A Type-Based Approach to Program Security*

Dennis Volpano[1] and Geoffrey Smith[2]

[1] Department of Computer Science, Naval Postgraduate School, Monterey, CA 93943, USA, email: volpano@cs.nps.navy.mil
[2] School of Computer Science, Florida International University, Miami, FL 33199, USA, email: smithg@cs.fiu.edu

Abstract. This paper presents a type system which guarantees that well-typed programs in a procedural programming language satisfy a *noninterference* security property. With all program inputs and outputs classified at various security levels, the property basically states that a program output, classified at some level, can never change as a result of modifying only inputs classified at higher levels. Intuitively, this means the program does not "leak" sensitive data. The property is similar to a notion introduced years ago by Goguen and Meseguer to model security in multi-level computer systems [7]. We also give an algorithm for inferring and simplifying *principal types*, which document the security requirements of programs.

1 Introduction

This paper presents a type system for a procedural language that guarantees that well-typed programs respect the security levels of the variables they manipulate. More precisely, it guarantees that well-typed programs are *noninterfering*, which basically means that high-security inputs cannot affect low-security outputs. Goguen and Meseguer introduced the idea of noninterference years ago as a notion of security for multi-level computing systems [7]; this papers applies the notion to programming languages. Our type soundness theorem is a proof that every well-typed program has the noninterference property. The proof depends on two lemmas that, interestingly, turn out to be typing analogs of two properties known for years within the security community as the simple security property and the confinement property (also known as the $*$-property). These are properties of the Bell and LaPadula model, developed in the early 70's as a model for multi-level security [4].

In an earlier work [17], we presented a type system to guarantee noninterference in a simple imperative language. In this work, we extend the analysis to a language with first-order procedures, which can be used polymorphically with respect to security classes. Also, we address the type inference problem here.

We begin with an overview of the type system. Then we formally present the system and prove its soundness relative to a standard natural semantics.

* Proceedings of TAPSOFT '97, Colloquium on Formal Approaches in Software Engineering, Lille France, 14-18 April, 1997.

In Section 6, we turn our attention to type inference and type simplification. Finally, we sketch some related efforts and some future research directions.

2 An Overview of the Type System

Noninterference was introduced as a model of security for multi-level computing systems [7]. The basic idea is that a system has users, some of whom supply high-level inputs and others who supply low-level inputs. Low-level users are only allowed to see low-level system outputs. (For the sake of simplifying the discussion, we shall consider only two security levels, *low* and *high*.) Such a system has the noninterference property if no matter how the high-level inputs change, the low-level system outputs remain the same.

The idea can also be applied to programming languages. Intuitively, the notion is that high-level program inputs can be altered without affecting any low-level outputs. As a simple example, consider a procedure with just two formal parameters x and y:

$$\mathbf{proc}\ P(\mathbf{inout}\ x : low,\ \mathbf{inout}\ y : high);$$

Here x and y are treated as variables with security levels low and high respectively. Suppose the calls $P(u : low, v : high)$ and $P(u : low, w : high)$ terminate with some final values for u, v, and w. The final values of v and w may differ. But if P is noninterfering, the final value of u will be the same in both cases. Our type system guarantees that well-typed programs are noninterfering.

2.1 Types

The types of the system are stratified into three levels. There are the τ types, which are the security levels, the π types, which are the types of expressions and commands, and the ρ types, which are the types of phrases. The security levels are assumed to be partially ordered by $\leq$. For example, one might have *low*, *high*, *trusted* and *untrusted* such that $low \leq high$ and $trusted \leq untrusted$. The relation $\leq$ is extended to a subtype relation $\subseteq$ over the phrase types.

Our phrase types are similar those of Forsythe [12], except that our command types are parameterized. A command type has the form $\tau\ cmd$; the intuition behind it is that a command c has this type only if every assignment in c is to a variable whose security level is τ or higher. So if a command has type $high\ cmd$, then it does not contain any assignments to low variables. Other phrase types are the types of variables, written $\tau\ var$, and the types of acceptors, written $\tau\ acc$. A variable of type $\tau\ var$ stores information whose security level is τ or lower. An acceptor is a write-only variable, used to type the **out** parameters of procedures. A variable is implicitly dereferenced, so there is a rule for converting $\tau\ var$ to τ. Likewise, there is a rule for converting a variable type to an acceptor type, which is necessary in the left sides of assignments and in procedure calls involving **out** parameters. The subtype relation is *contravariant* in both command and acceptor types.

2.2 The Core Language and Typing Rules

The typed language is a core imperative language with procedures; however, procedures are not first class values. Inspired by Denning's program certification rules [6], we have developed typing rules that ensure noninterference.

