap p)\, \mathsf{in}\, [\![e]\!]_p \\
[\![\mathsf{check}\, r\, \mathsf{then}\, e]\!]_p & = & \mathsf{let}\, \_ = s.r\, \mathsf{in}\, [\![e]\!]_p \\
[\![\mathsf{test}\, r\, \mathsf{then}\, e_1\, \mathsf{else}\, e_2]\!]_p & = & s?r\, (\lambda s.[\![e_1]\!]_p)\, (\lambda s.[\![e_2]\!]_p) \\
[\![f]\!]_p & = & [\![f]\!] \\
\\
[\![p.e]\!] & = & \mathsf{let}\, s = s \wedge p\, \mathsf{in}\, [\![e]\!]_p
\end{array}
$$

Fig. 5. Source-to-Target translation.

appear free in translated expressions. Translating an (unsigned) expression requires specifying the current principal p.

One will often wish to translate an expression under minimal hypotheses, i.e. under the principal *nobody* and a void security context. To do so, we define $(\!| e |\!) = [\![e]\!]_{nobody}[\varnothing/s]$. Notice that s does not appear free in $(\!| e |\!)$. If e is closed, then so is $(\!| e |\!)$.

The idea behind the translation is simple: the variable s is bound at all times to the set of currently enabled resources. Every function accepts s as an extra parameter, because it must execute within its caller's security context. As a result, every function call has s as its second parameter. The constructs $\mathsf{enable}\, r\, \mathsf{in}\, e$ and $p.e$ cause s to be locally bound to a new value, reflecting the new security context; more specifically, the former enables r, while the latter disables all privileges not available to p. The constructs $\mathsf{check}\, r\, \mathsf{then}\, e$ and $\mathsf{test}\, r\, \mathsf{then}\, e_1\, \mathsf{else}\, e_2$ are implemented simply by looking up the current value of s. In the latter, s is re-bound, within each branch, to the *same* value. This may appear superfluous at first sight, but has an important impact on typing, because it allows s to be given a different (more precise) type within each branch.

This translation can be viewed as a generalization of the security-passing style transformation [Wallach 1999; Wallach et al. 2000] to a higher-order setting. While Wallach et al. [2000] advocated this idea as an implementation technique, with efficiency in mind, we use it only as a vehicle in the proof of our type systems. Here, efficiency is not at stake: it is sufficient that the translation scheme be correct. The next section is devoted to proving this (in addition to its utility for our technical purposes, it is the first formal correctness result for security-passing style).

One should point out that this correctness proof is made necessary only by the fact that we chose to define the semantics of λ_{sec} at the source level (Section 2.2). If, instead, we had chosen to consider the security-passing style translation as a *definition* of λ_{sec}'s semantics, then no proof would be necessary. Banerjee and Naumann [2001] follow the latter approach, by giving a denotational semantics that incorporates the security-passing style translation.

4.2 Properties

A basic property of the translation is that s never appears free in the translation of a value. Furthermore, the translation of a value does not depend on the current principal, so we write $[\![v]\!]$ instead of $[\![v]\!]_p$.

For the purposes of our proofs, we need to isolate a particular subclass of target language reductions, which we wish to view as "administrative" (in a sense to be explained later). Let $\rightarrow_\sim$ be the subset of $\rightarrow^\star$ defined by

$$\begin{array}{rcll} a & ::= & R \mid a \vee R \mid a \wedge R & \\ \mathsf{let}\, s = a \,\mathsf{in}\, e & \rightarrow_\sim & e[R/s] & \text{if}\, a \rightarrow^\star R \\ E[e] & \rightarrow_\sim & E[e'] & \text{if}\, e \rightarrow_\sim e'. \end{array}$$

Our first lemma expresses the fact that the translation *implements* the forward stack inspection algorithm. It states that if $p, R, E \vdash R'$ holds (as per the rules of Figure 3), then evaluating $[\![E[e]]\!]_p$ in a context where s is bound to R leads to evaluating $[\![e]\!]_{p'}$, for some p', in a context where s is bound to R'. Furthermore, this is a purely administrative reduction sequence. That is, it only affects the security context, and does not reflect any computational steps apparent in the original program. The proof of the lemma presents no difficulty, because of the close similarity between the definitions of the translation function and of the stack inspection algorithm.

LEMMA 1. *Assume $p, R, S \vdash R'$ and $S = |E|$. Then, there exist a (target) evaluation context E' and a principal p' such that, for every source expression e,*

$$[\![E[e]]\!]_p[R/s] \rightarrow_\sim^\star E'[[\![e]\!]_{p'}[R'/s]].$$

PROOF. By induction over the structure of E. Let θ and θ' stand for the substitutions $[R/s]$ and $[R'/s]$, respectively.

Case $E = []$. Then, $S = \epsilon$ and $R = R'$. Thus, picking $E' = []$ and $p' = p$ trivially satisfies our requirement.

Case $E = E_1\, e_1$. Then,

$$[\![E[e]]\!]_p\theta = [\![E_1[e]]\!]_p\theta\ [\![e_1]\!]_p\theta\ R.$$

Furthermore, the induction hypothesis, applied to E_1, yields E_1' and p' such that $[\![E_1[e]]\!]_p\theta \rightarrow_\sim^\star E_1'[[\![e]\!]_{p'}\theta']$. So, picking $E' = E_1'\ [\![e_1]\!]_p\theta\ R$ fits the bill.

Case $E = v\, E_1$. This case is similar to the previous one. Apply the induction hypothesis to obtain E_1' and p'. Then, pick $E' = [\![v]\!]\ E_1'\ R$. (E' is indeed an evaluation context, because $[\![v]\!]$ is a value.)

Case $E = \mathsf{let}\, x = E_1 \,\mathsf{in}\, e_1$. This case is also similar. Apply the induction hypothesis to obtain E_1' and p'. Then, pick $E' = \mathsf{let}\, x = E_1' \,\mathsf{in}\, [\![e_1]\!]_p\theta$.

Case $E = \mathsf{enable}\, r \,\mathsf{in}\, E_1$. Then, $S = r.S_1$, where $S_1 = |E_1|$. Thus, from $p, R, S \vdash R'$, we may deduce $p, R_1, S_1 \vdash R'$, where R_1 stands for $R \cup (\{r\} \cap p)$. Define $\theta_1 = [R_1/s]$. Then,

$$\begin{array}{rcl} [\![E[e]]\!]_p\theta & = & \mathsf{let}\, s = R \vee (\{r\} \cap p) \,\mathsf{in}\, [\![E_1[e]]\!]_p \\ & \rightarrow_\sim & [\![E_1[e]]\!]_p\theta_1. \end{array}$$

Applying the induction hypothesis to E_1 yields E_1', p' such that $[\![E_1[e]]\!]_p\theta_1 \rightarrow_\sim^\star E_1'[[\![e]\!]_{p'}\theta']$. So, picking $E' = E_1'$ meets our goal.

Case $E = p_1.E_1$. Then, $S = p_1.S_1$, where $S_1 = |E_1|$. Thus, from $p, R, S \vdash R'$, we may deduce $p_1, R_1, S_1 \vdash R'$, where R_1 stands for $R \cap p_1$. Define $\theta_1 = [R_1/s]$.

Then,

$$\begin{array}{rcl} [\![E[e]]\!]_p \theta & = & \mathsf{let}\, s = R \wedge p_1 \,\mathsf{in}\, [\![E_1[e]]\!]_{p_1} \\ & \rightarrow_{\sim} & [\![E_1[e]]\!]_{p_1} \theta_1. \end{array}$$

Applying the induction hypothesis to E_1 yields E'_1, p' such that $[\![E_1[e]]\!]_{p_1}\theta_1 \rightarrow^{\star}_{\sim} E'_1[[\![e]\!]_{p'}\theta']$. So, picking $E' = E'_1$ meets our goal. □

We now come to our central lemma, stating that, if a source expression e leads, in one computation step, to a source expression e', then the translation of e reduces, modulo administrative reductions, to the translation of e'.

LEMMA 2. *$e \rightarrow e'$ implies $(\!| e |\!) \rightarrow^{\star} \cdot {}^{\star}_{\sim}\!\!\leftarrow (\!| e' |\!)$. Furthermore, if the reduction $e \rightarrow e'$ is a β-reduction step, then the reduction sequence $(\!| e |\!) \rightarrow^{\star} \cdot$ involves at least one β-reduction step.*

PROOF. The assertion $e \rightarrow e'$ must be an instance of one of the reduction rules that define the operational semantics (Section 2.2), all of which are of the form $E[e_0] \rightarrow E[e'_0]$. Thus, there exist E, e_0, and e'_0 such that e is $E[e_0]$ and e' is $E[e'_0]$ and e_0, e'_0 have the shape required by one of the reduction rules.

Let $S = |E|$. There exists a unique R such that *nobody*, $\varnothing, S \vdash R$. Clearly, for any resource r, $E \vdash r$ is equivalent to $r \in R$. Define $\theta = [R/s]$. According to Lemma 1, there exist an evaluation context E' and a principal p such that, for any source expression e,

$$(\!| E[e] |\!) \rightarrow^{\star}_{\sim} E'[[\![e]\!]_p \theta].$$

Assume, for the time being, that $[\![e_0]\!]_p\theta \rightarrow^{\star} [\![e'_0]\!]_p\theta$ holds. Then, we have

$$\begin{array}{rcl} (\!| e |\!) = (\!| E[e_0] |\!) & \rightarrow^{\star}_{\sim} & E'[[\![e_0]\!]_p\theta] \\ & \rightarrow^{\star} & E'[[\![e'_0]\!]_p\theta] \\ & {}^{\star}_{\sim}\!\!\leftarrow & (\!| E[e'_0] |\!) = (\!| e' |\!) \end{array}$$

which is the desired result. Hence, there only remains to prove $[\![e_0]\!]_p\theta \rightarrow^{\star} [\![e'_0]\!]_p\theta$, which we now do, by cases on the form of e_0 and e'_0. By definition of e_0 and e'_0, there is one case per reduction rule.

Case $e_0 = (\mathsf{fix}\, z.\lambda x.f)\, v$, $e'_0 = f[v/x][\mathsf{fix}\, z.\lambda x.f/z]$. Then,

$$\begin{array}{lll} [\![e_0]\!]_p\theta & = [\![(\mathsf{fix}\, z.\lambda x.f)\, v]\!]_p\theta & \\ & = ([\![\mathsf{fix}\, z.\lambda x.f]\!]\, [\![v]\!]\, s)\theta & \\ & = (\mathsf{fix}\, z.\lambda x.\lambda s.[\![f]\!])\, [\![v]\!]\, R & \text{because } s \text{ cannot appear free in values} \\ & \rightarrow^2 [\![f]\!][[\![v]\!]/x][[\![\mathsf{fix}\, z.\lambda x.f]\!]/z]\theta & \\ & = [\![f[v/x][\mathsf{fix}\, z.\lambda x.f/z]]\!]\theta & \text{by a straightforward auxiliary lemma} \\ & = [\![e'_0]\!]_p\theta. & \end{array}$$

The auxiliary lemma mentioned above takes advantage of the fact that the translation of a value $[\![v]\!]_p$ does not depend upon the parameter p. We omit its proof.

Case $e_0 = \mathsf{let}\, x = v\, \mathsf{in}\, e_1$, $e'_0 = e_1[v/x]$. Then,

$$\begin{array}{rcll} [\![e_0]\!]_p\theta & = & [\![\mathsf{let}\, x = v\, \mathsf{in}\, e_1]\!]_p\theta & \\ & = & \mathsf{let}\, x = [\![v]\!]\, \mathsf{in}\, [\![e_1]\!]_p\theta & \text{because } s \text{ is not free in } [\![v]\!] \\ & \rightarrow & [\![e_1]\!]_p\theta[[\![v]\!]/x] & \\ & = & [\![e_1]\!]_p[[\![v]\!]/x]\theta & \\ & = & [\![e_1[v/x]]\!]_p\theta & \text{by the same auxiliary lemma} \\ & = & [\![e'_0]\!]_p\theta. & \end{array}$$

Case $e_0 = \mathsf{enable}\, r\, \mathsf{in}\, v$, $e'_0 = v$. Then,

$$\begin{array}{rcl} [\![e_0]\!]_p\theta = [\![\mathsf{enable}\, r\, \mathsf{in}\, v]\!]_p\theta & = & \mathsf{let}\, s = R \vee (\{r\} \cap p)\, \mathsf{in}\, [\![v]\!] \\ & \rightarrow^2 & [\![v]\!] = [\![e'_0]\!]_p\theta. \end{array}$$

Again, we take advantage of the fact that s does not occur free in $[\![v]\!]$.

Case $e_0 = \mathsf{check}\, r\, \mathsf{then}\, e_1$, $e'_0 = e_1$. We must have $E \vdash r$, hence $r \in R$. Then,

$$\begin{array}{rcll} [\![e_0]\!]_p\theta = [\![\mathsf{check}\, r\, \mathsf{then}\, e_1]\!]_p\theta & = & \mathsf{let}\, \_ = R.r\, \mathsf{in}\, [\![e_1]\!]_p\theta & \\ & \rightarrow^2 & [\![e_1]\!]_p\theta & \text{because } r \in R \\ & = & [\![e'_0]\!]_p\theta. & \end{array}$$

Case $e_0 = \mathsf{test}\, r\, \mathsf{then}\, e_1\, \mathsf{else}\, e_2$. Then, e'_0 equals e_i, where $i = 1$ if $E \vdash r$ (or, equivalently, if $r \in R$), and $i = 2$ otherwise. Thus, we have

$$\begin{array}{rcl} [\![e_0]\!]_p\theta = [\![\mathsf{test}\, r\, \mathsf{then}\, e_1\, \mathsf{else}\, e_2]\!]_p\theta & = & R?r\, (\lambda s.[\![e_1]\!]_p)\, (\lambda s.[\![e_2]\!]_p) \\ & \rightarrow^3 & (\lambda s.[\![e_i]\!]_p)\, R \\ & \rightarrow & [\![e_i]\!]_p\theta = [\![e'_0]\!]_p\theta. \end{array}$$

Case $e_0 = p_1.v$, $e'_0 = v$. Then,

$$\begin{array}{rcl} [\![e_0]\!]_p\theta = [\![p_1.v]\!]_p\theta & = & \mathsf{let}\, s = R \wedge p_1\, \mathsf{in}\, [\![v]\!] \\ & \rightarrow^2 & [\![v]\!] = [\![e'_0]\!]_p\theta \end{array}$$

Again, we take advantage of the fact that s does not occur free in $[\![v]\!]_p$, and of the fact that this expression does not depend on p. □

This result is easily generalized to reduction sequences of arbitrary length:

LEMMA 3. *$e \rightarrow^\star e'$ implies $(\!| e |\!) \rightarrow^\star \cdot \overset{\star}{\underset{\sim}{}}\!\!\leftarrow (\!| e' |\!)$. Furthermore, if the reduction sequence $e \rightarrow^\star e'$ involves k β-reduction steps, then the reduction sequence $(\!| e |\!) \rightarrow^\star \cdot$ involves at least k β-reduction steps.*

PROOF. By induction on the length of the reduction sequence $e \rightarrow^\star e'$. In the base case, we have $e = e'$, and the result is immediate. In the inductive case, we have $e \rightarrow e_1 \rightarrow^\star e'$. By applying Lemma 2, on the one hand, and the induction hypothesis, on the other hand, we obtain

$$(\!| e |\!) \rightarrow^\star \cdot \overset{\star}{\underset{\sim}{}}\!\!\leftarrow (\!| e_1 |\!) \rightarrow^\star \cdot \overset{\star}{\underset{\sim}{}}\!\!\leftarrow (\!| e' |\!),$$

where the number of β-reduction steps in the sequences $(\!| e |\!) \rightarrow^\star \cdot$ and $(\!| e_1 |\!) \rightarrow^\star \cdot$ is at least as high as in the source reduction sequences $e \rightarrow e_1$ and $e_1 \rightarrow^\star e'$, respectively. Because the operational semantics of the target language is deterministic, one of the two reduction sequences starting at $(\!| e_1 |\!)$ above must be a subsequence of the other. In either case, the diagram collapses down to

$$(\!| e |\!) \rightarrow^{\star} \cdot \overset{\star}{\underset{\sim}{\leftarrow}} (\!| e' |\!).$$

Furthermore, because β-reduction is not an administrative reduction, the number of β-reduction steps in the sequence $(\!| e |\!) \rightarrow^{\star} \cdot$ is at least as high as in the original reduction sequence $e \rightarrow^{\star} e'$. □

As a corollary, we obtain a soundness theorem for the translation. It essentially states that security-passing style is a valid implementation of the Java stack inspection discipline.