For instance, suppose that l and h are variables and that the identifier typing γ gives l type *low var* and gives h type *high var*. Then the assignment $l := h$ must be rejected, since a change in the initial value of h will affect the final value of l. This is what Denning termed an *explicit flow* from h to l. So we introduce the following typing rule:

$$\frac{\gamma \vdash e : \tau\ acc, \quad \gamma \vdash e' : \tau}{\gamma \vdash e := e' : \tau\ cmd}$$

This rule requires variables l and h in our example to agree on their security levels. Since they do not agree, even using subtyping, the assignment is rejected. On the other hand, $h := l$ is accepted. Since $low \leq high$, we can coerce the type of l from *low* to *high* to get agreement, allowing the assignment to be given type *high cmd*. Alternatively, we can coerce the type of h from *high acc* to *low acc* to give the assignment type *low cmd*.

It is worth pointing out that subtyping is neither covariant nor contravariant in variable types, because a variable is both an expression (which behaves covariantly) and an acceptor (which behaves contravariantly). Hence *low var* is unrelated to *high var*.

As another example, suppose we try to copy h to l indirectly as follows:

$$\begin{array}{l} \textbf{while}\ h > 0\ \textbf{do} \\ \quad l := l + 1; \\ \quad h := h - 1 \\ \textbf{od} \end{array}$$

Again the final value of l is affected by the initial value of h. This is what Denning termed an *implicit flow* from h to l. Thus, the typing rule for **while** insists that the guard and body of the loop be typed at the same security level:

$$\frac{\gamma \vdash e : \tau, \quad \gamma \vdash c : \tau\ cmd}{\gamma \vdash \textbf{while}\ e\ \textbf{do}\ c : \tau\ cmd}$$

Determining whether a given program is noninterfering is, of course, undecidable. As we shall see, our type system is a sound and decidable logic for reasoning about the noninterference of a program. Therefore, it is necessarily incomplete—some noninterfering programs are rejected by the type system.

2.3 Security Type Inference

Type inference in this setting attempts to prove that a program is noninterfering and produces a *principal type* that succinctly conveys how the program can be executed securely. A principal type is a constrained type scheme [13] with a *contraint set* of flat subtype inequalities among security levels. Consider, for instance, the following procedure that indirectly copies x to y:

$$\begin{array}{l}
\mathbf{proc}\ (\mathbf{in}\ x,\ \mathbf{out}\ y) \\
\quad \mathbf{letvar}\ a := x\ \mathbf{in} \\
\quad \mathbf{letvar}\ b := 0\ \mathbf{in} \\
\qquad \mathbf{while}\ a > 0\ \mathbf{do} \\
\qquad\quad b := b + 1; \\
\qquad\quad a := a - 1; \\
\qquad y := b
\end{array}$$

(The construct **letvar** $x := e$ **in** c allocates a local variable whose scope is c.) One principal type for this procedure is

$$\forall \alpha, \beta\ \mathbf{with}\ \alpha \leq \beta\ .\ \beta\ proc(\alpha, \beta\ acc)$$

where α and β are type variables such that α corresponds to the security level of x and β to the security level of y. A call to this procedure can be executed securely provided that the arguments have security levels that, when substituted for the bound variables of the type, satisfy the inequality. The call itself will have type $\beta\ cmd$, as conveyed by $\beta\ proc$. In this sense, the procedure is *polymorphic*. The above principal type can be simplified to $\forall \beta\ .\ \beta\ proc(\beta, \beta\ acc)$ due to subtyping of procedure types. As a practical matter, it is very important to simplify the inferred principal types by exploiting the antisymmetry of $\leq$ and the monotonicities of the type constructors. Type inference and simplification are discussed in detail in Section 6.

3 A Formal Treatment of the Type System

The syntax of the core imperative language is given below.

$$\begin{array}{lll}
(Phrase) & p ::= e \mid c & \\
(Expr) & e ::= x \mid n \mid l \mid e + e' \mid e - e' \mid e = e' \mid & \\
 & \qquad e < e' \mid \mathbf{proc}\ (\mathbf{in}\ x_1,\ \mathbf{inout}\ x_2,\ \mathbf{out}\ x_3)\ c & \\
(Comm) & c ::= e := e' \mid c\,;\, c' \mid e(e_1, e_2, e_3) \mid \mathbf{while}\ e\ \mathbf{do}\ c \mid & \\
 & \qquad \mathbf{if}\ e\ \mathbf{then}\ c\ \mathbf{else}\ c' \mid \mathbf{letvar}\ x := e\ \mathbf{in}\ c \mid & \\
 & \qquad \mathbf{letproc}\ x(\mathbf{in}\ x_1,\ \mathbf{inout}\ x_2,\ \mathbf{out}\ x_3)\ c\ \mathbf{in}\ c' &
\end{array}$$

Meta-variable x ranges over identifiers, n ranges over integer literals and l ranges over *locations*, which are used in our language for input and output: the initial values of any locations in a program represent inputs, and the final values of the locations represent outputs. (In addition, as will be seen in the natural semantics, evaluating a **letvar** causes a new location to be allocated, and later deallocated.) Also, we assume for simplicity that each procedure has exactly three parameters (one of each kind), and we use 0 for false and 1 for true. Finally, a phrase is *closed* if it has no free identifiers.