THEOREM 2. *If $e \rightarrow^{\star} v$, then $(\!| e |\!) \rightarrow^{\star} (\!| v |\!)$. If e goes wrong, then $(\!| e |\!)$ goes wrong. If e diverges, then $(\!| e |\!)$ diverges.*

PROOF. First, assume e reduces to a value v. Then, Lemma 3 yields $(\!| e |\!) \rightarrow^{\star} \cdot \overset{\star}{\underset{\sim}{\leftarrow}} (\!| v |\!)$. Because $(\!| v |\!)$ is a value, this diagram collapses down to $(\!| e |\!) \rightarrow^{\star} (\!| v |\!)$.

Second, assume e goes wrong. Then, $e \rightarrow^{\star} e'$, where e' is stuck, holds. We prove that $(\!| e |\!)$ goes wrong by induction on the length of this reduction sequence.

In the base case, we have $e = e'$, that is, e is stuck. So, e must be of the form $E[\mathsf{check}\, r\, \mathsf{then}\, e_1]$, where $\neg(E \vdash r)$. Let $S = |E|$. There exists a unique R' such that $nobody, \varnothing, S \vdash R'$. Necessarily, $r \notin R'$. According to Lemma 1, $(\!| e |\!)$ may be reduced to a term of the form $E'[[\![\mathsf{check}\, r\, \mathsf{then}\, e_1]\!]_{p'}\theta']$, where $\theta' = [R'/s]$. It is easy to check that such a term is stuck. Hence, $(\!| e |\!)$ goes wrong.

In the inductive case, we have $e \rightarrow e_1 \rightarrow^{\star} e'$. Our induction hypothesis shows that $(\!| e_1 |\!)$ goes wrong. Furthermore, Lemma 2 shows that $(\!| e |\!)$ reduces to some reduct of $(\!| e_1 |\!)$. Because reduction is deterministic, $(\!| e |\!)$ must go wrong as well. The result follows.

Third, assume e admits an infinite reduction sequence. This sequence must involve an infinite number of β-reduction steps, because the semantics of λ_{sec}, deprived of the β-reduction rule, is terminating. By Lemma 3, $(\!| e |\!)$ admits an infinite reduction sequence as well. □

5. TYPES FOR λ_{set}

We define a type system for the target calculus as an instance of the parametric framework HM(X) [Odersky et al. 1999; Sulzmann 2000; Skalka and Pottier 2002]. HM(X) is a generic type system in the Hindley–Milner tradition, parameterized by an abstract constraint system X. Section 5.1 briefly recalls its definition. Section 5.2 defines a specific constraint system called SETS, yielding the type system HM(SETS). Section 5.3 extends HM(SETS) to the entire language λ_{set}, by assigning types to its primitive operations. Section 5.4 states type safety results and discusses several choices for our type system, which may be defined as either a *unification*- or *constraint*-based system, and which is flexible with respect to the accuracy of initial type bindings.

5.1 The System HM(X)

We adopt the definition of HM(X) given in Skalka and Pottier [2002]. The framework is parameterized by a *constraint system X*, that is, by notions of *types τ*, *constraints C*, and *interpretation* of constraints in a *model*.

$$\frac{\Gamma(x) = \sigma \qquad C \Vdash \sigma}{C, \Gamma \vdash x : \sigma}\ \textsc{hm-Var} \qquad C, \Gamma \vdash \mathbf{c} : \Delta(\mathbf{c})\ \textsc{hm-Const} \qquad \frac{C, \Gamma \vdash e : \tau \qquad C \Vdash \tau \leq \tau'}{C, \Gamma \vdash e : \tau'}\ \textsc{hm-Sub}$$

$$\frac{C \wedge D, \Gamma \vdash v : \tau \qquad \bar{\alpha} \cap \mathsf{fv}(C, \Gamma) = \varnothing}{C \wedge \exists\bar{\alpha}.D, \Gamma \vdash v : \forall\bar{\alpha}[D].\tau}\ \textsc{hm-}\forall\ \textsc{Intro} \qquad \frac{C, \Gamma \vdash v : \forall\bar{\alpha}[D].\tau \qquad C \Vdash [\bar{\tau}/\bar{\alpha}]D}{C, \Gamma \vdash v : [\bar{\tau}/\bar{\alpha}]\tau}\ \textsc{hm-}\forall\ \textsc{Elim}$$

$$\frac{C, (\Gamma; x : \tau; z : \tau \rightarrow \tau') \vdash e : \tau'}{C, \Gamma \vdash \mathsf{fix}\, z.\lambda x.e : \tau \rightarrow \tau'}\ \textsc{hm-Abs} \qquad \frac{C, \Gamma \vdash e_1 : \tau_2 \rightarrow \tau \qquad C, \Gamma \vdash e_2 : \tau_2}{C, \Gamma \vdash e_1\, e_2 : \tau}\ \textsc{hm-App}$$

$$\frac{C, \Gamma \vdash v : \sigma \qquad C, (\Gamma; x : \sigma) \vdash e : \tau}{C, \Gamma \vdash \mathsf{let}\, x = v\, \mathsf{in}\, e : \tau}\ \textsc{hm-Let}$$

Fig. 6. The system HM(X).

Given a constraint system, a *type scheme* is a triple of a set of quantifiers $\bar{\alpha}$, a constraint C, and a type τ (which, in this paper, must be of kind *Type*; see Section 5.2), written $\sigma ::= \forall\bar{\alpha}[C].\tau$. A *type environment* Γ is a partial mapping of program variables to type schemes. A *judgement* is a quadruple of a constraint C, a type environment Γ, an expression e and a type scheme σ, written $C, \Gamma \vdash e : \sigma$, derivable using the rules of Figure 6. These rules correspond to those given in Skalka and Pottier [2002], less the rules relevant to stateful features, which are not needed in this presentation. Note that via the HM-CONST rule, populating Δ with initial bindings allows typing new language constants in particular instances of HM(X). In the case of λ_{set}, $\mathbf{c}$ will range over the four primitive operators $._r$, $\vee_R$, $\wedge_R$ and $?_r$.

The following syntactic type safety theorem, in the style of Wright and Felleisen [1994], is proven in Skalka and Pottier [2002]. Significantly, the theorem holds with respect to a call-by-value λ-calculus with `let` in *any* instance of HM(X), and the theorem may be easily extended to incorporate additional constants by proving soundness of initial bindings with respect to the semantics of functional constants, the so-called *δ-typability* property.

THEOREM 3. *If $C, \varnothing \vdash e : \sigma$ holds and C is satisfiable, then e does not go wrong.*

We discuss δ-typability and type safety for λ_{set} more thoroughly in Section 5.4.

5.2 The Constraint System SETS

In order to give precise types to the primitive set operations in λ_{set}, we need specific types and constraints. Together with their logical interpretation, which defines their meaning, these form a constraint system called SETS.

The syntax of types and constraints is defined in Figure 7. The type language features four so-called *presence* constructors, two standard *row* constructors [Rémy 1992b], and a *set* type constructor $\{\cdot\}$.

Presence types are used to record whether a resource r appears in a privilege set. **Pre** means r is known to appear in the set, while **Abs** means r is known not

$$\begin{array}{rcll} \tau & ::= & \alpha, \beta, \ldots \mid \tau \rightarrow \tau \mid \{\tau\} \mid r : \tau\,;\, \tau \mid \partial\tau \mid c & \textit{types} \\ c & ::= & \bot \mid \mathbf{Pre} \mid \mathbf{Abs} \mid \top & \textit{presence constructors} \\ C & ::= & \mathbf{true} \mid C \wedge C \mid \exists\alpha.C \mid \tau = \tau \mid \tau \leq \tau & \textit{constraints} \\ & \mid & \text{if } c \leq \tau \text{ then } \tau \leq \tau & (c \neq \bot) \end{array}$$

Fig. 7. SETS grammar.

to appear in it. Of course, our analysis is sometimes approximate: $\top$ means that it is not known whether r is a member of the set. Lastly, concerns of efficiency of type inference call for a fourth presence constructor $\bot$, which, roughly speaking, means that it is irrelevant whether r appears in the set, because the code that requires this privilege test is unreachable. In addition to these four constants, a presence type can also be a variable.

To describe the contents of a set, we use *rows* of presence types. A row is a finite description of an infinite object, namely a (possibly partial) function from resource names to presence types. More precisely, a row describes a function that maps almost all resources in its domain (i.e., all but a finite number of them) to the same type. Rows can be formed using two basic building blocks. First, the row constructor ∂ allows forming constant rows: if τ is a presence type, then $\partial\tau$ is a row that maps all resources in its domain to τ. Second, the row constructor $(r : \cdot\,;\, \cdot)$ allows adding an entry to an existing row: $(r : \tau_1\,;\, \tau_2)$ is a row that maps r to the presence type τ_1 and otherwise behaves as the row τ_2. Lastly, a row can also be a variable. The original presentations of rows [Rémy 1992b,1994] equip row types with an equational theory, which, in particular, allows row entries to commute. In our presentation, these equations are not axioms; they simply happen to hold in our interpretation of types (given below).

A whole set is described by a row τ whose domain is $\mathcal{R}$, wrapped within the *set* type constructor, yielding a type of the form $\{\tau\}$. To determine whether a particular resource r appears in the set, one queries the row τ at r, yielding a presence type. Such a query is carried out by unifying τ against $(r : \gamma\,;\, \beta)$, where γ and β are fresh presence and row variables, respectively. For instance, the singleton set $\{r\}$ is one (and the only) value of type $\{r : \mathbf{Pre}\,;\ \partial\mathbf{Abs}\}$. To determine whether a resource s appears within that set, we solve the equation $(r : \mathbf{Pre}\,;\ \partial\mathbf{Abs}) = (s : \gamma\,;\ \beta)$. If r and s are distinct, this leads to $\gamma = \mathbf{Abs}$ and $\beta = (r : \mathbf{Pre}\,;\ \partial\mathbf{Abs})$, the former of which reflects the fact that s does not belong to $\{r\}$. This treatment of sets is inspired by Wand and Rémy's treatment of records: a set is, in fact, a degenerate record where every field has unit type.

The constraint language offers standard equality and subtyping constraints, as well as a simple form of conditional constraints. Their use will be illustrated in Section 5.3 and 7.2.

To ensure that only meaningful types and constraints can be built, we immediately equip them with *kinds*, defined by:

$$k ::= \mathit{Pres} \mid \mathit{Row}_R \mid \mathit{Type}$$

where R ranges over finite subsets of $\mathcal{R}$. Kinds allow distinguishing presence types, rows, and (regular) types. Furthermore, kinds keep track of every row's domain: a row of kind Row_R represents a function of domain $\mathcal{R} \backslash R$. In particular,

$$\frac{\alpha \in \mathcal{V}_k}{\alpha : k} \qquad \frac{\tau, \tau' : \mathit{Type}}{\tau \to \tau' : \mathit{Type}} \qquad \frac{\tau : \mathit{Row}_{\varnothing}}{\{\tau\} : \mathit{Type}} \qquad \frac{\begin{array}{c}\tau : \mathit{Pres} \qquad r \notin R \\ \tau' : \mathit{Row}_{R \cup \{r\}}\end{array}}{(r : \tau \,;\, \tau') : \mathit{Row}_R} \qquad \frac{\tau : \mathit{Pres}}{\partial \tau : \mathit{Row}_R} \qquad \frac{}{c : \mathit{Pres}}$$

$$\frac{}{\vdash \mathbf{true}} \qquad \frac{\vdash C_1 \qquad \vdash C_2}{\vdash C_1 \wedge C_2} \qquad \frac{\vdash C}{\vdash \exists \alpha . C} \qquad \frac{\tau, \tau' : k}{\begin{array}{c}\vdash \tau = \tau' \\ \vdash \tau \leq \tau'\end{array}} \qquad \frac{\tau, \tau', \tau'' : k \qquad k \neq \mathit{Type}}{\vdash \text{if } c \leq \tau \text{ then } \tau' \leq \tau''}$$

Fig. 8. Kinding rules.

a complete row, that is, a total function from $\mathcal{R}$ to presence types, has kind $\mathit{Row}_{\varnothing}$. For every kind k, we assume given a distinct, denumerable set of *type variables* $\mathcal{V}_k$. We use $\alpha, \beta, \gamma, \ldots$ to represent type variables. From here on, we consider only *well-kinded* types and constraints, as defined in Figure 8. The purpose of these rules is to guarantee that every constraint has a well-defined interpretation within our model, whose definition follows.

To every kind k, we associate a mathematical structure $[\![k]\!]$. $[\![\mathit{Pres}]\!]$ is the set of all four presence constructors. Given a finite set of resources $R \subseteq \mathcal{R}$, $[\![\mathit{Row}_R]\!]$ is the set of total, almost constant functions from $\mathcal{R} \setminus R$ into $[\![\mathit{Pres}]\!]$. $[\![\mathit{Type}]\!]$ is the free algebra generated by the constructors $\to$, with signature $[\![\mathit{Type}]\!] \times [\![\mathit{Type}]\!] \to [\![\mathit{Type}]\!]$, and $\{\cdot\}$, with signature $[\![\mathit{Row}_{\varnothing}]\!] \to [\![\mathit{Type}]\!]$.

Each of these structures is then equipped with an ordering. Here, a choice has to be made. If we do *not* wish to allow subtyping, we merely define the ordering on every $[\![k]\!]$ as equality. Otherwise, we proceed as follows: First, a lattice over $[\![\mathit{Pres}]\!]$ is defined, whose least (respectively, greatest) element is $\bot$ (respectively, $\top$), and where **Abs** and **Pre** are incomparable. This ordering is then extended, point-wise and covariantly, to every $[\![\mathit{Row}_R]\!]$. Finally, it is extended inductively to $[\![\mathit{Type}]\!]$ by viewing the constructor $\{\cdot\}$ as covariant, and the constructor $\to$ as contravariant (respectively, covariant) in its first (respectively, second) argument. This gives rise to a so-called *structural*, *atomic* subtyping relation: that is, two related types may differ only in their presence annotations.

We may now give the interpretation of types and constraints within the model. It is parameterized by a kind-preserving *assignment* ρ, that is, a function which, for every kind k, maps $\mathcal{V}_k$ into $[\![k]\!]$. The interpretation of types is obtained by extending ρ so as to map every type of kind k to an element of $[\![k]\!]$, as follows:

$$\begin{array}{rlrl}
\rho(\tau \to \tau') = \rho(\tau) \to \rho(\tau') & & \rho(\{\tau\}) = \{\rho(\tau)\} & \\
\rho(r : \tau \,;\, \tau')(r) = \rho(\tau) & & \rho(r : \tau \,;\, \tau')(r') = \rho(\tau')(r') & \quad (r \neq r') \\
\rho(\partial \tau)(r) = \rho(\tau) & & \rho(c) = c. &
\end{array}$$

Notice how the interpretation of the two row constructors reflects the informal explanation given above, and validates the expected equational theory. Figure 9 defines the constraint satisfaction predicate $\cdot \vdash \cdot$, whose arguments are an assignment ρ and a constraint C. (The notation $\rho = \rho'[\alpha]$ means that ρ and ρ' coincide except possibly on α.) This definition is standard. The last rule specifies that a conditional constraint whose components are rows is to be interpreted point-wise, that is, as an (infinite) conjunction of conditional constraints bearing on presence types. *Entailment* is then defined as

$$\frac{}{\rho \vdash \mathbf{true}} \qquad \frac{\rho \vdash C_1 \qquad \rho \vdash C_2}{\rho \vdash C_1 \wedge C_2} \qquad \frac{\rho = \rho'\,[\alpha] \qquad \rho' \vdash C}{\rho \vdash \exists \alpha . C}$$

$$\frac{\rho(\tau) = \rho(\tau')}{\rho \vdash \tau = \tau'} \qquad \frac{\rho(\tau) \leq \rho(\tau')}{\rho \vdash \tau \leq \tau'} \qquad \frac{\tau, \tau', \tau'' : \mathit{Pres} \qquad c \leq \rho(\tau) \Rightarrow \rho(\tau') \leq \rho(\tau'')}{\rho \vdash \text{if } c \leq \tau \text{ then } \tau' \leq \tau''}$$

$$\frac{\tau, \tau', \tau'' : \mathit{Row}_R \qquad \forall r \in \mathcal{R} \setminus R \quad c \leq \rho(\tau)(r) \Rightarrow \rho(\tau')(r) \leq \rho(\tau'')(r)}{\rho \vdash \text{if } c \leq \tau \text{ then } \tau' \leq \tau''}$$

Fig. 9. Interpretation of constraints.

usual: $C \Vdash C'$ (read: C entails C') holds iff, for every assignment ρ, $\rho \vdash C$ implies $\rho \vdash C'$.