The types of the core language are stratified as follows:

$$\begin{aligned}
\tau &::= s \\
\pi &::= \tau \mid \tau\ proc(\tau_1,\ \tau_2\ var,\ \tau_3\ acc) \mid \tau\ cmd \\
\rho &::= \pi \mid \tau\ var \mid \tau\ acc
\end{aligned}$$

Meta-variable s ranges over a set of security levels, which is partially ordered by $\leq$. The rules of the type system are given in Figure 1. We omit typing rules for some compound expressions since they are similar to rule (SUM). Notice that rule (INT) allows an integer literal to be given *every* security level. Intuitively, a value is never intrinsically sensitive—it is sensitive only if it *comes from* a sensitive location. Note also that rule (LETPROC) allows procedures to be used polymorphically. The remaining rules of the type system constitute the subtyping logic and are given in Figure 2.

In the typing judgment $\lambda; \gamma \vdash p : \rho$, meta-variable γ ranges over identifier typings and λ over location typings. An *identifier typing* is a finite function mapping identifiers to types of the form τ, $\tau\ var$ or $\tau\ acc$; $\gamma(x)$ is the type assigned to x by γ, and $\gamma[x : \rho]$ is a modified identifier typing that assigns type ρ to x and assigns type $\gamma(x')$ to any identifier x' other than x. A *location typing* is a finite function mapping locations to τ types with similar notational conventions.

To facilitate the soundness proof, we introduce a *syntax-directed* set of typing rules. The rules of this system are just the rules of Figure 1 with rules (IDENT), (R-VAL), (ASSIGN), (IF), and (WHILE) replaced by their syntax-directed counterparts in Figure 3. The subtyping rules in Figure 2 are not included in the syntax-directed system. We write judgments in the syntax-directed system as $\lambda; \gamma \vdash_s p : \rho$. The benefit of the syntax-directed system is that the last rule used in the derivation of a typing $\lambda; \gamma \vdash_s p : \rho$ is uniquely determined by the form of p and of ρ. It is also helpful in determining where coercions are needed during type inference.

Next we establish that the syntax-directed system is actually equivalent to our original system with respect to the π types. First we need two lemmas:

Lemma 1. *If* $\lambda; \gamma[x : \rho'] \vdash_s p : \pi$ *and* $\vdash \rho \subseteq \rho'$, *then* $\lambda; \gamma[x : \rho] \vdash_s p : \pi$.

Lemma 2. *If* $\lambda; \gamma \vdash_s p : \pi$ *and* $\vdash \pi \subseteq \pi'$, *then* $\lambda; \gamma \vdash_s p : \pi'$.

Equivalence is now expressed by the following theorem:

Theorem 3. $\lambda; \gamma \vdash p : \pi$ *iff* $\lambda; \gamma \vdash_s p : \pi$.

From now on, we shall assume that all typing derivations are done in the syntax-directed type system, and therefore shall take $\vdash$ to mean $\vdash_s$.

4 A Natural Semantics

We give a natural semantics for closed phrases. A closed phrase is evaluated relative to a *memory* μ, which is a finite function from locations to integers. The