We refer to the type and constraint logic, together with its interpretation, as SETS. More precisely, we have defined two logics, where $\leq$ is interpreted as either equality or as a nontrivial subtype ordering. We will refer to them as SETS$^{=}$ and SETS$^{\leq}$, respectively.

5.3 Dealing with the Primitive Operations in λ_{set}

The typing rules of HM(X) cover only the λ-calculus with let. To extend HM(SETS) to the whole language λ_{set}, we must assign types to its primitive operations. Let us define an initial type environment Δ_1 as follows:

$$\begin{array}{rl}
R & : \{R : \mathbf{Pre}\,;\ \partial\mathbf{Abs}\} \\
.r & : \forall\beta.\{r : \mathbf{Pre}\,;\ \beta\} \rightarrow \{r : \mathbf{Pre}\,;\ \beta\} \\
\vee_R & : \forall\beta\bar{\gamma}.\{R : \bar{\gamma}\,;\ \beta\} \rightarrow \{R : \mathbf{Pre}\,;\ \beta\} \\
\wedge_R & : \forall\beta\bar{\gamma}.\{R : \bar{\gamma}\,;\ \beta\} \rightarrow \{R : \bar{\gamma}\,;\ \partial\mathbf{Abs}\} \\
?_r & : \forall\alpha\beta\gamma.\{r : \gamma\,;\ \beta\} \rightarrow (\{r : \mathbf{Pre}\,;\ \beta\} \rightarrow \alpha) \rightarrow (\{r : \mathbf{Abs}\,;\ \beta\} \rightarrow \alpha) \rightarrow \alpha.
\end{array}$$

We let α, β, γ range over type variables of kind *Type*, $\mathit{Row}_\star$, *Pres*, respectively. In this definition and from here on, *p and R range over finite sets of resources only*. We exploit this restriction to define the following concise notation, which is used above: if R is $\{r_1, \ldots, r_n\}$, then $R : c$ stands for $r_1 : c\,; \ldots ;\ r_n : c$, and $R : \bar{\gamma}$ stands for $r_1 : \gamma_1\,; \ldots ;\ r_n : \gamma_n$. We note that it is possible to deal with *cofinite* sets of resources as well, by writing $\bar{R}$ for $\mathcal{R} \setminus R$ and by employing the following bindings when R is cofinite:

$$\begin{array}{rl}
R & : \{\bar{R} : \mathbf{Abs}\,;\ \partial\mathbf{Pre}\} \\
\vee_R & : \forall\beta\bar{\gamma}.\{\bar{R} : \bar{\gamma}\,;\ \beta\} \rightarrow \{\bar{R} : \bar{\gamma}\,;\ \partial\mathbf{Pre}\} \\
\wedge_R & : \forall\beta\bar{\gamma}.\{\bar{R} : \bar{\gamma}\,;\ \beta\} \rightarrow \{\bar{R} : \mathbf{Abs}\,;\ \beta\}.
\end{array}$$

Cofinite sets of resources allow modeling principals that enjoy all privileges but a finite number. For the sake of simplicity and brevity, we deal with finite sets of resources only in the following, although, in practice, dealing with both finite and cofinite sets does not raise any additional difficulty.

We may also use conditional constraints to assign a more flexible type scheme to $?_r$. Let Δ_2 be the initial type environment obtained by replacing the last binding in Δ_1 with:

$$
\begin{aligned}
?_r \ :\ & \forall \bar{\alpha}\bar{\beta}\gamma[C].\{r : \gamma\ ;\ \beta\} \to (\{r : \mathbf{Pre}\ ;\ \beta_1\} \to \alpha_1) \to (\{r : \mathbf{Abs}\ ;\ \beta_2\} \to \alpha_2) \to \alpha \\
& \text{where } C = \quad (\text{if } \mathbf{Pre} \leq \gamma \text{ then } \beta \leq \beta_1) \wedge (\text{if } \mathbf{Abs} \leq \gamma \text{ then } \beta \leq \beta_2) \\
& \qquad\qquad \wedge\ (\text{if } \mathbf{Pre} \leq \gamma \text{ then } \alpha_1 \leq \alpha) \wedge (\text{if } \mathbf{Abs} \leq \gamma \text{ then } \alpha_2 \leq \alpha).
\end{aligned}
$$

Here, the input and output of each branch (represented by β_i and α_i, respectively) are linked to the input and output of the whole construct (represented by β and α) through conditional constraints. Intuitively, this means that the security requirements and the return type of a branch may be entirely ignored unless the branch seems liable to be taken. (For more background on conditional constraints, the reader is referred to Aiken et al. [1994] and Pottier [2000].)

5.4 The Type Systems $\mathcal{S}_i^{rel}$

Section 5.2 describes two constraint systems, SETS$^=$ and SETS$^\leq$. Section 5.3 defines two initial typing environments, Δ_1 and Δ_2. These choices give rise to four related type systems, which we refer to as $\mathcal{S}_i^{rel}$, where *rel* and i range over $\{=, \leq\}$ and $\{1, 2\}$, respectively. Each of them offers a different compromise between accuracy, readability and cost of analysis. In each case, Theorem 3 may be extended to the entire language λ_{set} by proving a simple *δ-typability* [Wright and Felleisen 1994] lemma, that is, by checking that Δ_i correctly describes the behavior of the primitive operations. This is the subject of the next section.

Despite sharing a common formalism, these systems may call for vastly different implementations. Indeed, every instance of HM(X) must come with a constraint solving algorithm. $\mathcal{S}_1^=$ is a simple extension of the Hindley-Milner type system with rows, and its constraint solver is row unification [Rémy 1992a]. $\mathcal{S}_2^=$ is similar, but requires conditional (i.e., delayed) unification constraints. $\mathcal{S}_1^\leq$ and $\mathcal{S}_2^\leq$ require solving (structural) subtyping constraints, usually leading to more complex implementations based on transitive closure computations and on-the-fly constraint simplifications (see, e.g., Simonet [2003]). A worst-case time bound for solving possibly conditional subtyping constraints in the presence of rows is given in Pottier [2003]: it is cubic in the size of the program and close to linear in the number of resources that appear in the program, either individually or as part of a principal p. In practice, for all four systems, it is possible to design a constraint solver that scales well.

One should also point out that, when the programming language is extended with a mechanism for *declaring* the type of an expression (or, in Java, of a method), it is necessary to be able to check that the type inferred by the analysis for this expression matches the declaration. This requires an algorithm for deciding *constraint entailment*. In the setting of unification and of structural subtyping, such algorithms exist and are efficient. In the presence of conditional constraints, however, entailment becomes a hard problem [Su and Aiken 2001], making the use of such constraints problematic.

5.5 Proof of δ-typability for λ_{set}

Let us first state some basic properties of sets and set types, whose proofs are omitted.

LEMMA 4. *Let v be a closed value. If $C, \Gamma \vdash v : \{\tau\}$ holds in $\mathcal{S}_i^{rel}$, then v is a set R and $C \Vdash (R : \mathbf{Pre}\,;\ \partial\mathbf{Abs}) \leq \tau$.*

LEMMA 5. *If $C, \Gamma \vdash R : \{R' : \mathbf{Pre}\,;\ \tau\}$ holds in $\mathcal{S}_i^{rel}$, then $R' \subseteq R$.*

LEMMA 6. *If $C, \Gamma \vdash R : \{R' : \bar{\tau}\,;\ \tau\}$ holds in $\mathcal{S}_i^{rel}$, then so do $C, \Gamma \vdash R \cup R' : \{R' : \mathbf{Pre}\,;\ \tau\}$ and $C, \Gamma \vdash R \cap R' : \{R' : \bar{\tau}\,;\ \partial\mathbf{Abs}\}$.*

As mentioned in Section 5.1, extending Theorem 3 to all of λ_{set} only requires proving soundness of the initial bindings for the primitive operators. Let $\delta(\mathbf{c}, v) = v'$ if and only if $\mathbf{c}\, v \rightarrow v'$. We state the so-called *δ-typability* property in the style of Skalka and Pottier [2002]:

LEMMA 7. *In every $\mathcal{S}_i^{rel}$, for every constant $\mathbf{c}$ and closed value v, if $C, \Gamma \vdash \mathbf{c} : \tau_1 \rightarrow \tau_2$ and $C, \Gamma \vdash v : \tau_1$ hold, then $\delta(\mathbf{c}, v)$ is defined and $C, \Gamma \vdash \delta(\mathbf{c}, v) : \tau_2$ holds.*

PROOF. Suppose $C, \Gamma \vdash \mathbf{c} : \tau_1 \rightarrow \tau_2$ and $C, \Gamma \vdash v : \tau_1$. We consider two cases: first, the case where $C, \Gamma \vdash \mathbf{c} : \tau_1 \rightarrow \tau_2$ is obtained via HM-∀ ELIM and HM-SUB; second, the case where it is obtained via HM-∀ ELIM alone. According to the *normalization* result proved in Skalka and Pottier [2002], this is enough.

In the first case, HM-SUB's premises are of the form $C, \Gamma \vdash \mathbf{c} : \tau_1' \rightarrow \tau_2'$ (1) and $C \Vdash \tau_1' \rightarrow \tau_2' \leq \tau_1 \rightarrow \tau_2$ (2). By properties of $\leq$, (2) implies $C \Vdash \tau_1 \leq \tau_1'$ (3) and $C \Vdash \tau_2' \leq \tau_2$ (4). By assumption and HM-SUB, (3) implies $C, \Gamma \vdash v : \tau_1'$ (5). According to the next case of the proof, (1) and (5) imply that $\delta(\mathbf{c}, v)$ is defined and $C, \Gamma \vdash \delta(\mathbf{c}, v) : \tau_2'$ (6) holds. The result follows from (4) and (6) by HM-SUB.

Let us now consider the second case. $\Delta_i(\mathbf{c})$ is a type scheme of the form $\forall\bar{\alpha}[D].\tau_1' \rightarrow \tau_2'$. Because the derivation of $C, \Gamma \vdash \mathbf{c} : \tau_1 \rightarrow \tau_2$ consists of a single instance of HM-∀ ELIM, we have $\tau_1 = \varphi(\tau_1')$ and $\tau_2 = \varphi(\tau_2')$, where φ is a substitution of domain $\bar{\alpha}$ and $C \Vdash \varphi(D)$. We now proceed by case analysis on $\mathbf{c}$ and i:

Case $\mathbf{c} = ._r$. In this case $\tau_1 = \tau_2 = \{r : \mathbf{Pre}\,;\ \varphi\beta\}$. By Lemma 4, v is a set R. By Lemma 5, we further obtain $\{r\} \subseteq R$, hence $\delta(._r, v) = v$. The result $C, \Gamma \vdash \delta(\mathbf{c}, v) : \tau_2$ follows.

Case $\mathbf{c} = \vee_R$. In this case $\tau_1 = \{R : \varphi\bar{\gamma}\,;\ \varphi\beta\}$ and $\tau_2 = \{R : \mathbf{Pre}\,;\ \varphi\beta\}$. By Lemma 4, v is a set R', and $\delta(\vee_R, R') = R \cup R'$. Then, Lemma 6 yields $C, \Gamma \vdash \delta(\mathbf{c}, v) : \tau_2$.

Case $\mathbf{c} = \wedge_R$. In this case $\tau_1 = \{R : \varphi\bar{\gamma}\,;\ \varphi\beta\}$ and $\tau_2 = \{R : \varphi\bar{\gamma}\,;\ \partial\mathbf{Abs}\}$. By Lemma 4, v is a set R', and $\delta(\wedge_R, R') = R \cap R'$. Then, Lemma 6 yields $C, \Gamma \vdash \delta(\mathbf{c}, v) : \tau_2$.

Case $\mathbf{c} = ?_r$ and $i = 1$. In this case $\tau_1 = \{r : \varphi\gamma\,;\ \varphi\beta\}$ and $\tau_2 = (\{r : \mathbf{Pre}\,;\ \varphi\beta\} \rightarrow \varphi\alpha) \rightarrow (\{r : \mathbf{Abs}\,;\ \varphi\beta\} \rightarrow \varphi\alpha) \rightarrow \varphi\alpha$. By Lemma 4, v is a set R, so $\delta(?_r, v)$ is defined. Let us assume $r \in R$ (the other case is analogous). Then, $\delta(?_r, v)$ is $\lambda x.\lambda y.(x\, R)$. By Lemma 4, we have $C \Vdash (R : \mathbf{Pre}\,;\ \partial\mathbf{Abs}) \leq (r : \varphi\gamma\,;\ \varphi\beta)$. This implies $C \Vdash (R : \mathbf{Pre}\,;\ \partial\mathbf{Abs}) \leq (r : \mathbf{Pre}\,;\ \varphi\beta)$ (we have simply made the two rows agree at r). Because $C, \Gamma \vdash R : \{R : \mathbf{Pre}\,;\ \partial\mathbf{Abs}\}$ holds, HM-SUB yields $C, \Gamma \vdash R : \{r : \mathbf{Pre}\,;\ \varphi\beta\}$. From this fact, it is easy to derive $C, \Gamma \vdash \lambda x.\lambda y.(x\, R) :\ \tau_2$.

Case $\mathbf{c} = ?_r$ and $i = 2$. In this case $\tau_1 = \{r : \varphi\gamma\ ;\ \varphi\beta\}$ and $\tau_2 = (\{r : \mathbf{Pre}\ ;\ \varphi\beta_1\} \rightarrow \varphi\alpha_1) \rightarrow (\{r : \mathbf{Abs}\ ;\ \varphi\beta_2\} \rightarrow \varphi\alpha_2) \rightarrow \varphi\alpha$. By Lemma 4, v is a set R, so $\delta(?_r, v)$ is defined. Let us assume $r \in R$ (the other case is analogous). Then, $\delta(?_r, v)$ is $\lambda x.\lambda y.(x\ R)$. By Lemma 4, we have $C \Vdash (R : \mathbf{Pre}\ ;\ \partial\mathbf{Abs}) \leq (r : \varphi\gamma\ ;\ \varphi\beta)$. This implies, in particular, $\mathbf{Pre} \leq \varphi\gamma$ (we have simply looked up the two rows at r). Because $C \Vdash \varphi(D)$, and by definition of the satisfaction of conditional constraints, we must then have $C \Vdash \varphi\beta \leq \varphi\beta_1$ and $C \Vdash \varphi\alpha_1 \leq \varphi\alpha$. Furthermore, as in the previous case, we have $C, \Gamma \vdash R : \{r : \mathbf{Pre}\ ;\ \varphi\beta\}$. From these facts, it is easy to derive $C, \Gamma \vdash \lambda x.\lambda y.(x\ R) : \tau_2$. □

6. TYPES FOR λ_{sec}

6.1 Indirect Type Systems

Section 5 defined a type system, $\mathcal{S}_i^{rel}$, for λ_{set}. Section 4 defined a translation of λ_{sec} into λ_{set}. Composing the two automatically gives rise to a type system for λ_{sec}, also called $\mathcal{S}_i^{rel}$ for simplicity, whose safety is a direct consequence of Theorems 2 and 3.

Definition 1. Let e be a λ_{sec} expression. By definition, $C, \Gamma \vdash e : \sigma$ holds if and only if $C, \Gamma \vdash (\!| e |\!) : \sigma$ holds.

THEOREM 4. *If $C, \varnothing \vdash e : \sigma$ holds and C is satisfiable, then e does not go wrong.*

Turning type safety into a trivial corollary was the main motivation for basing our approach on a translation. Indeed, because Theorem 2 concerns untyped terms, its proof is straightforward. (The δ-typability lemma established in Section 5.3 does involve types, but is very straightforward.) A direct type safety proof would duplicate most of the steps involved in proving HM(X) correct.

Although the above theorem only mentions type safety, it is possible to also establish a subject reduction result for λ_{sec}. Indeed, according to Lemma 2, subject reduction for λ_{sec} follows directly from subject reduction for λ_{set} and from the fact that administrative expansion $\underset{\sim}{\leftarrow}$ preserves types, which is easy to check.

6.2 Reformulation: Direct Type Systems

Definition 1, although simple, is not a direct definition of typing for λ_{sec}. But a direct type system is desirable, for several reasons. First, given a direct type system, it becomes unnecessary to actually translate expressions down to λ_{set}. Also, with a direct type system, more succinct and intuitive type and judgment forms can be adopted. Finally, understandable type error reporting is much more feasible in a direct type system. Therefore, we define rules which allow typing λ_{sec} expressions without explicitly translating them into λ_{set}. These so-called *direct* or *derived* rules can be obtained in a rather systematic way from the definition of $\mathcal{S}_i^{rel}$ and the definition of the translation, making the direct type safety proof straightforward, by appeal to the pre-existing result in λ_{set} and Theorem 2.

$$\textsc{Var}\quad \frac{\Gamma(x) = \sigma}{p, \varsigma, \Gamma \vdash x : \sigma} \qquad \textsc{Abs}\quad \frac{\star, \varsigma_2, (\Gamma; z : \tau_1 \xrightarrow{\varsigma_2} \tau_2; x : \tau_1) \vdash f : \tau_2}{p, \varsigma_1, \Gamma \vdash \mathsf{fix}\, z.\lambda x.f : \tau_1 \xrightarrow{\varsigma_2} \tau_2}$$

$$\textsc{App}\quad \frac{p, \varsigma, \Gamma \vdash e_1 : \tau_2 \xrightarrow{\varsigma} \tau \qquad p, \varsigma, \Gamma \vdash e_2 : \tau_2}{p, \varsigma, \Gamma \vdash e_1\, e_2 : \tau} \qquad \textsc{Let}\quad \frac{p, \varsigma, \Gamma \vdash e_1 : \sigma \qquad p, \varsigma, (\Gamma; x : \sigma) \vdash e_2 : \tau}{p, \varsigma, \Gamma \vdash \mathsf{let}\, x = e_1\, \mathsf{in}\, e_2 : \tau}$$

$$\forall\ \textsc{Intro}\quad \frac{p, \varsigma, \Gamma \vdash e : \tau \qquad \bar{\alpha} \cap \mathsf{fv}(\varsigma, \Gamma) = \varnothing}{p, \varsigma, \Gamma \vdash e : \forall \bar{\alpha}.\tau} \qquad \forall\ \textsc{Elim}\quad \frac{p, \varsigma, \Gamma \vdash e : \forall \bar{\alpha}.\tau}{p, \varsigma, \Gamma \vdash e : [\bar{\tau}/\bar{\alpha}]\tau} \qquad \textsc{Enable Failure}\quad \frac{p, \{\rho\}, \Gamma \vdash e : \tau \qquad r \notin p}{p, \{\rho\}, \Gamma \vdash \mathsf{enable}\, r\, \mathsf{in}\, e : \tau}$$

$$\textsc{Enable Success}\quad \frac{p, \{r : \mathbf{Pre}\,;\ \rho\}, \Gamma \vdash e : \tau \qquad r \in p}{p, \{r : \varphi\,;\ \rho\}, \Gamma \vdash \mathsf{enable}\, r\, \mathsf{in}\, e : \tau} \qquad \textsc{Check}\quad \frac{p, \{r : \mathbf{Pre}\,;\ \rho\}, \Gamma \vdash e : \tau}{p, \{r : \mathbf{Pre}\,;\ \rho\}, \Gamma \vdash \mathsf{check}\, r\, \mathsf{then}\, e : \tau}$$

$$\textsc{Test}\quad \frac{p, \{r : \mathbf{Pre}\,;\ \rho\}, \Gamma \vdash e_1 : \tau \qquad p, \{r : \mathbf{Abs}\,;\ \rho\}, \Gamma \vdash e_2 : \tau}{p, \{r : \varphi\,;\ \rho\}, \Gamma \vdash \mathsf{test}\, r\, \mathsf{then}\, e_1\, \mathsf{else}\, e_2 : \tau} \qquad \textsc{Sign}\quad \frac{p, \{p : \bar{\varphi}\,;\ \partial\mathbf{Abs}\}, \Gamma \vdash e : \tau}{\star, \{p : \bar{\varphi}\,;\ \rho\}, \Gamma \vdash p.e : \tau}$$

Fig. 10. Typing rules for λ_{sec} derived from $\mathcal{S}_1^=$.

In these rules, the symbols τ and ς range over types of kind *Type*; more specifically, ς is used to represent some security context, that is, a set of available resources. The symbols ρ and φ range over types of kind $Row_\star$ and *Pres*, respectively. The $\star$ symbol in the rules stands for an arbitrary principal. In the source-to-target translation, all functions are given an additional parameter, yielding types of the form $\tau_1 \to \varsigma \to \tau_2$. To recover the more familiar and appealing notation proposed in Skalka and Smith [2000], we define the macro $\tau_1 \xrightarrow{\varsigma} \tau_2 =_{def} \tau_1 \to \varsigma \to \tau_2$.

Figure 10 gives derived rules for $\mathcal{S}_1^=$, the simplest of our type systems. There, all constraints are equations. As a result, all type information can be represented in term form, rather than in constraint form [Sulzmann et al. 1999], provided types are identified modulo the (standard) equational theory for rows. We exploit this fact to give a simple presentation of the derived rules. Type schemes have the form $\forall \bar{\alpha}.\tau$, and judgments have the form $p, \varsigma, \Gamma \vdash e : \sigma$. Although rule Enable Failure naturally arises through the translation, it may be desirable, in practice, to remove it. Thus, any attempt to enable a privilege by a principal who does not own it would result in an immediate static type error.

Figure 11 gives rules for the system derived from $\mathcal{S}_2^\leq$, the most complex element in our array of type systems. Judgments have the form $p, \varsigma, C, \Gamma \vdash e : \sigma$. The most significant differences are the accuracy of the Test rule, reflecting the more precise binding for $?_r$ in Δ_2, and the addition of subtyping constraints.

Because the system presented in Figure 10 is based on unification, it is efficient, easy to implement, and yields readable types. Also, we conjecture that, thanks to the power of row polymorphism, it is flexible enough for many practical uses (see Section 7). Therefore, we will focus on this system in the rest of this article . We prove that this system is correct in Section 6.3.

$$\textsc{Var}\quad \frac{\Gamma(x) = \sigma \qquad C \Vdash \sigma}{p, \varsigma, C, \Gamma \vdash x : \sigma}
\qquad\qquad
\textsc{Sub}\quad \frac{p, \varsigma, C, \Gamma \vdash e : \tau \qquad C \Vdash \tau \leq \tau'}{p, \varsigma, C, \Gamma \vdash e : \tau'}$$

$$\textsc{Abs}\quad \frac{\star, \varsigma_2, C, (\Gamma; z : \tau_1 \xrightarrow{\varsigma_2} \tau_2; x : \tau_1) \vdash f : \tau_2}{p, \varsigma_1, C, \Gamma \vdash \mathsf{fix}\, z.\lambda x.f : \tau_1 \xrightarrow{\varsigma_2} \tau_2}
\qquad\qquad
\textsc{App}\quad \frac{p, \varsigma, C, \Gamma \vdash e_1 : \tau_2 \xrightarrow{\varsigma} \tau \qquad p, \varsigma, C, \Gamma \vdash e_2 : \tau_2}{p, \varsigma, C, \Gamma \vdash e_1\, e_2 : \tau}$$

$$\textsc{Let}\quad \frac{p, \varsigma, C, \Gamma \vdash e_1 : \sigma \qquad p, \varsigma, C, (\Gamma; x : \sigma) \vdash e_2 : \tau}{p, \varsigma, C, \Gamma \vdash \mathsf{let}\, x = e_1\, \mathsf{in}\, e_2 : \tau}
\qquad\qquad
\forall\ \textsc{Intro}\quad \frac{p, \varsigma, C \wedge D, \Gamma \vdash e : \tau \qquad \bar{\alpha} \cap \mathsf{fv}(\varsigma, C, \Gamma) = \varnothing}{p, \varsigma, C \wedge \exists \bar{\alpha}.D, \Gamma \vdash e : \forall \bar{\alpha}[D].\tau}$$

$$\forall\ \textsc{Elim}\quad \frac{p, \varsigma, C, \Gamma \vdash e : \forall \bar{\alpha}[D].\tau \qquad C \Vdash [\bar{\tau}/\bar{\alpha}]D}{p, \varsigma, C, \Gamma \vdash e : [\bar{\tau}/\bar{\alpha}]\tau}
\qquad\qquad
\textsc{Enable Failure}\quad \frac{p, \{\rho\}, C, \Gamma \vdash e : \tau \qquad r \notin p}{p, \{\rho\}, C, \Gamma \vdash \mathsf{enable}\, r\, \mathsf{in}\, e : \tau}$$

$$\textsc{Enable Success}\quad \frac{p, \{r : \mathbf{Pre}\,;\ \rho\}, C, \Gamma \vdash e : \tau \qquad r \in p}{p, \{r : \varphi\,;\ \rho\}, C, \Gamma \vdash \mathsf{enable}\, r\, \mathsf{in}\, e : \tau}
\qquad\qquad
\textsc{Check}\quad \frac{p, \{r : \mathbf{Pre}\,;\ \rho\}, C, \Gamma \vdash e : \tau}{p, \{r : \mathbf{Pre}\,;\ \rho\}, C, \Gamma \vdash \mathsf{check}\, r\, \mathsf{then}\, e : \tau}$$

$$\textsc{Test}\quad \frac{\begin{array}{c} p, \{r : \mathbf{Pre}\,;\ \rho_1\}, C, \Gamma \vdash e_1 : \tau_1 \\ p, \{r : \mathbf{Abs}\,;\ \rho_2\}, C, \Gamma \vdash e_2 : \tau_2 \qquad C \Vdash \text{if } \mathbf{Pre} \leq \varphi \text{ then } \rho \leq \rho_1 \\ C \Vdash \text{if } \mathbf{Abs} \leq \varphi \text{ then } \rho \leq \rho_2 \qquad C \Vdash \text{if } \mathbf{Pre} \leq \varphi \text{ then } \tau_1 \leq \tau \qquad C \Vdash \text{if } \mathbf{Abs} \leq \varphi \text{ then } \tau_2 \leq \tau \end{array}}{p, \{r : \varphi\,;\ \rho\}, C, \Gamma \vdash \mathsf{test}\, r\, \mathsf{then}\, e_1\, \mathsf{else}\, e_2 : \tau}$$

$$\textsc{Sign}\quad \frac{p, \{p : \bar{\varphi}\,;\ \partial\mathbf{Abs}\}, C, \Gamma \vdash e : \tau}{\star, \{p : \bar{\varphi}\,;\ \rho\}, C, \Gamma \vdash p.e : \tau}$$

Fig. 11. Typing rules for λ_{sec} derived from $\mathcal{S}_2^{\leq}$.

6.3 Direct Type Correctness

In this section, we prove the correctness of the type system derived from $\mathcal{S}_1^{=}$, that is, we prove Lemma 11. We begin by proving soundness of the derived system with respect to $S_1^{=}$.

Lemma 8. *$p, \varsigma, \Gamma \vdash e : \sigma$ implies* $\mathbf{true}, (\Gamma; s : \varsigma) \vdash [\![e]\!]_p : \sigma$.

Proof. By structural induction on the derivation of $p, \varsigma, \Gamma \vdash e : \sigma$. Let Γ' stand for $(\Gamma; s : \varsigma)$.

Case Var. In this case, e is a variable x. Because s is a distinguished variable, we have $x \neq s$, so $\Gamma(x)$ and $\Gamma'(x)$ coincide. Furthermore, $[\![x]\!]_p$ is x. The result follows by hm-Var.

Case Abs. In this case, e is $\mathsf{fix}\, z.\lambda x.f$, σ is $\tau_1 \xrightarrow{\varsigma'} \tau_2$ and $p', \varsigma', (\Gamma; z : \sigma; x : \tau_1) \vdash f : \tau_2$ is derivable. By the induction hypothesis, $\mathbf{true}, (\Gamma; z : \sigma; x : \tau_1; s : \varsigma') \vdash [\![f]\!]_{p'} : \tau_2$ is derivable. This judgment can also be written $\mathbf{true}, (\Gamma'; z : \sigma; x : \tau_1; s : \varsigma') \vdash [\![f]\!] : \tau_2$. Thus, $\mathbf{true}, \Gamma' \vdash \mathsf{fix}\, z.\lambda x.\lambda s.[\![f]\!] : \tau_1 \rightarrow \varsigma' \rightarrow \tau_2$ is derivable by two applications of hm-Abs. Given the definition of $[\![e]\!]_p$ in this case, this was the goal.

Case App. In this case, $e = e_1\, e_2$, $\sigma = \tau$ and $p, \varsigma, \Gamma \vdash e_1 : \tau_2 \xrightarrow{\varsigma} \tau$ and $p, \varsigma, \Gamma \vdash e_2 : \tau_2$ are derivable. By the induction hypothesis, $\mathbf{true}, \Gamma' \vdash [\![e_1]\!]_p : \tau_2 \to \varsigma \to \tau$ and $\mathbf{true}, \Gamma' \vdash [\![e_2]\!]_p : \tau_2$ are derivable. Furthermore, $\mathbf{true}, \Gamma' \vdash s : \varsigma$ holds by hm-Var. Hence, $\mathbf{true}, \Gamma' \vdash [\![e_1]\!]_p\, [\![e_2]\!]_p\, s : \tau_2$ is derivable by two applications of hm-App. Given the definition of $[\![e]\!]_p$ in this case, this was the goal.

Case Let. In this case, $e = \mathsf{let}\, x = e_1 \,\mathsf{in}\, e_2$, and $p, \varsigma, \Gamma \vdash e_1 : \sigma'$ and $p, \varsigma, (\Gamma; x : \sigma') \vdash e_2 : \sigma$ are derivable. By the induction hypothesis, $\mathbf{true}, \Gamma' \vdash [\![e_1]\!]_p : \sigma'$ and $\mathbf{true}, (\Gamma'; x : \sigma') \vdash [\![e_2]\!]_p : \sigma$ hold. The result follows by hm-Let and by definition of $[\![e]\!]_p$ in this case.

Case $\forall$ Intro. In this case, $\sigma = \forall\bar{\alpha}[\mathbf{true}].\tau$ where $\bar{\alpha} \cap \mathrm{fv}(\varsigma, \Gamma) = \varnothing$ and $p, \varsigma, \Gamma \vdash e : \tau$ is derivable. By the induction hypothesis, $\mathbf{true}, \Gamma' \vdash [\![e]\!]_p : \tau$ is derivable. Furthermore, we have $\bar{\alpha} \cap \mathrm{fv}(\mathbf{true}, \Gamma') = \varnothing$. Thus, by hm-$\forall$ Intro, $\mathbf{true}, \Gamma' \vdash [\![e]\!]_p : \sigma$ is derivable. We have implicitly used the equivalences $\mathbf{true} \equiv \mathbf{true} \wedge \mathbf{true}$ and $\mathbf{true} \equiv \exists\bar{\alpha}.\mathbf{true}$.