$$
\begin{array}{lll}
(\textsc{ident}) & \lambda;\gamma \vdash x : \tau & \gamma(x) = \tau \\[4pt]
(\textsc{var}) & \lambda;\gamma \vdash x : \tau\ \mathit{var} & \gamma(x) = \tau\ \mathit{var} \\[4pt]
(\textsc{acceptor}) & \lambda;\gamma \vdash x : \tau\ \mathit{acc} & \gamma(x) = \tau\ \mathit{acc} \\[4pt]
(\textsc{varloc}) & \lambda;\gamma \vdash l : \tau\ \mathit{var} & \lambda(l) = \tau \\[4pt]
(\textsc{int}) & \lambda;\gamma \vdash n : \tau & \\[4pt]
(\textsc{r-val}) & \dfrac{\lambda;\gamma \vdash e : \tau\ \mathit{var}}{\lambda;\gamma \vdash e : \tau} & \\[10pt]
(\textsc{l-val}) & \dfrac{\lambda;\gamma \vdash e : \tau\ \mathit{var}}{\lambda;\gamma \vdash e : \tau\ \mathit{acc}} & \\[10pt]
(\textsc{sum}) & \dfrac{\lambda;\gamma \vdash e : \tau,\ \ \lambda;\gamma \vdash e' : \tau}{\lambda;\gamma \vdash e + e' : \tau} & \\[10pt]
(\textsc{compose}) & \dfrac{\lambda;\gamma \vdash c : \tau\ \mathit{cmd},\ \ \lambda;\gamma \vdash c' : \tau\ \mathit{cmd}}{\lambda;\gamma \vdash c\,;\,c' : \tau\ \mathit{cmd}} & \\[10pt]
(\textsc{letvar}) & \dfrac{\lambda;\gamma \vdash e : \tau,\ \ \lambda;\gamma[x : \tau\ \mathit{var}] \vdash c : \tau'\ \mathit{cmd}}{\lambda;\gamma \vdash \textbf{letvar}\ x := e\ \textbf{in}\ c : \tau'\ \mathit{cmd}} & \\[10pt]
(\textsc{assign}) & \dfrac{\lambda;\gamma \vdash e : \tau\ \mathit{acc},\ \ \lambda;\gamma \vdash e' : \tau}{\lambda;\gamma \vdash e := e' : \tau\ \mathit{cmd}} & \\[10pt]
(\textsc{if}) & \dfrac{\lambda;\gamma \vdash e : \tau,\ \ \lambda;\gamma \vdash c : \tau\ \mathit{cmd},\ \ \lambda;\gamma \vdash c' : \tau\ \mathit{cmd},}{\lambda;\gamma \vdash \textbf{if}\ e\ \textbf{then}\ c\ \textbf{else}\ c' : \tau\ \mathit{cmd}} & \\[10pt]
(\textsc{while}) & \dfrac{\lambda;\gamma \vdash e : \tau,\ \ \lambda;\gamma \vdash c : \tau\ \mathit{cmd}}{\lambda;\gamma \vdash \textbf{while}\ e\ \textbf{do}\ c : \tau\ \mathit{cmd}} & \\[10pt]
(\textsc{procedure}) & \dfrac{\lambda;\gamma[x_1 : \tau_1, x_2 : \tau_2\ \mathit{var}, x_3 : \tau_3\ \mathit{acc}] \vdash c : \tau\ \mathit{cmd}}{\begin{array}{l}\lambda;\gamma \vdash \textbf{proc}\ (\textbf{in}\ x_1,\ \textbf{inout}\ x_2,\ \textbf{out}\ x_3)\ c : \\ \qquad\qquad \tau\ \mathit{proc}(\tau_1,\ \tau_2\ \mathit{var},\ \tau_3\ \mathit{acc})\end{array}} & \\[14pt]
(\textsc{apply}) & \dfrac{\begin{array}{l}\lambda;\gamma \vdash e : \tau\ \mathit{proc}(\tau_1,\ \tau_2\ \mathit{var},\ \tau_3\ \mathit{acc}), \\ \lambda;\gamma \vdash e_1 : \tau_1,\ \ \lambda;\gamma \vdash e_2 : \tau_2\ \mathit{var},\ \ \lambda;\gamma \vdash e_3 : \tau_3\ \mathit{acc}\end{array}}{\lambda;\gamma \vdash e(e_1, e_2, e_3) : \tau\ \mathit{cmd}} & \\[14pt]
(\textsc{letproc}) & \dfrac{\begin{array}{l}\lambda;\gamma \vdash \textbf{proc}\ (\textbf{in}\ x_1,\ \textbf{inout}\ x_2,\ \textbf{out}\ x_3)\ c : \pi, \\ \lambda;\gamma \vdash [\textbf{proc}\ (\textbf{in}\ x_1,\ \textbf{inout}\ x_2,\ \textbf{out}\ x_3)\ c/x]c' : \tau\ \mathit{cmd}\end{array}}{\lambda;\gamma \vdash \textbf{letproc}\ x(\textbf{in}\ x_1,\ \textbf{inout}\ x_2,\ \textbf{out}\ x_3)\ c\ \textbf{in}\ c' : \tau\ \mathit{cmd}} &
\end{array}
$$

Fig. 1. Rules of the Type System

contents of a location $l \in dom(\mu)$ is the integer $\mu(l)$, and we write $\mu[l := n]$ for the memory that assigns n to location l, and $\mu(l')$ to a location $l' \neq l$; thus $\mu[l := n]$ is an update of μ if $l \in dom(\mu)$ and an extension of μ if $l \notin dom(\mu)$.

Since expressions and commands are pure, our semantics uses $\mu \vdash e \Rightarrow n$ for the evaluation of an expression and $\mu \vdash c \Rightarrow \mu'$ for the evaluation of a command. Commands are nonexpansive in that $dom(\mu) = dom(\mu')$. We let $\mu - l$ stand for μ with location l removed from its domain.

$$\text{(BASE)}\quad \frac{\tau \le \tau'}{\vdash \tau \subseteq \tau'}$$

$$\text{(REFLEX)}\quad \vdash \rho \subseteq \rho$$

$$\text{(TRANS)}\quad \frac{\vdash \rho \subseteq \rho',\ \ \vdash \rho' \subseteq \rho''}{\vdash \rho \subseteq \rho''}$$

$$\text{(ACC}^{-}\text{)}\quad \frac{\vdash \tau \subseteq \tau'}{\vdash \tau'\ acc \subseteq \tau\ acc}$$

$$\text{(CMD}^{-}\text{)}\quad \frac{\vdash \tau \subseteq \tau'}{\vdash \tau'\ cmd \subseteq \tau\ cmd}$$

$$\text{(PROC)}\quad \frac{\vdash \tau_1' \subseteq \tau_1,\ \ \vdash \tau_3 \subseteq \tau_3',\ \ \vdash \tau' \subseteq \tau}{\vdash \tau\ proc(\tau_1,\ \tau_2\ var,\ \tau_3\ acc) \subseteq \tau'\ proc(\tau_1',\ \tau_2\ var,\ \tau_3'\ acc)}$$

$$\text{(SUBTYPE)}\quad \frac{\lambda; \gamma \vdash p : \rho,\ \ \vdash \rho \subseteq \rho'}{\lambda; \gamma \vdash p : \rho'}$$