Case $\forall$ Elim. In this case, $\sigma = [\bar{\tau}/\bar{\alpha}]\tau$ and $p, \varsigma, \Gamma \vdash e : \forall\bar{\alpha}[\mathbf{true}].\tau$ is derivable. By the induction hypothesis, $\mathbf{true}, \Gamma' \vdash [\![e]\!]_p : \forall\bar{\alpha}[\mathbf{true}].\tau$ is derivable. Furthermore, $[\bar{\tau}/\bar{\alpha}]\mathbf{true}$ is $\mathbf{true}$, so the result follows by hm-$\forall$ Elim.

Case Enable Failure. In this case, $e = \mathsf{enable}\, r \,\mathsf{in}\, e'$ where $r \notin p$ so that $\{r\} \cap p = \varnothing$, $\varsigma = \{\rho\}$, $\sigma = \tau$ and $p, \varsigma, \Gamma \vdash e' : \tau$ is derivable. Now, by definition of Δ_1, by hm-Const and hm-$\forall$ Elim, $\mathbf{true}, \Gamma' \vdash \vee_{\varnothing} : \varsigma \to \varsigma$ is derivable. Furthermore, $\mathbf{true}, \Gamma' \vdash s : \varsigma$ follows from hm-Var. Therefore, hm-App yields $\mathbf{true}, \Gamma' \vdash s \vee \varnothing : \varsigma$. The induction hypothesis yields $\mathbf{true}, \Gamma' \vdash [\![e']\!]_p : \tau$, so also $\mathbf{true}, (\Gamma'; s : \varsigma) \vdash [\![e']\!]_p : \tau$. The result follows by hm-Let and the definition of $[\![e]\!]_p$ in this case.

Case Enable Success. In this case, $e = \mathsf{enable}\, r \,\mathsf{in}\, e'$ where $r \in p$ so that $\{r\} \cap p = \{r\}$, $\varsigma = \{r : \varphi; \rho\}$, $\sigma = \tau$ and $p, \{r : \mathbf{Pre}; \rho\}, \Gamma \vdash e' : \tau$ is derivable. Now, by definition of Δ_1, by hm-Const and hm-$\forall$ Elim, $\mathbf{true}, \Gamma' \vdash \vee_{\{r\}} : \{r : \varphi; \rho\} \to \{r : \mathbf{Pre}; \rho\}$ is derivable. Furthermore, $\mathbf{true}, \Gamma' \vdash s : \varsigma$ follows from hm-Var. Therefore, hm-App yields $\mathbf{true}, \Gamma' \vdash s \vee \{r\} : \{r : \mathbf{Pre}; \rho\}$. The induction hypothesis yields $\mathbf{true}, (\Gamma; s : \{r : \mathbf{Pre}; \rho\}) \vdash [\![e']\!]_p : \tau$, so also $\mathbf{true}, (\Gamma'; s : \{r : \mathbf{Pre}; \rho\}) \vdash [\![e']\!]_p : \tau$. The result follows by hm-Let and the definition of $[\![e]\!]_p$ in this case.

Case Check. In this case $e = \mathsf{check}\, r \,\mathsf{then}\, e'$ and $\sigma = \tau$, $\varsigma = \{r : \mathbf{Pre}; \rho\}$ and $p, \varsigma, \Gamma \vdash e' : \tau$ is derivable. Now, by definition of Δ_1, by hm-Const, hm-$\forall$ Elim, hm-Var and hm-App, $\mathbf{true}, \Gamma' \vdash s.r : \varsigma$ is derivable. By the induction hypothesis, $\mathbf{true}, \Gamma' \vdash [\![e']\!]_p : \tau$ is derivable, so also $\mathbf{true}, (\Gamma'; \_ : \varsigma) \vdash [\![e']\!]_p : \tau$, if $\_$ is a variable that does not appear free in e'. The result follows by hm-Let and the definition of $[\![e]\!]_p$ in this case.

Case Test. In this case $e = \mathsf{test}\, r \,\mathsf{then}\, e_1 \,\mathsf{else}\, e_2$ and $\sigma = \tau$, $\varsigma = \{r : \varphi;\ \rho\}$ and $p, \{r : \mathbf{Pre};\ \rho\}, \Gamma \vdash e_1 : \tau$ and $p, \{r : \mathbf{Abs};\ \rho\}, \Gamma \vdash e_2 : \tau$ are derivable. By the induction hypothesis, we have $\mathbf{true}, (\Gamma; s : \{r : \mathbf{Pre};\ \rho\}) \vdash [\![e_1]\!]_p : \tau$. By hm-Abs, this implies $\mathbf{true}, \Gamma \vdash \lambda s.[\![e_1]\!]_p : \{r : \mathbf{Pre};\ \rho\} \to \tau$. By weakening, we also have $\mathbf{true}, \Gamma' \vdash \lambda s.[\![e_1]\!]_p : \{r : \mathbf{Pre};\ \rho\} \to \tau$. Similarly, $\mathbf{true}, \Gamma' \vdash \lambda s.[\![e_2]\!]_p : \{r : \mathbf{Abs};\ \rho\} \to \tau$ holds. The result follows by definition of Δ_1, by hm-Const, hm-$\forall$ Elim, hm-Var, hm-App and by definition of $[\![e]\!]_p$ in this case.

Case SIGN. In this case $e = p'.e'$, $\varsigma = \{p' : \bar{\varphi}\,;\ \rho\}$, $\sigma = \tau$ and $p', \varsigma', \Gamma \vdash e' : \tau$ is derivable, where $\varsigma' = \{p' : \bar{\varphi}\,;\ \partial\mathbf{Abs}\}$. By the induction hypothesis, $\mathbf{true}, (\Gamma; s : \varsigma') \vdash [\![e']\!]_{p'} : \tau$ holds, so also $\mathbf{true}, (\Gamma'; s : \varsigma') \vdash [\![e']\!]_{p'} : \tau$. Now, by definition of Δ_1, by HM-CONST, HM-∀ ELIM, HM-VAR and HM-APP, $\mathbf{true}, \Gamma' \vdash s \wedge p' : \varsigma'$ holds. The result follows by HM-LET and the definition of $[\![e]\!]_p$ in this case. □

Our next task is to prove completeness of the derived type system with respect to $\mathcal{S}_1^=$. We begin with a *normalization* result analogous to the one proved in Skalka and Pottier [2002].

LEMMA 9. *If $C, \Gamma \vdash e : \tau$ holds, then it may be derived via an instance of* SUB *from a judgment $C, \Gamma \vdash e : \tau'$, which itself follows from an instance of a syntax-directed rule and at most one instance of* ∀ ELIM.

We may now proceed to demonstrate completeness. In this lemma, we abbreviate type schemes $\forall\bar{\alpha}[\mathbf{true}].\tau$ as $\forall\bar{\alpha}.\tau$ and judgements $\mathbf{true}, \Gamma \vdash e : \sigma$ as $\Gamma \vdash e : \sigma$, omitting the trivial requirement $\mathbf{true} \Vdash \mathbf{true}$ from instances of ∀ ELIM and VAR.

LEMMA 10. *$(\Gamma; s : \varsigma) \vdash [\![e]\!]_p : \tau$ implies $p, \varsigma, \Gamma \vdash e : \tau$.*

PROOF. In this proof, we will write $\tau = \tau'$ for $\mathbf{true} \Vdash \tau = \tau'$, which amounts to identifying types modulo the equational theory on rows and allows us to ignore instances of HM-SUB in the derivation d of $(\Gamma; s : \varsigma) \vdash [\![e]\!]_p : \tau$. By Lemma 9, we may assume that d ends with a syntax-directed rule and at most one instance of ∀ ELIM. The proof proceeds by induction on the structure of e and analysis of the derivation d. Let $\Gamma' = (\Gamma; s : \varsigma)$.

Case $e = [\![e]\!]_p = x$. By assumption, we have $x \neq s$. The derivation d must involve HM-VAR possibly followed by HM-∀ ELIM. As a result, τ must be of the form $[\bar{\tau}/\bar{\alpha}]\tau'$, where $\Gamma(x) = \forall\bar{\alpha}.\tau'$. By VAR and ∀ ELIM, this implies $p, \varsigma, \Gamma \vdash x : [\bar{\tau}/\bar{\alpha}]\tau'$. Therefore, this case holds.

Case $e = \mathsf{fix}\, z.\lambda x.f$ *and* $[\![e]\!]_p = \mathsf{fix}\, z.\lambda x.\lambda s.[\![f]\!]_p$. By Lemma 9, we may assume that d ends with two instances of HM-ABS, as follows:

$$\dfrac{\dfrac{\Gamma'; z : \tau_1 \to \varsigma' \to \tau_2; x : \tau_1; s : \varsigma' \vdash [\![f]\!]_p : \tau_2}{\Gamma'; z : \tau_1 \to \varsigma' \to \tau_2; x : \tau_1 \vdash \lambda s.[\![f]\!]_p : \varsigma' \to \tau_2}}{\Gamma' \vdash \mathsf{fix}\, z.\lambda x.\lambda s.[\![f]\!]_p : \tau_1 \to \varsigma' \to \tau_2}\,.$$

Here, τ is $\tau_1 \to \varsigma' \to \tau_2$. Now, we have:

$$(\Gamma'; z : \tau_1 \to \varsigma' \to \tau_2; x : \tau_1; s : \varsigma') = (\Gamma; z : \tau_1 \to \varsigma' \to \tau_2; x : \tau_1; s : \varsigma').$$

This allows applying the induction hypothesis, yielding $p, \varsigma', (\Gamma; z : \tau_1 \to \varsigma' \to \tau_2; x : \tau_1) \vdash f : \tau_2$. By ABS, this implies $p, \varsigma, \Gamma \vdash \mathsf{fix}\, z.\lambda x.f : \tau_1 \to \varsigma' \to \tau_2$.

Case $e = e_1 e_2$ and $[\![e]\!]_p = [\![e_1]\!]_p [\![e_2]\!]_p s$. By Lemma 9, we may assume that d ends with two instances of HM-APP, as follows:

$$\dfrac{\dfrac{\Gamma' \vdash [\![e_1]\!]_p : \tau' \to \varsigma \to \tau \qquad \Gamma' \vdash [\![e_2]\!]_p : \tau'}{\Gamma' \vdash [\![e_1]\!]_p [\![e_2]\!]_p : \varsigma \to \tau} \qquad \dfrac{\Gamma'(s) = \varsigma}{\Gamma' \vdash s : \varsigma}}{\Gamma' \vdash [\![e_1]\!]_p [\![e_2]\!]_p s : \tau}\,.$$

By the induction hypothesis, we have $p, \varsigma, \Gamma \vdash e_1 : \tau' \to \varsigma \to \tau$ and $p, \varsigma, \Gamma \vdash e_2 : \tau'$. The judgment $p, \varsigma, \Gamma \vdash e_1 e_2 : \tau$ follows by App.

Case $e = \mathsf{let}\, x = e_1 \,\mathsf{in}\, e_2$ *and* $[\![e]\!]_p = \mathsf{let}\, x = [\![e_1]\!]_p \,\mathsf{in}\, [\![e_2]\!]_p$. Then, d ends with an instance of hm-Let:

$$\frac{\Gamma' \vdash [\![e_1]\!]_p : \forall\bar{\alpha}[D].\tau' \qquad (\Gamma; s : \varsigma; x : \forall\bar{\alpha}[D].\tau') \vdash [\![e_2]\!]_p : \tau}{\Gamma' \vdash \mathsf{let}\, x = [\![e_1]\!]_p \,\mathsf{in}\, [\![e_2]\!]_p : \tau}.$$

Here, we have $\exists\bar{\alpha}.D \equiv \mathbf{true}$, which implies that the constraint D—a system of equations—admits a most general unifier. In that case, the type scheme $\forall\bar{\alpha}[D].\tau'$ can be shown equivalent to an unconstrained type scheme, so we may assume, without loss of generality, that D is in fact **true**. We may further assume, without loss of generality, that the left-hand premise is an instance of hm-$\forall$ Intro:

$$\frac{\Gamma' \vdash [\![e_1]\!]_p : \tau' \qquad \bar{\alpha} \cap \mathrm{fv}(\Gamma') = \varnothing}{\Gamma' \vdash [\![e_1]\!]_p : \forall\bar{\alpha}.\tau'}$$

The induction hypothesis yields $p, \varsigma, \Gamma \vdash e_1 : \tau'$. We have $\bar{\alpha} \cap \mathrm{fv}(\varsigma, \Gamma) = \varnothing$, so, by $\forall$ Intro, we obtain $p, \varsigma, \Gamma \vdash e_1 : \forall\bar{\alpha}.\tau'$. Since $x \neq s$, we have $(\Gamma; s : \varsigma; x : \forall\bar{\alpha}.\tau') = (\Gamma; x : \forall\bar{\alpha}.\tau'; s : \varsigma)$, therefore the induction hypothesis yields $p, \varsigma, (\Gamma; x : \forall\bar{\alpha}.\tau') \vdash e_2 : \tau$. The result follows by Let.

Case $e = \mathsf{enable}\, r \,\mathsf{in}\, e'$ *and* $[\![e]\!]_p = \mathsf{let}\, s = s \vee (\{r\} \cap p) \,\mathsf{in}\, [\![e']\!]_p$. By Lemma 9 and definition of Δ_1, the derivation d must be of the following form, where $R = \{r\} \cap p$, $\varsigma = \{R : \bar{\varphi}\, ;\, \rho\}$ and $\varsigma' = \{R : \mathbf{Pre}\, ;\, \rho\}$:

$$\frac{\dfrac{\dfrac{\Gamma' \vdash \vee_R : \varsigma \to \varsigma' \qquad \Gamma' \vdash s : \varsigma}{\Gamma' \vdash s \vee R : \varsigma'} \qquad \bar{\alpha} \cap \mathrm{fv}(\Gamma') = \varnothing}{\Gamma' \vdash s \vee R : \forall\bar{\alpha}.\varsigma'} \qquad \Gamma'; s : \forall\bar{\alpha}.\varsigma' \vdash [\![e']\!]_p : \tau}{\Gamma' \vdash \mathsf{let}\, s = s \vee R \,\mathsf{in}\, [\![e']\!]_p : \tau}.$$

Since ς appears in Γ', the free type variables of ρ are free in Γ' as well, so the free type variables of ς' are free in Γ'. As a result, the type scheme $\forall\bar{\alpha}.\varsigma'$ is equivalent to the monotype ς'. We will thus assume, without loss of generality, that $\bar{\alpha}$ is empty. Since $(\Gamma'; s : \varsigma') = (\Gamma; s : \varsigma')$, the induction hypothesis yields $p, \varsigma', \Gamma \vdash e' : \tau$. As a result, $p, \varsigma, \Gamma \vdash \mathsf{enable}\, r \,\mathsf{in}\, e' : \tau$ is derivable by Enable Failure if $r \notin p$ and by Enable Success if $r \in p$.

Case $e = \mathsf{check}\, r \,\mathsf{then}\, e'$ *and* $[\![e]\!]_p = \mathsf{let}\, \_ = s.r \,\mathsf{in}\, [\![e']\!]_p$. By Lemma 9 and definition of Δ_1, the derivation d must be of the following form, where $\varsigma = \{r : \mathbf{Pre}\, ;\, \rho\}$:

$$\frac{\dfrac{\dfrac{\Gamma' \vdash .r : \varsigma \to \varsigma \qquad \Gamma' \vdash s : \varsigma}{\Gamma' \vdash s.r : \varsigma} \qquad \bar{\alpha} \cap \mathrm{fv}(\Gamma') = \varnothing}{\Gamma' \vdash s.r : \forall\bar{\alpha}.\varsigma} \qquad \Gamma'; \_ : \forall\bar{\alpha}.\varsigma \vdash [\![e']\!]_p : \tau}{\Gamma' \vdash \mathsf{let}\, \_ = s.r \,\mathsf{in}\, [\![e']\!]_p :}.$$

Since _ does not occur in e', by weakening, we have $\Gamma' \vdash [\![e']\!]_p : \tau$. As a result, the induction hypothesis yields $p, \varsigma, \Gamma \vdash e' : \tau$. Thus, $p, \varsigma, \Gamma \vdash \mathsf{check}\, r \,\mathsf{then}\, e' : \tau$ is derivable by Check.

Case $e = \mathsf{test}\, r\, \mathsf{then}\, e_1\, \mathsf{else}\, e_2$ *and* $[\![e]\!]_p = s?r\,(\lambda s.[\![e_1]\!]_p)(\lambda s.[\![e_2]\!]_p)$. By Lemma 9 and definition of Δ_1, d must be of the following form, where $\varsigma = \{r : \varphi\,;\ \rho\}$:

$$\dfrac{\begin{array}{c}\Gamma' \vdash ?_r : \{r : \varphi\,;\ \rho\} \to (\{r : \mathbf{Pre}\,;\ \rho\} \to \tau) \to (\{r : \mathbf{Abs}\,;\ \rho\} \to \tau) \to \tau \\ \Gamma' \vdash s : \{r : \varphi\,;\ \rho\}\end{array}}{\Gamma' \vdash s?r : (\{r : \mathbf{Pre}\,;\ \rho\} \to \tau) \to (\{r : \mathbf{Abs}\,;\ \rho\} \to \tau) \to \tau} \quad (1)$$

$$\dfrac{\Gamma'; s : \{r : \mathbf{Pre}\,;\ \rho\} \vdash [\![e_1]\!]_p : \tau}{\Gamma' \vdash \lambda s.[\![e_1]\!]_p : \{r : \mathbf{Pre}\,;\ \rho\} \to \tau} \quad (2) \qquad \dfrac{\Gamma'; s : \{r : \mathbf{Abs}\,;\ \rho\} \vdash [\![e_2]\!]_p : \tau}{\Gamma' \vdash \lambda s.[\![e_2]\!]_p : \{r : \mathbf{Abs}\,;\ \rho\} \to \tau} \quad (3)$$

$$\dfrac{\dfrac{(1) \qquad (2)}{\Gamma' \vdash s?r\,(\lambda s.[\![e_1]\!]_p) : (\{r : \mathbf{Abs}\,;\ \rho\} \to \tau) \to \tau} \qquad (3)}{\Gamma' \vdash s?r\,(\lambda s.[\![e_1]\!]_p)(\lambda s.[\![e_2]\!]_p) : \tau}\ .$$

By the induction hypothesis, $p, \{r : \mathbf{Pre}\,;\ \rho\}, \Gamma \vdash e_1 : \tau$ and $p, \{r : \mathbf{Abs}\,;\ \rho\}, \Gamma \vdash e_2 : \tau$ hold. The judgment $p, \{r : \varphi\,;\ \rho\}, \Gamma \vdash \mathsf{test}\, r\, \mathsf{then}\, e_1\, \mathsf{else}\, e_2 : \tau$ follows by TEST.

Case $e = p'.e'$ *and* $[\![e]\!]_p = \mathsf{let}\, s = s \wedge p'\, \mathsf{in}\, [\![e']\!]_{p'}$. By Lemma 9 and definition of Δ_1, the derivation d must be of the following form, where $\varsigma = \{p' : \bar\varphi\,;\ \rho\}$ and $\varsigma' = \{p' : \bar\varphi\,;\ \partial\mathbf{Abs}\}$:

$$\dfrac{\dfrac{\dfrac{\Gamma' \vdash \wedge_{p'} : \varsigma \to \varsigma' \qquad \Gamma' \vdash s : \varsigma}{\Gamma' \vdash s \wedge p' : \varsigma'} \qquad \bar\alpha \cap \mathsf{fv}(\Gamma') = \varnothing}{\Gamma' \vdash s \wedge p' : \forall\bar\alpha.\varsigma'} \qquad \Gamma'; s : \forall\bar\alpha.\varsigma' \vdash [\![e']\!]_{p'} : \tau}{\Gamma' \vdash \mathsf{let}\, s = s \wedge p'\, \mathsf{in}\, [\![e']\!]_{p'} : \tau}\ .$$

Since ς appears in Γ', the free type variables of $\bar\varphi$ are free in Γ' as well, so the free type variables of ς' are free in Γ'. As a result, the type scheme $\forall\bar\alpha.\varsigma'$ is equivalent to the monotype ς'. We will thus assume, without loss of generality, that $\bar\alpha$ is empty. Since $(\Gamma'; s : \varsigma') = (\Gamma; s : \varsigma')$, the induction hypothesis yields $p', \varsigma', \Gamma \vdash e' : \tau$. As a result, $p, \varsigma, \Gamma \vdash p'.e' : \tau$ is derivable by SIGN. □

We are now ready to demonstrate correctness of the derived type system.

LEMMA 11. *nobody*, $\{\delta\mathbf{Abs}\}, \varnothing \vdash e : \tau$ *holds for some* τ *if and only if* $C, \varnothing \vdash (\!|\, e \,|\!) : \tau$ *holds for some satisfiable* C *and for some* τ.

PROOF. Suppose on the one hand that *nobody*, $\{\delta\mathbf{Abs}\}, \varnothing \vdash e : \tau$ holds. By Lemma 8, we have $\mathbf{true}, s : \{\partial\mathbf{Abs}\} \vdash [\![e]\!]_{nobody} : \tau$. Now, by definition of Δ_1 and by CONST, we have $\mathbf{true}, \varnothing \vdash \varnothing : \{\partial\mathbf{Abs}\}$. By the substitution lemma for HM(X), which is proved in Skalka and Pottier [2002], this leads to $\mathbf{true}, \varnothing \vdash [\![e]\!]_{nobody}[\varnothing/s] : \tau$. Because $\mathbf{true}$ is satisfiable and because $[\![e]\!]_{nobody}[\varnothing/s]$ is $(\!|\, e \,|\!)$, this yields the goal.

Suppose on the other hand that $C, \varnothing \vdash (\!|\, e \,|\!) : \tau$ holds, where C is satisfiable. Because C is satisfiable, it admits a unifier φ, which has the property that the constraint $\varphi(C)$ is equivalent to $\mathbf{true}$. Thus, by the substitution lemma, we have that $\mathbf{true}, \varnothing \vdash (\!|\, e \,|\!) : \varphi(\tau)$ holds. Now, as above, we have $(\!|\, e \,|\!) = [\![e]\!]_{nobody}[\varnothing/s]$ and $\mathbf{true}, \varnothing \vdash \varnothing : \{\partial\mathbf{Abs}\}$. By a simple inverse substitution lemma, which we do not explicitly establish here, this implies $\mathbf{true}, s : \{\partial\mathbf{Abs}\} \vdash [\![e]\!]_{nobody} : \varphi(\tau)$. The result follows by Lemma 10. □

In other words, Lemma 11 states that a closed λ_{sec} program e is well-typed in the derived type system under the initial principal *nobody* and the empty security context $\{\partial\mathbf{Abs}\}$ if and only if $(\!| e |\!)$ is well-typed in the original type system. Furthermore, by Theorem 4, such programs cannot go wrong.

7. EXAMPLES

In this section, we give examples which illustrate the expressivity (and limitations) of our type system. These examples facilitate a discussion of the differences between the variants of the system, yielding insights into the possible tradeoffs between precision and cost.

7.1 Security Wrappers

A library writer often needs to surround numerous internal functions with "boilerplate" security code before making them accessible. To avoid redundancy, it seems desirable to allow the definition of generic *security wrappers*. When applied to a function, a wrapper returns a new function that has the same computational meaning but different security requirements.

Assume given a principal $p = \{r, s\}$. Here are two wrappers likely to be of use to this principal:

$$\begin{aligned} enable_r &= \lambda f.p.\lambda x.p.\mathsf{enable}\, r \,\mathsf{in}\, f\ x \\ require_r &= \lambda f.p.\lambda x.p.\mathsf{check}\, r \,\mathsf{then}\, f\ x. \end{aligned}$$

In system $\mathcal{S}_1^=$, these wrappers receive the following (most general) type schemes. All of the type variables which appear in them are universally quantified, so we do not give the quantifier prefix explicitly.

$$enable_r : \forall \ldots (\alpha_1 \xrightarrow{\{r:\mathbf{Pre}\,;\, s:\gamma_1\,;\, \partial\mathbf{Abs}\}} \alpha_2) \xrightarrow{\{\beta_1\}} (\alpha_1 \xrightarrow{\{r:\gamma_2\,;\, s:\gamma_1\,;\, \beta_2\}} \alpha_2)$$

$$require_r : \forall \ldots (\alpha_1 \xrightarrow{\{r:\mathbf{Pre}\,;\, s:\gamma_1\,;\, \partial\mathbf{Abs}\}} \alpha_2) \xrightarrow{\{\beta_1\}} (\alpha_1 \xrightarrow{\{r:\mathbf{Pre}\,;\, s:\gamma_1\,;\, \beta_2\}} \alpha_2)$$

These types are very similar; they may be read as follows. Both wrappers expect a function f which allows that r be enabled ($r : \mathbf{Pre}$), that is, one which *either* requires r to be enabled, *or* doesn't care about its status. (Indeed, as in ML, the type of the actual argument may be more general than that of the formal.) They return a new function with identical domain and codomain (α_1, α_2), which works regardless of r's status (*enable*$_r$ yields $r : \gamma_2$) or requires r to be enabled (*require*$_r$ yields $r : \mathbf{Pre}$). The new function retains f's expectations about s ($s : \gamma_1$). f must not require any further privileges ($\partial\mathbf{Abs}$), because it is invoked by p, which enjoys privileges r and s only.

These polymorphic types are very expressive. Our main concern is that, even though the privilege s is not mentioned in the *code* of these wrappers, it does appear in their *type*. More generally, every privilege available to p may show up in the type of a function written on behalf of principal p, which may lead to very verbose types. An appropriate type abbreviation mechanism may be able to address this problem; this is left as a subject for future work.

7.2 Use and Types of Security Tests

In this section, we discuss two typical programming idioms involving test. One (arguably the most common) is very simple, and may be typed in $\mathcal{S}_1^=$. The other is more complex and requires at least $\mathcal{S}_2^=$. We take this opportunity to discuss various problems related to the interpretation of conditional constraints.

Imagine an operating system with two kinds of processes, root processes and user processes. Killing a user process is always allowed, while killing a root process requires the privilege k. At least one distinguished principal *root* has this privilege. The system functions which perform the killing are implemented by *root*, as follows:

$$\begin{aligned} kill &= \lambda(p : proc).root.\mathsf{check}\,k\ \mathsf{then}\ \cdots \quad \textit{– kill the process} \\ killIfUser &= \lambda(p : proc).root.\cdots \quad \textit{– kill the process if it is user-level.} \end{aligned}$$

In system $\mathcal{S}_1^=$, these functions receive the following (most general) types:

$$\begin{aligned} kill &: \forall\beta.proc \xrightarrow{\{k:\mathbf{Pre}\,;\,\beta\}} unit \\ killIfUser &: \forall\gamma\beta.proc \xrightarrow{\{k:\gamma\,;\,\beta\}} unit. \end{aligned}$$

The first function can be called only if it can be statically proven that the privilege k is enabled. The second one, on the other hand, can be called at any time, but will never kill a root process. To complement these functions, it may be desirable to define a function which provides a "best attempt" given the current (dynamic) security context. This may be done by dynamically checking whether the privilege is enabled, then calling the appropriate function:

$$\begin{aligned} tryKill = \lambda(p : proc)&.root. \\ &\mathsf{test}\,k\ \mathsf{then}\,kill(p)\ \mathsf{else}\,killIfUser(p). \end{aligned}$$

This function is well typed in system $\mathcal{S}_1^=$. Indeed, within the first branch of the test construct, it is statically known that the privilege k must be enabled; this is why the subexpression $kill(p)$ is well typed. The inferred type shows that *tryKill* does not have any security requirements:

$$tryKill : \forall\gamma\beta.proc \xrightarrow{\{k:\gamma\,;\,\beta\}} unit.$$

The sensitive action $kill(p)$ is performed within the lexical scope of the test construct, which is why it is easily seen to be safe. However, one can also move it outside of the scope, as follows:

$$\begin{aligned} tryKill' \triangleq\ &\lambda(p : proc).root. \\ &\quad \mathsf{let}\,action = \mathsf{test}\,k\ \mathsf{then}\,kill\ \mathsf{else}\,killIfUser\ \mathsf{in}\,action(p). \end{aligned}$$

Here, the dynamic security check yields a closure, whose behavior depends on the check's outcome. It can be passed on and used in further computations. Such a programming idiom is useful in practice, because it allows hoisting a security check out of a loop. For instance, if we were to kill a set of processes, instead of a single one, we would apply *action* successively to each element of the set. Thus, only one security check would have to be performed, regardless of the number of processes in the set.

Is *tryKill'* also well typed? This is more subtle. In $\mathcal{S}_1^{rel}$, the two branches of a test construct must receive the same type. Because the function *kill* requires a non-trivial security context, it is conservatively assumed that *action* may do so as well. As a result, in (say) $\mathcal{S}_1^{=}$, *tryKill'* has (most general) type $\forall\beta.proc \to \{k : \mathbf{Pre}\,;\ \beta\} \to unit$, just as *kill*. Thus, it is well typed, but its type is more restrictive than expected.

To solve this problem, we need to keep track of the fact that the behavior (i.e., the type) of *action* depends on the outcome of the test, that is, on whether the privilege k is enabled. This is precisely the reason for moving to the column $i = 2$ in our array of type systems. In this column, the result of a test construct is described by conditional constraints, which encode the desired dependency. Indeed, in $\mathcal{S}_2^{=}$, *tryKill'* has (most general) inferred type

$$\begin{array}{l}
\forall \ldots . proc \to \{k : \gamma_1\,;\ \beta_1\} \to \alpha \\
\text{where} \\
\quad \text{if } \mathbf{Abs} = \gamma_1, \text{ then } \partial\mathbf{Abs} = \beta_2 \\
\quad \text{if } \mathbf{Pre} = \gamma_1, \text{ then } \partial\mathbf{Abs} = \beta_3 \\
\quad \text{if } \mathbf{Abs} = \gamma_1, \text{ then } proc \to \{k : \gamma_1\,;\ \partial\mathbf{Abs}\} \to \alpha = proc \to \{k : \gamma_2\,;\ \beta_4\} \to unit \\
\quad \text{if } \mathbf{Pre} = \gamma_1, \text{ then } proc \to \{k : \gamma_1\,;\ \partial\mathbf{Abs}\} \to \alpha = proc \to \{k : \mathbf{Pre}\,;\ \beta_5\} \to unit.
\end{array}$$

The four conditional constraints are generated by Test (see Figure 11). Of course, the meaning of such a constrained type scheme is quite obscure, but it is possible to simplify it, as follows. First, because there is only one occurrence of the variable β_2, this variable can be quantified locally. That is, the first conditional constraint can be written

$$\text{if } \mathbf{Abs} = \gamma_1, \text{ then } \exists\beta_2.(\partial\mathbf{Abs} = \beta_2)$$

It is now evident that this constraint is a tautology—that is, it is equivalent to **true**—so it can be suppressed. The second constraint can be suppressed in a similar way. Then, the third and fourth constraints, whose conclusions are equations between terms of similar structure, can be decomposed into a conjunction of conditional constraints whose conclusions are equations between atomic terms. Performing this decomposition and again suppressing tautological constraints, we obtain

$$\begin{array}{l}
\forall \ldots . proc \to \{k : \gamma_1\,;\ \beta_1\} \to \alpha \\
\text{where} \\
\quad \text{if } \mathbf{Abs} = \gamma_1, \text{ then } \alpha = unit \\
\quad \text{if } \mathbf{Pre} = \gamma_1, \text{ then } \gamma_1 = \mathbf{Pre} \\
\quad \text{if } \mathbf{Pre} = \gamma_1, \text{ then } \alpha = unit.
\end{array}$$

The second constraint above is again a tautology (of a different kind) and may be suppressed. Thus, the simplification process yields

$$\begin{array}{l}\forall \ldots.proc \rightarrow \{k : \gamma_1 \,;\, \beta_1\} \rightarrow \alpha \\ \text{where} \\ \quad \text{if } \mathbf{Abs} = \gamma_1, \text{ then } \alpha = unit \\ \quad \text{if } \mathbf{Pre} = \gamma_1, \text{ then } \alpha = unit.\end{array}$$

It is important to note that this simplification process can be automated. We chose to show its intermediate steps, because it would otherwise be difficult to relate the final type scheme to the code for *tryKill'*. We now see that this type scheme does not require the privilege k to be enabled: our analysis was smart enough to prove that this code is safe.