Fig. 2. Subtyping rules

$$\text{(IDENT}'\text{)}\quad \frac{\gamma(x) = \tau,\ \ \tau \le \tau'}{\lambda; \gamma \vdash x : \tau'}$$

$$\text{(R-VAL}'\text{)}\quad \frac{\lambda; \gamma \vdash e : \tau\ var,\ \ \tau \le \tau'}{\lambda; \gamma \vdash e : \tau'}$$

$$\text{(ASSIGN}'\text{)}\quad \frac{\lambda; \gamma \vdash e : \tau\ acc,\ \ \lambda; \gamma \vdash e' : \tau,\ \ \tau' \le \tau}{\lambda; \gamma \vdash e := e' : \tau'\ cmd}$$

$$\text{(IF}'\text{)}\quad \frac{\lambda; \gamma \vdash e : \tau,\ \ \lambda; \gamma \vdash c : \tau\ cmd,\ \ \lambda; \gamma \vdash c' : \tau\ cmd,\ \ \tau' \le \tau}{\lambda; \gamma \vdash \textbf{if}\ e\ \textbf{then}\ c\ \textbf{else}\ c' : \tau'\ cmd}$$

$$\text{(WHILE}'\text{)}\quad \frac{\lambda; \gamma \vdash e : \tau,\ \ \lambda; \gamma \vdash c : \tau\ cmd,\ \ \tau' \le \tau}{\lambda; \gamma \vdash \textbf{while}\ e\ \textbf{do}\ c : \tau'\ cmd}$$

Fig. 3. Syntax-directed typing rules

The evaluation rules are given in Figure 4. We write $[e'/x]e$ to denote the capture-avoiding substitution of e' for all free occurrences of x in e. Note the use of substitution in rules (CALL), (BINDVAR) and (BINDPROC); this allows us to avoid environments and closures in the semantics.

5 Type Soundness as Noninterference

In this section, we establish the semantic soundness of our type system by proving a noninterference theorem. Before proving soundness, we require some lemmas that establish useful properties of the type system and semantics.

Lemma 4 (Expression Substitution). *If $\lambda; \gamma[x : \tau] \vdash p : \rho$, then $\lambda; \gamma \vdash [n/x]p : \rho$, and if $\lambda; \gamma \vdash l : \rho$ and $\lambda; \gamma[x : \rho] \vdash p : \rho'$, then $\lambda; \gamma \vdash [l/x]p : \rho'$.*

(VAL)	$\mu \vdash n \Rightarrow n$	
(CONTENTS)	$\mu \vdash l \Rightarrow \mu(l) \qquad l \in dom(\mu)$	

$$\text{(ADD)} \quad \frac{\mu \vdash e \Rightarrow n, \quad \mu \vdash e' \Rightarrow n'}{\mu \vdash e + e' \Rightarrow n + n'}$$

$$\text{(SEQUENCE)} \quad \frac{\mu \vdash c \Rightarrow \mu', \quad \mu' \vdash c' \Rightarrow \mu''}{\mu \vdash c\,;\, c' \Rightarrow \mu''}$$

$$\text{(BRANCH)} \quad \frac{\mu \vdash e \Rightarrow 1, \quad \mu \vdash c \Rightarrow \mu'}{\mu \vdash \mathbf{if}\ e\ \mathbf{then}\ c\ \mathbf{else}\ c' \Rightarrow \mu'}$$

$$\frac{\mu \vdash e \Rightarrow 0, \quad \mu \vdash c' \Rightarrow \mu'}{\mu \vdash \mathbf{if}\ e\ \mathbf{then}\ c\ \mathbf{else}\ c' \Rightarrow \mu'}$$

$$\text{(CALL)} \quad \frac{\mu \vdash e \Rightarrow n, \quad \mu \vdash [n, l, l'/x_1, x_2, x_3]c \Rightarrow \mu'}{\mu \vdash (\mathbf{proc}\ (\mathbf{in}\ x_1,\ \mathbf{inout}\ x_2,\ \mathbf{out}\ x_3)\ c)(e, l, l') \Rightarrow \mu'}$$

$$\text{(UPDATE)} \quad \frac{\mu \vdash e \Rightarrow n, \quad l \in dom(\mu)}{\mu \vdash l := e \Rightarrow \mu'[l := n]}$$