The reader may wonder why we can't further simplify this type scheme by unifying α with *unit*, since both $\gamma_1 = \mathbf{Pre}$ and $\gamma_1 = \mathbf{Abs}$ imply $unit = \alpha$. This is because there remain other cases (namely $\gamma_1 = \bot$ and $\gamma_1 = \top$) where α is unconstrained; as a result, these conditional constraints do not logically imply $unit = \alpha$.

To fix this apparent problem, one possibility would be to remove $\bot$ and $\top$ from the model. In that case, replacing the two constraints above with $unit = \alpha$ would be a valid simplification. However, this change would effectively add *disjunction* to the constraint language—indeed, it would then be possible to encode the disjunction $C_1 \vee C_2$ as $\exists\gamma.(\text{if } \mathbf{Pre} = \gamma, \text{ then,} C_1 \wedge \text{if } \mathbf{Abs} = \gamma, \text{ then } C_2)$. (When $\bot$ *is* part of the model, such an encoding becomes impossible, because of the side condition $c \neq \bot$ in Figure 7.) We conjecture that the constraint satisfaction problem would then have exponential time complexity, while it currently has quasi-linear time complexity.

Another interesting possibility consists in giving a different interpretation to conditional constraints. Notice that we really wish to use conditional constraints in only a very limited way. Indeed, we want to allow the branches of a test construct to receive different types. But we do not wish for these types to differ in *arbitrary* ways; we only wish to allow their *security annotations* to differ. It is in fact possible to enforce such a restriction. Define $\approx$ as the binary relation which is uniformly true on $[\![Pres]\!]$. Extend it straightforwardly to $[\![k]\!]$ for every kind k. Then, redefine the interpretation of conditional constraints as follows:

$$\frac{\rho(\tau') \approx \rho(\tau'') \qquad c \leq \rho(\tau) \Rightarrow \rho \vdash \tau' \leq \tau''}{\rho \vdash \text{if } c \leq \tau \text{ then } \tau' \leq \tau''}.$$

This interpretation requires the types that appear in the conclusion of a conditional constraint (here, τ' and τ'') to be equal modulo security annotations. This allows the *structure* of types to be determined using rigid rules (which is desirable, because many programming errors are then detected earlier), while keeping the flexibility of conditional reasoning on security annotations. Under such an interpretation, the type of *tryKill'* may be simplified to

$$\forall \gamma_1 \beta_1.proc \rightarrow \{k : \gamma_1 \,;\, \beta_1\} \rightarrow unit,$$

as desired. From a practical point of view, this change in the interpretation of conditional constraints requires implementing two unification algorithms on top of one another—one for $=$ and one for $\approx$—which is straightforward. This variant of $\mathcal{S}_2^=$ may offer another good compromise between precision, efficiency, and readability of the types inferred.

7.3 Subtyping

All of the examples given so far can be given useful types in $\mathcal{S}_i^=$ for some $i \in \{1, 2\}$. In other words, these examples do not require subtyping. Nevertheless, there are a few cases where the extra precision afforded by subtyping becomes necessary.

Imagine we write a slightly modified version of the wrapper *enable*$_r$ presented in Section 7.1 as follows, where P is some arbitrary condition:

$$\mathit{maybeEnable}_r \triangleq \lambda f.p.\lambda x.p.\,\mathsf{if}\,P\,\mathsf{then}\,f\,x\,\mathsf{else}\,\mathsf{enable}\,r\,\mathsf{in}\,f\,x.$$

This wrapper may or may not enable the privilege r before calling f. In $\mathcal{S}_i^=$, its (most general) type is

$$\mathit{maybeEnable}_r : \forall \ldots (\alpha_1 \xrightarrow{\{r:\mathbf{Pre}\,;\,s:\gamma_1\,;\,\partial\mathbf{Abs}\}} \alpha_2) \xrightarrow{\{\beta_1\}} (\alpha_1 \xrightarrow{\{r:\mathbf{Pre}\,;\,s:\gamma_1\,;\,\beta_2\}} \alpha_2),$$

that is, exactly the same as that of *require*$_r$ in Section 7.1. In other words, the type system asserts, more conservatively than necessary, that *maybeEnable*$_r$ *requires* the privilege r. How was this conclusion drawn?

Because f is bound by λ and because HM(X) is restricted to Hindley–Milner polymorphism, the two uses of f must receive the same type, say $\alpha_1 \to \sigma \to \alpha_2$. In the second branch of the if statement, f is called with r enabled. Thus, σ must be of the form $\{r : \mathbf{Pre}\,;\ldots\}$. Since, in the first branch of the if statement, f is called within an unmodified security context, the type-checker concludes that the wrapped function also has $\{r : \mathbf{Pre}\,;\ldots\}$ as a security requirement.

The flaw is really in our use of equality constraints. Because f *may* be called with r enabled, they lead us to require $\sigma = \{r : \mathbf{Pre}\,;\ \ldots\}$, that is, to believe f *must* be called with r enabled. This extremely coarse approximation is good enough when f has polymorphic type, because we are then able to deal separately with each of its call sites. Here, however, polymorphism is inhibited, making the problem unbearable.

A standard solution is to move to a system where equality is replaced with subtyping, for example, $\mathcal{S}_1^\leq$. There, we obtain

$$\mathit{maybeEnable}_r : \forall \ldots (\alpha_1 \xrightarrow{\{r:\gamma\,;\,s:\gamma_1\,;\,\partial\mathbf{Abs}\}} \alpha_2) \xrightarrow{\{\beta_1\}} (\alpha_1 \xrightarrow{\{r:\gamma_2\,;\,s:\gamma_1\,;\,\beta_2\}} \alpha_2)$$
$$\text{where } \mathbf{Pre} \leq \gamma \wedge \gamma_2 \leq \gamma$$

This type scheme is much more permissive, because $\gamma_2 \leq \gamma \geq \mathbf{Pre}$ does not allow concluding $\gamma_2 \leq \mathbf{Pre}$ (as was the case when $\leq$ was interpreted by equality). Indeed, γ_2 may take the value $\mathbf{Abs}$, that is, the wrapped function may be called in a context where r is disabled. The constraint $\mathbf{Pre} \leq \gamma \wedge \gamma_2 \leq \gamma$ then requires $\top \leq \gamma$, that is, f must be able to accept either state of the privilege r.

Our experience seems to indicate that subtyping is useful only where polymorphism is inhibited, that is, when using higher-order functions. Java has no such construct. Java does have first-class objects, which contain methods; but it seems reasonable to require that methods be given explicit polymorphic types by the user as part of class declarations. Considering that subtyping has substantial cost in terms of readability and efficiency, it may then be interesting *not* to use it in a real-world system. However, more work is needed to confirm this conjecture.

7.4 Expressiveness versus Discipline

It is undecidable whether the execution of a given program eventually leads to a security failure. As a result, a safe type system equipped with decidable type inference must be conservative, that is, reject programs that in fact do not violate the security policy. For instance, in every $\mathcal{S}_i^{rel}$, a function f that requires privilege *r unless some condition P holds* receives a type that specifies that f requires *r always*, leading to a type error if f is invoked in a context where P holds and r is not available. Our types, viewed as a specification language for security policies, only have limited expressiveness. This is a curse and a blessing: while it prevents some legitimate programming idioms, it also forces programmers to stick to a reasonably straightforward programming style. The key, as always, is to strike a good compromise between expressiveness and discipline.

8. DISCUSSION

8.1 Extensions

There should be no particular difficulty in extending the ideas of this paper to more advanced language features such as exceptions, state, modules, and threads. In fact, for some of these features, we expect the type-the-translation approach to prove fruitful, by layering for example, an exceptions encoding on top of the security-passing encoding.

8.1.1 *Java.* The approach taken here has recently been shown to be extensible to the Java bytecode language [Higuchi and Ohori 2003], so the ideas here do transfer to the full JVM. But, modeling all the features of the Java security architecture is not possible statically. Java views privileges as first-class objects, making static typing problematic. In our model, privileges are identifiers, and expressions cannot compute privileges. It would be desirable to extend the static framework to at least handle first-class parameters of privileges, so for example, a Java `FilePermission`, which takes a parameter that is a specific file, could be modeled. The additional expressiveness of Java's implementation, including dynamic addition of permissions, and dynamically computable parameters to privileges (e.g., a `FilePermission` for the string `"/tmp/scratch"` that was created by appending strings `"/tmp"` and `"scratch"`), is very difficult to model statically.

From a manual inspection of the Sun JDK libraries, a substantial majority of the security code checks there can be statically typechecked. However, some of the uses are fundamentally dynamic. These include conditional checking of privileges where the condition is fundamentally dynamic and so cannot be captured statically. So, a purely static alternative would require some recoding of libraries, and a rethinking of where the security boundary is to be drawn. This is a deep problem, and it remains an open question whether the best completely static reworking of the architecture would be powerful enough to make the limitations of the static system acceptable.

An alternative approach is to accept that a completely static approach is not possible, and to use soft typing [Aiken et al. 1994; Wright and Cartwright 1997]. We discuss this further below, and also remark on extending our model to include exceptions.

With the addition of JAAS in the JDK 1.4 [Lai et al. 1999], the architecture also supports general authentication based on principals, not just codebases. The `doAs` instruction enables a block of code to be executed under a particular principal. We do not directly model JAAS, but for principals that are groups fixed in advance, the structure is static (and, desirably, more declarative than code that refers to specific users), and so our type system will be able to model it. So, principals `Alice` and `Bob` are not modeled statically, but fixed groups such as `DepartmentUser` and `GuestUser` which could contain `Alice` and `Bob`, respectively, could be declared and checked statically; only the membership of `Alice` in `DepartmentUser` would need to be checked dynamically.

8.1.2 *Soft Typing.* A soft typing system is a cross between a type system and a static optimizer. In our context, a soft typing system would allow some ill-typed check operations through, and mark them as requiring run-time checking. In principle, there is no problem with applying the soft typing approach in our framework, and allows our ideas to be applied directly to the JDK Security Architecture as now defined. Marked check operations would be treated much like test operations. The type system should provide a wealth of information to enable an efficient implementation of these tests. The constraint-based conditional type systems such as $\mathcal{S}_2^{\leq}$ are particularly appropriate for soft typing since the added expressiveness will allow more checks to be statically verified.

8.1.3 *Implementation of* test. Although our system statically checks whether all check operations will succeed at run-time, there is still a need to carry some privilege information at run-time to support test, which must *dynamically* branch on presence or absence of a privilege. We believe a static optimizer may be able to remove much of the run-time overhead of test. However, this implementation issue is beyond the scope of the current foundational study, and is a subject for future work.

8.1.4 *Exceptions.* In the simple language presented so far, security violations are fatal: they cause the program to halt. However, in Java, a security violation gives rise to an exception, which can be observed and dealt with by any (direct or indirect) caller. Thus, if our static security type system is to be viewed as realistic, it must be able to deal with exceptions.

For the sake of simplicity, we haven't included exceptions in our source language. However, it should be easy to add them as a second layer, with only little modification to our current proofs. In short, the idea is to introduce a new source language, featuring exceptions in addition to the security constructs, and to translate it down into an extension of λ_{sec} with sums. Indeed, it is a well-known fact that exceptions can be defined in terms of sums [Wadler 1985; Moggi 1989; Spivey 1990]. Any $\mathcal{S}_i^{rel}$ can then be lifted, through this new translation, up to the new source language. This construction shows that the "typing-by-encoding" approach can be used to account for exceptions. It also shows that several layers of encodings can be stacked on top of one another, making the proofs somewhat more modular.

This construction gives rise to type systems where function types carry not only a security precondition ς, but also an *effect* ϵ, which describes the exceptions that may be thrown when the function is invoked. This is a standard feature of type-based exception analyses [Guzmán and Suárez 1994; Aiken and Fähndrich 1997; Pessaux and Leroy 2000]. In Java terms, an effect is essentially a `throws` clause. However, a `throws` clause is constant, whereas, in our type systems, effects would be allowed to contain presence variables (i.e., type variables of kind *Pres*). These could be related, via constraints, to the function's security precondition ς, allowing properties such as "if privilege r is disabled, then this function may throw exception E" to be encoded in the types—and inferred by a type reconstruction algorithm.

A security check which throws an exception (instead of halting the program) upon failure can be defined, in the new source language, by combining test and throw. Thus, our new source language has both fatal and nonfatal forms of security checks. It is interesting to notice that each form has its advantages. Indeed, if a function yields a fatal error when the privilege r is disabled, then its type will quite concisely encode the sentence "r must be enabled", and the type-checker will automatically enforce this condition at every call site. If, on the other hand, the function throws an exception, then its type will more closely encode the sentence "if r is disabled, then the function may raise an exception", and the type-checker will not enforce any pre-condition when calling the function. (It is still possible to manually assert, using a type annotation, that a given call does not yield an exception, thus forcing r to be provably enabled at this call site.) The former may be preferred, because it is more legible, and because it documents the programmer's intent more precisely. On the other hand, the use of exceptions leads to a more modular programming style, because there is often no telling, at the time a particular piece of code is written, where and how security violations should be handled. We conclude that both forms of security checks may be of use in practice.

8.2 Related Work

8.2.1 *Other Analyses of Stack Inspection.* Banerjee and Naumann [2001] have developed an alternate proof of type safety for a programming language equipped with stack inspection. However, the denotational semantics of their language is in fact a security-passing style transform, which means that the correctness of this transform is taken for granted. Besson et al. [2001] and Jensen et al. [1999] define a whole-program static analysis based on model-checking temporal logic formulæ. Stack inspection is one (but not the only) application of their framework. A later paper [Besson et al. 2002] takes the analysis one step further by introducing a notion of *secure calling context*, symbolically represented as a temporal logic formula. However, the analysis is still not quite compositional, because the control flow graph of the entire program must be available. Bartoletti et al. [2001] propose a static analysis expressed as a fixpoint computation. Like Besson et al. [2001] and Jensen et al. [1991], they assume that programs are represented as graphs where only security checks and control flow are made explicit. Higuchi and Ohori [2003] impose a monomorphic type system with subtyping, reminiscent of the one developed in Skalka and Smith [2000], on a simple fragment on the JVM bytecode language. They note that, since check instructions can never fail in a well-typed program, they are no longer true operations: they are really only type annotations. For this reason, they suggest removing check from the language and replacing it with a more declarative type annotation mechanism. Allowing or requiring the programmer to assign a security-annotated type to each method provides one such mechanism. Koved et al. [2002] implement a flow-sensitive, context-sensitive analysis that determines, in a conservative fashion, which access rights are required by a piece of Java code. The analysis is precise—in particular, it keeps track of string constants, which are used in the creation of `Permission` objects, whereas we do not—and scales well. However, the paper does not contain enough detail for the reader to be able to implement the analysis. Koved et al.'s [2002] goals appear somewhat different from ours: They analyze unmodified Java programs, while our intention is to require programmers to annotate method headers with security requirements. While their approach requires less programmer effort, it is not clear whether it allows libraries to be analyzed in isolation, and whether it is able to provide an explanation for unexpected analysis results. We believe that a type-based approach, although more costly in terms of programmer effort, helps enforce a discipline that the programmer understands and controls. Naumovich [2002] describes a data flow analysis that ensures that certain privileges *must* be held in order to reach a certain program point. His purpose is dual to ours. Indeed, our type system is intended to ensure that no privilege checks may fail at runtime, but does not directly guarantee that the program is secure, while Naumovich's approach allows establishing security properties, but does not eliminate the possibility of a runtime failure. On a more theoretical level, Fournet and Gordon [2002] offer an in-depth study of the semantics of stack inspection; they establish equivalence laws which allow compilers to optimize away certain security-related instructions. Clements and Felleisen [2003] continue this line of work by developing an alternate but equivalent implementation of stack inspection that is shown to be tail-call optimizing.

8.2.2 *Other Approaches Based on a Translation.* Several researchers have proposed ways of defining efficient, provably correct compilation schemes for languages whose security policy is expressed by a *security automaton* [Erlingsson and Schneider 1999, 2000].