$$\text{(BINDVAR)} \quad \frac{\mu \vdash e \Rightarrow n, \quad l \text{ is the first location not in } dom(\mu), \quad \mu[l := n] \vdash [l/x]c \Rightarrow \mu'}{\mu \vdash \mathbf{letvar}\ x := e\ \mathbf{in}\ c \Rightarrow \mu' - l}$$

$$\text{(LOOP)} \quad \frac{\mu \vdash e \Rightarrow 0}{\mu \vdash \mathbf{while}\ e\ \mathbf{do}\ c \Rightarrow \mu}$$

$$\frac{\mu \vdash e \Rightarrow 1, \quad \mu \vdash c \Rightarrow \mu', \quad \mu' \vdash \mathbf{while}\ e\ \mathbf{do}\ c \Rightarrow \mu''}{\mu \vdash \mathbf{while}\ e\ \mathbf{do}\ c \Rightarrow \mu''}$$

$$\text{(BINDPROC)} \quad \frac{\mu \vdash [\mathbf{proc}\ (\mathbf{in}\ x_1,\ \mathbf{inout}\ x_2,\ \mathbf{out}\ x_3)\ c/x]c' \Rightarrow \mu'}{\mu \vdash \mathbf{letproc}\ x(\mathbf{in}\ x_1,\ \mathbf{inout}\ x_2,\ \mathbf{out}\ x_3)\ c\ \mathbf{in}\ c' \Rightarrow \mu'}$$

Fig. 4. The Evaluation Rules

Lemma 5 (Simple Security). *If $\lambda; \gamma \vdash e : \tau$, then for every l in e, $\lambda(l) \leq \tau$, and for every x free in e, $\gamma(x) \leq \tau$.*

Lemma 6 (Confinement). *If $\lambda \vdash c : \tau\ cmd$, $\mu \vdash c \Rightarrow \mu'$, $dom(\lambda) = dom(\mu)$, and l is a location assigned to in c, then $\lambda(l) \geq \tau$ or $\mu'(l) = \mu(l)$.*

Now we are ready to prove the soundness theorem.

Theorem 7 (Noninterference). *Suppose*

(a) $\lambda \vdash c : \pi$,
(b) $\mu \vdash c \Rightarrow \mu'$,
(c) $\nu \vdash c \Rightarrow \nu'$,
(d) $dom(\mu) = dom(\nu) = dom(\lambda)$, *and*
(e) $\nu(l) = \mu(l)$ *for all l such that* $\lambda(l) \leq \tau$.

Then $\nu'(l) = \mu'(l)$ for all l such that $\lambda(l) \leq \tau$.

In the absence of procedures, this theorem can be proved directly [17]. Here, however, we prove the Noninterference Theorem as a corollary to the following theorem, whose proof is omitted due to space restrictions.

Theorem 8. *Suppose*

(a) $\lambda; [x_1 : \tau_1, \ldots, x_k : \tau_k] \vdash c : \pi$,
(b) $\mu \vdash [n_1, \ldots, n_k / x_1, \ldots, x_k]c \Rightarrow \mu'$,
(c) $\nu \vdash [n'_1, \ldots, n'_k / x_1, \ldots, x_k]c \Rightarrow \nu'$,
(d) $dom(\mu) = dom(\nu) = dom(\lambda)$,
(e) $\nu(l) = \mu(l)$ *for all* l *such that* $\lambda(l) \leq \tau$, *and*
(f) $\not\vdash \tau_i \leq \tau$, *for all* i *such that* $1 \leq i \leq k$.

Then $\nu'(l) = \mu'(l)$ *for all* l *such that* $\lambda(l) \leq \tau$.

It is well known that polymorphic variables can easily break traditional forms of type soundness [16]. The same is true of a security type system. Giving a variable polymorphic type opens the door to "laundering". It would be possible to store high information and retrieve it as something low. But soundness can also break in more subtle ways due to mutable objects, like variables and first-class references, coupled with higher-order polymorphic procedures. It is interesting to note that if the core language were extended with these features, then existing techniques such as weak types [14] or limiting polymorphism to values [19] could be used to preserve soundness.

6 Type Inference

For the sake of describing type inference in this setting, we need to introduce *extended types* that can contain type variables (α, β,...) in place of security levels. We use metavariables $\widehat{\tau}$, $\widehat{\pi}$, and $\widehat{\rho}$ to range over extended types. Also, we use $\widehat{\gamma}$ to range over extended identifier typings that map identifiers to extended types; $FTV(\widehat{\gamma})$ gives the set of free type variables of $\widehat{\gamma}$.

A type inference algorithm W, defined by cases on the phrases of the language, is given in Figures 5 and 6. It takes as input a location typing λ, an extended identifier typing $\widehat{\gamma}$, a program phrase p, and a set V of type variables, which represents the set of "stale" type variables; this allows W to choose "fresh" type variables as necessary. If it succeeds, then it returns a set of flat subtype inequalities C, an extended type $\widehat{\pi}$, and an updated set V' of stale type variables. Note that the constraint $\widehat{\tau} = \widehat{\tau}'$ abbreviates the two inequalities $\widehat{\tau} \leq \widehat{\tau}'$ and $\widehat{\tau}' \leq \widehat{\tau}$.

We now establish the correctness of algorithm W. An *instantiation* I is a mapping from type variables to (ordinary) τ types. It can be applied, in the usual way, to extended types, to extended identifier typings, and to sets of inequalities among extended types.

Lemma 9. *If* $FTV(\widehat{\gamma}) \subseteq V$ *and* $(C, \widehat{\pi}, V') = W(\lambda, \widehat{\gamma}, p, V)$ *succeeds, then* V' *contains all type variables in* C, $\widehat{\pi}$, *and* V.

$W(\lambda, \widehat{\gamma}, p, V) =$ case p of

x : case $\widehat{\gamma}(x)$ of
$\quad \widehat{\tau} : (\{\widehat{\tau} \leq \alpha\}, \alpha, V \cup \{\alpha\}) \quad \alpha \notin V$
$\quad \widehat{\tau}\ var : (\{\widehat{\tau} \leq \alpha\}, \alpha, V \cup \{\alpha\}) \quad \alpha \notin V$
$\quad$ default : fail

$n : (\{\ \}, \alpha, V \cup \{\alpha\}) \quad \alpha \notin V$

$l : (\{\lambda(l) \leq \alpha\}, \alpha, V \cup \{\alpha\}) \quad \alpha \notin V$

$e_1 + e_2$:
$\quad$ let $(C_1, \widehat{\tau}_1, V') = W(\lambda, \widehat{\gamma}, e_1, V)$
$\quad$ let $(C_2, \widehat{\tau}_2, V'') = W(\lambda, \widehat{\gamma}, e_2, V')$
$\quad$ in $(C_1 \cup C_2 \cup \{\widehat{\tau}_1 = \widehat{\tau}_2\}, \widehat{\tau}_1, V'')$

proc (**in** x_1, **inout** x_2, **out** x_3) c :
$\quad$ let $(C, \widehat{\tau}\ cmd, V') = W(\lambda, \widehat{\gamma}[x_1 : \alpha, x_2 : \beta\ var, x_3 : \delta\ acc], c, V \cup \{\alpha, \beta, \delta\})$
$\quad$ in $(C, \widehat{\tau}\ proc(\alpha,\ \beta\ var,\ \delta\ acc), V') \quad \alpha, \beta$ and $\delta \notin V$

$c_1;\ c_2$: let $(C_1, \widehat{\tau}_1\ cmd, V') = W(\lambda, \widehat{\gamma}, c_1, V)$
$\quad$ let $(C_2, \widehat{\tau}_2\ cmd, V'') = W(\lambda, \widehat{\gamma}, c_2, V')$
$\quad$ in $(C_1 \cup C_2 \cup \{\widehat{\tau}_1 = \widehat{\tau}_2\}, \widehat{\tau}_1\ cmd, V'')$

if e **then** c_1 **else** c_2 :
$\quad$ let $(C, \widehat{\tau}, V') = W(\lambda, \widehat{\gamma}, e, V)$
$\quad$ let $(C_1, \widehat{\tau}_1\ cmd, V'') = W(\lambda, \widehat{\gamma}, c_1, V')$
$\quad$ let $(C_2, \widehat{\tau}_2\ cmd, V''') = W(\lambda, \widehat{\gamma}, c_2, V'')$
$\quad$ in $(C \cup C_1 \cup C_2 \cup \{\widehat{\tau} = \widehat{\tau}_1 = \widehat{\tau}_2, \alpha \leq \widehat{\tau}\}, \alpha\ cmd, V''' \cup \{\alpha\}) \quad \alpha \notin V'''$

while e **do** c :
$\quad$ let $(C, \widehat{\tau}, V') = W(\lambda, \widehat{\gamma}, e, V)$
$\quad$ let $(C', \widehat{\tau}'\ cmd, V'') = W(\lambda, \widehat{\gamma}, c, V')$
$\quad$ in $(C \cup C' \cup \{\widehat{\tau} = \widehat{\tau}', \alpha \leq \widehat{\tau}\}, \alpha\ cmd, V'' \cup \{\alpha\}) \quad \alpha \notin V''$