Walker [2000] defines a source language, equipped with such a security policy, then shows how to compile it into a dependently typed target language, whose type system, by encoding assertions about security states, guarantees that no run-time violations will occur. Walker first builds the target type system, then defines a *typed* translation. On the opposite, our approach consists in defining an *untyped* translation, whose output we feed through a type checker or inferencer for the target language. The composition yields a security-aware type checker or inferencer for the source language. In principle, our approach, which was developed with stack inspection in mind, is also applicable to security policies specified by security automata. Type inference for the target language, where the automaton's states and transition function are built-in constants, seems feasible: dedicated constraint language and constraint solver may be employed to allow statically reasoning about them. The untyped translation would thread the security automaton's state through every computation, making it an extra argument and an extra result of every function. Thus, in the derived type system, every function type would carry two annotations, a precondition and a postcondition, representing the automaton's state upon entry and upon exit. Again, these annotations could be type variables, related via constraints. In contrast with Walker's work, our approach makes security information visible in the type system of the source language: indeed, our aim is not only to gain performance by eliminating many dynamic checks, but also to define a programming discipline.

Thiemann's [2001] approach to security automata [Thiemann 2001] may be viewed as closely related to ours: he also starts with an untyped security-passing translation, whose output he then feeds through a standard program specializer. The composition automatically yields an optimizing translation.

8.2.3 *The Connection with Monads.* The encoding of exceptions alluded to in Section 8.1.4 is a monadic translation [Moggi 1989]. So is the security-passing style translation described in Section 4. In fact, an alternate semantics for our source language can be defined by successively layering [Filinski 1999] the following on top of a purely functional core:

(1) a failure monad, defined by $F\,\alpha = \alpha + 1$, representing the possibility of abrupt program termination;
(2) a security monad, defined by $S\,\alpha = \mathsf{PrivSet} \rightarrow \alpha$, where PrivSet represents privilege sets; enable, check and test can be defined as primitive operations at this level;
(3) (optionally) an exception monad, defined by $E\alpha = \alpha + \mathsf{Exc}$, where Exc represents exceptions.

Choosing such a semantics for our source language would remove the need to prove the translation sound, thus reducing even further the amount of work needed to prove the correctness of our type system. However, our choice of a concise operational semantics possibly brings us closer to the original description of Java stack inspection.

Monadic type systems have been used as a tool to isolate [Peyton Jones and Wadler 1993] or analyze [Wadler and Thiemann 2003] the use of impure language features in pure functional languages. Yet, as deplored in Wadler and Thiemann [2003], there is still "a need to create a new effect system for each new effect". In this light, our work may be viewed as a *systematic* construction of an "effect" type system adapted to our particular effectful programming language.

8.3 Final Remarks

From this methodological study emerge two type systems which improve on our previous work in type systems for access control. System $\mathcal{S}_1^=$ infers what appear to be very readable types, while remaining surprisingly expressive, and can be implemented very efficiently [Rémy 1992a]. System $\mathcal{S}_2^\leq$ is even more flexible and could form the basis of a soft typing system for the Java JDK platform. These systems were developed using a transformational technique and the system HM(X), which simplified proof effort and inspired design.

REFERENCES

AIKEN, A. S. AND FÄHNDRICH, M. 1997. Program analysis using mixed term and set constraints. In *Static Analysis Symposium (SAS)*. 114–126. URL: `http://www.cs.berkeley.edu/~aiken/publications/papers/sas97.ps`.

AIKEN, A. S., WIMMERS, E. L., AND LAKSHMAN, T. K. 1994. Soft typing with conditional types. In *ACM Symposium on Principles of Programming Languages (POPL)*. ACM, New York, 163–173. URL: `http://http.cs.berkeley.edu/~aiken/ftp/popl94.ps`.

BANERJEE, A. AND NAUMANN, D. A. 2001. A simple semantics and static analysis for Java security. Tech. Rep. 2001-1, Stevens Institute of Technology. June. URL: `http://guinness.cs.stevens-tech.edu/~naumann/publications/tr2001.ps`.

BARTOLETTI, M., DEGANO, P., AND FERRARI, G. 2001. Static analysis for stack inspection. In *International Workshop on Concurrency and Coordination*. Electronic Notes in Theoretical Computer Science, vol. 54. Elsevier Science, Amsterdam, The Netherlands.

BESSON, F., DE GRENIER DE LATOUR, T., AND JENSEN, T. 2002. Secure calling contexts for stack inspection. In *Proceedigs of the ACM International Conference on Principles and Practice of Declarative Programming (PPDP)*. ACM, New York, 76–87. URL: `http://www.irisa.fr/lande/jensen/ppdp02.pdf`.

BESSON, F., JENSEN, T. P., LE MÉTAYER, D., AND THORN, T. 2001. Model checking security properties of control flow graphs. *Journal of Computer Security 9*, 3, 217–250. URL: `http://www.irisa.fr/lande/jensen/jcs.pdf`.

CLEMENTS, J. AND FELLEISEN, M. 2003. A tail-recursive semantics for stack inspections. In *Proceedings of the European Symposium on Programming (ESOP)*. Lecture Notes in Computer Science, vol. 2618. Springer-Verlag, New York, 22–37. URL: `http://www.ccs.neu.edu/scheme/pubs/esop2003-cf.ps.gz`.

ERLINGSSON, Ú. AND SCHNEIDER, F. B. 1999. SASI enforcement of security policies: a retrospective. In *Proceedings of the New Security Paradigms Workshop*. 87–95. URL: `http://www.cs.cornell.edu/fbs/publications/sasiNSPW.ps`.

FILINSKI, A. 1999. Representing layered monads. In *Proceedings of the ACM Symposium on Principles of Programming Languages (POPL)*. ACM, New York, 175–188. URL: `http://www.brics.dk/~andrzej/papers/RLM.ps.gz`.

FOURNET, C. AND GORDON, A. D. 2002. Stack inspection: Theory and variants. In *Proceedings of the ACM Symposium on Principles of Programming Languages (POPL)*. ACM, New York, 307–318. URL: `http://research.microsoft.com/~fournet/papers/stack-inspection-theory-and-variants-popl-02.ps`.

GONG, L. 1998. Java security architecture (JDK1.2). URL: `http://java.sun.com/products/jdk/1.2/docs/guide/security/spec/security-spec.doc.html`.

GONG, L. AND SCHEMERS, R. 1998. Implementing protection domains in the Java Development Kit 1.2. In *Proceedings of the Internet Society Symposium on Network and Distributed System Security*. 125–134. URL: `http://java.sun.com/people/gong/papers/jdk12impl.ps.gz`.

Guzmán, J. C. and Suárez, A. 1994. An extended type system for exceptions. In *Proceedings of the ACM Workshop on ML and its Applications*. Number 2265 in INRIA Research Reports. INRIA, 127–135.

Higuchi, T. and Ohori, A. 2003. A static type system for JVM access control. In *Proceedings of the ACM International Conference on Functional Programming (ICFP)*. 227–237. URL: `http://www.jaist.ac.jp/~ohori/research/higuchiOhoriIcfp03.pdf`.

Jensen, T., Le Mètayer, D., and Thorn, T. 1999. Verifying security properties of control-flow graphs. In *Proceedings of the IEEE Symposium on Security and Privacy (S&P)*. IEEE Computer Society Press, Los Alamitos, Calif., 89–105. URL: `http://www.irisa.fr/lande/jensen/papers/SP99.ps`.

Koved, L., Pistoia, M., and Kershenbaum, A. 2002. Access rights analysis for Java. In *Proceedings of the ACM Conference on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA)*. ACM, New York, 359–372. URL: `http://www.research.ibm.com/javasec/OOPSLA2002preprint.pdf`.

Lai, C., Gong, L., Koved, L., Nadalin, A. J., and Schemers, R. 1999. User authentication and authorization in the Java platform. In *Proceedings of the Annual Computer Security Applications Conference*. 285–290. URL: `http://java.sun.com/people/gong/papers/jass.pdf`.

Moggi, E. 1989. Computational λ-calculus and monads. In *Proceedings of the IEEE Symposium on Logic in Computer Science (LICS)*. IEEE Computer Society Press, Los Alamitos, Calif., 14–23. URL: `http://www.disi.unige.it/person/MoggiE/ftp/lics89.ps.gz`.

Naumovich, G. 2002. A conservative algorithm for computing the flow of permissions in Java programs. In *Proceedings of the International Symposium on Software Testing and Analysis (ISSTA)*. 33–43. URL: `http://cis.poly.edu/gnaumovi/papers/permission-analysis.ps`.

Odersky, M., Sulzmann, M., and Wehr, M. 1999. Type inference with constrained types. *Theory Pract. Obj. Syst. 5*, 1, 35–55. URL: `http://www.comp.nus.edu.sg/~sulzmann/publications/tapos.ps`.

Pessaux, F. and Leroy, X. 2000. Type-based analysis of uncaught exceptions. *ACM Trans. Prog. Lang. Syst. 22*, 2, 340–377. URL: `http://pauillac.inria.fr/~xleroy/publi/exceptions-toplas.ps.gz`.

Peyton Jones, S. and Wadler, P. 1993. Imperative functional programming. In *Proceedings of the ACM Symposium on Principles of Programming Languages (POPL)*. ACM, New York. URL: `http://www.research.avayalabs.com/user/wadler/papers/imperative/imperative.ps.gz`.

Pottier, F. 2000. A versatile constraint-based type inference system. *Nordic J. Comput. 7*, 4 (Nov.), 312–347. URL: `http://pauillac.inria.fr/~fpottier/publis/fpottier-njc-2000.ps.gz`.

Pottier, F. 2003. A constraint-based presentation and generalization of rows. In *Proceedings of the IEEE Symposium on Logic in Computer Science (LICS)*. IEEE Computer Society Press, Los Alamitos, Calif., 331–340. URL: `http://pauillac.inria.fr/~fpottier/publis/fpottier-lics03.ps.gz`.

Pottier, F. and Conchon, S. 2000. Information flow inference for free. In *Proceedings of the ACM International Conference on Functional Programming (ICFP)*. ACM, New York, 46–57. URL: `http://pauillac.inria.fr/~fpottier/publis/fpottier-conchon-icfp00.ps.gz`.

Pottier, F., Skalka, C., and Smith, S. 2001. A systematic approach to static access control. In *Proceedings of the European Symposium on Programming (ESOP)*. Lecture Notes in Computer Science, vol. 2028. Springer-Verlag, New York, 30–45. URL: `http://pauillac.inria.fr/~fpottier/publis/fpottier-skalka-smith-esop01.ps.gz`.

Rémy, D. 1992a. Extending ML type system with a sorted equational theory. Tech. Rep. 1766, INRIA, Rocquencourt, BP 105, 78153 Le Chesnay Cedex, France. URL: `ftp://ftp.inria.fr/INRIA/Projects/cristal/Didier.Remy/eq-theory-on-types.ps.gz`.

Rémy, D. 1992b. Projective ML. In *Proceedins of the ACM Symposium on* Lisp *and Functional Programming (LFP)*. ACM, New York, 66–75. URL: `ftp://ftp.inria.fr/INRIA/Projects/cristal/Didier.Remy/lfp92.ps.gz`.

Rémy, D. 1994. Type inference for records in a natural extension of ML. In *Theoretical Aspects of Object-Oriented Programming. Types, Semantics and Language Design*, C. A. Gunter and J. C. Mitchell, Eds. MIT Press, Cambridge, Mass. URL: `ftp://ftp.inria.fr/INRIA/Projects/cristal/Didier.Remy/taoop1.ps.gz`.

SCHNEIDER, F. B. 2000. Enforceable security policies. *ACM Trans. Inf. Syst. Sec. 3*, 1 (Feb.), 1–50. URL: `http://www.cs.cornell.edu/fbs/publications/EnfSecPols.pdf`.

SIMONET, V. 2003. Type inference with structural subtyping: a faithful formalization of an efficient constraint solver. In *Proceedings of the Asian Symposium on Programming Languages and Systems*. URL: `http://cristal.inria.fr/~simonet/publis/simonet-structural-subtyping.ps.gz`.

SKALKA, C. 2002. Types for programming language-based security. Ph.D. dissertation. The Johns Hopkins University. URL: `http://www.cs.uvn.edu/~skalka/skalka-pubs/skalka-phd-thesis.ps`.

SKALKA, C. AND POTTIER, F. 2002. Syntactic type soundness for HM(X). In *Proceedings of the Workshop on Types in Programming (TIP)*. Electronic Notes in Theoretical Computer Science, vol. 75. URL: `http://pauillac.inria.fr/~fpottier/publis/skalka-fpottier-tip-02.ps.gz`.

SKALKA, C. AND SMITH, S. 2000. Static enforcement of security with types. In *Proceedings of the ACM International Conference on Functional Programming (ICFP)*. ACM, New York, 34–45. URL: `http://www.cs.uvm.edu/~skalka/skalka-pubs/skalka-smith-icfp00.ps`.

SPIVEY, M. 1990. A functional theory of exceptions. *Sci. Comput. Prog. 14*, 25–42.

SU, Z. AND AIKEN, A. 2001. Entailment with conditional equality constraints. In *Proceedings of the European Symposium on Programming (ESOP)*. Lecture Notes in Computer Science, vol. 2028. Springer-Verlag, 170–189. URL: `http://www.cs.ucdavis.edu/~su/publications/esop01.pdf`.

SULZMANN, M. 2000. A general framework for Hindley/Milner type systems with constraints. Ph.D. dissertation, Yale University, Department of Computer Science. URL: `http://www.comp.nus.edu.sg/~sulzmann/publications/diss.ps.gz`.

SULZMANN, M., MÜLLER, M., AND ZENGER, C. 1999. Hindley/Milner style type systems in constraint form. Research Report ACRC–99–009, University of South Australia, School of Computer and Information Science. July. URL: `http://www.ps.uni-sb.de/~mmueller/papers/hm-constraints.ps.gz`.

THIEMANN, P. 2001. Enforcing security properties using type specialization. In *Proceedings of the European Symposium on Programming (ESOP)*. Lecture Notes in Computer Science. Springer-Verlag, New York, URL: `http://www.informatik.uni-freiburg.de/~thiemann/papers/espps-het.ps.gz`.

WADLER, P. AND THIEMANN, P. 2003. The marriage of effects and monads. *ACM Trans. Comput. Logic 4*, 1 (Jan.), 1–32. URL: `http://www.research.avayalabs.com/user/wadler/papers/effectstocl/effectstocl.ps.gz`.

WADLER, P. L. 1985. How to replace failure by a list of successes. In *Proceedings of the Conference on Functional Programming Languages and Computer Architecture (FPCA)*. Lecture Notes in Computer Science, vol. 201. Springer-Verlag, New York, 113–128.

WALKER, D. 2000. A type system for expressive security policies. In *Proceedings of the ACM Symposium on Principles of Programming Languages (POPL)*. 254–267. URL: `http://www.cs.cornell.edu/home/walker/papers/sa-popl00_ps.gz`.

WALLACH, D. S. 1999. A new approach to mobile code security. Ph.D. dissertation, Princeton University. URL: `http://www.cs.princeton.edu/sip/pub/dwallach-dissertation.html`.

WALLACH, D. S., APPEL, A. W., AND FELTEN, E. W. 2000. SAFKASI: A security mechanism for language-based systems. *ACM Trans. Softw. Eng. Method. 9*, 4 (Oct.), 341–378. URL: `http://www.cs.rice.edu/~dwallach/pub/tosem2000.ps`.

WRIGHT, A. K. AND CARTWRIGHT, R. 1997. A practical soft type system for Scheme. *ACM Trans. Prog. Lang. Syst. 19*, 1 (Jan.), 87–152. URL: `http://doi.acm.org/10.1145/239912.239917`.

WRIGHT, A. K. AND FELLEISEN, M. 1994. A syntactic approach to type soundness. *Inf. Comput. 115*, 1 (Nov.), 38–94. URL: `http://www.cs.rice.edu/CS/PLT/Publications/Scheme/ic94-wf.ps.gz`.

Received December 2002; revised October 2003; accepted February 2004