$e_1 := e_2$:
$\quad$ let $(C, \widehat{\tau}', V') = W(\lambda, \widehat{\gamma}, e_2, V)$
$\quad$ case e_1 of
$\quad\quad x$: if $\widehat{\gamma}(x) = \widehat{\tau}\ var$ or $\widehat{\gamma}(x) = \widehat{\tau}\ acc$ then
$\quad\quad\quad (C \cup \{\widehat{\tau} = \widehat{\tau}', \alpha \leq \widehat{\tau}'\}, \alpha\ cmd, V' \cup \{\alpha\}) \quad \alpha \notin V'$
$\quad\quad\quad$ else fail
$\quad\quad l : (C \cup \{\lambda(l) = \widehat{\tau}', \alpha \leq \widehat{\tau}'\}, \alpha\ cmd, V' \cup \{\alpha\}) \quad \alpha \notin V'$
$\quad\quad$ default : fail

letvar $x := e$ **in** c :
$\quad$ let $(C, \widehat{\tau}, V') = W(\lambda, \widehat{\gamma}, e, V)$
$\quad$ let $(C', \widehat{\tau}'\ cmd, V'') = W(\lambda, \widehat{\gamma}[x : \widehat{\tau}\ var], c, V')$
$\quad$ in $(C \cup C', \widehat{\tau}'\ cmd, V'')$

letproc x(**in** x_1, **inout** x_2, **out** x_3) c **in** c' :
$\quad$ let $(C, \widehat{\pi}, V') = W(\lambda, \widehat{\gamma},$ **proc** (**in** x_1, **inout** x_2, **out** x_3) $c, V)$
$\quad$ let $(C', \widehat{\tau}\ cmd, V'') = W(\lambda, \widehat{\gamma},$ [**proc** (**in** x_1, **inout** x_2, **out** x_3) $c/x]c', V')$
$\quad$ in $(C \cup C', \widehat{\tau}\ cmd, V'')$

Fig. 5. Algorithm W

$e(e_1, e_2, e_3)$:
  let $(C, \widehat{\tau}\ proc(\widehat{\tau}_1,\ \widehat{\tau}_2\ var,\ \widehat{\tau}_3\ acc), V') = W(\lambda, \widehat{\gamma}, e, V)$
  let $(C', \widehat{\tau}', V'') = W(\lambda, \widehat{\gamma}, e_1, V')$
  let C'' = case e_2 of
    x : if $\widehat{\gamma}(x) = \widehat{\tau}''\ var$ then $C \cup C' \cup \{\widehat{\tau}' = \widehat{\tau}_1, \widehat{\tau}'' = \widehat{\tau}_2\}$ else fail
    l : $C \cup C' \cup \{\widehat{\tau}' = \widehat{\tau}_1, \lambda(l) = \widehat{\tau}_2\}$
    default : fail
  in case e_3 of
    x : if $\widehat{\gamma}(x) = \widehat{\tau}''\ var$ or $\widehat{\gamma}(x) = \widehat{\tau}''\ acc$ then $(C'' \cup \{\widehat{\tau}'' = \widehat{\tau}_3\}, \widehat{\tau}\ cmd, V'')$
      else fail
    l : $(C'' \cup \{\lambda(l) = \widehat{\tau}_3\}, \widehat{\tau}\ cmd, V'')$
    default : fail

Fig. 6. Algorithm W, continued

Theorem 10 (Soundness). *Suppose $(C, \widehat{\pi}, V') = W(\lambda, \widehat{\gamma}, p, V)$ succeeds, and I is an instantiation such that $I(C)$ is true, and $I(\widehat{\gamma})$ and $I(\widehat{\pi})$ contain no type variables. Then $\lambda; I(\widehat{\gamma}) \vdash p : I(\widehat{\pi})$.*

Proof. By induction on the structure of p. We show the most interesting case; the other cases are similar and follow straightforwardly by induction.

Suppose $(C, \widehat{\tau}\ cmd, V'') = W(\lambda, \widehat{\gamma}, \mathbf{letvar}\ x := e\ \mathbf{in}\ c, V)$, $I(C)$ is true and $I(\widehat{\gamma})$ and $I(\widehat{\tau})$ are closed. From W, we have $C = C_1 \cup C_2$ where

$$(C_1, \widehat{\tau}', V') = W(\lambda, \widehat{\gamma}, e, V)$$

and

$$(C_2, \widehat{\tau}\ cmd, V'') = W(\lambda, \widehat{\gamma}[x : \widehat{\tau}'\ var], c, V') \ .$$

Let I' extend I so that $I'(\widehat{\tau}')$ is closed. Clearly, $I'(\widehat{\gamma}) = I(\widehat{\gamma})$ and $I'(\widehat{\tau}) = I(\widehat{\tau})$ since I' extends I and $I(\widehat{\gamma})$ and $I(\widehat{\tau})$ are closed. Further, $I'(C_1)$ is true since $I(C)$ is true. So by induction, $\lambda; I'(\widehat{\gamma}) \vdash e : I'(\widehat{\tau}')$, or $\lambda; I(\widehat{\gamma}) \vdash e : I'(\widehat{\tau}')$. Also, $I'(\widehat{\gamma}[x : \widehat{\tau}'\ var])$ is closed and $I'(C_2)$ is true, since $I(C)$ is true. So by a second use of induction, $\lambda; I'(\widehat{\gamma}[x : \widehat{\tau}'\ var]) \vdash c : I'(\widehat{\tau})\ cmd$. But $I'(\widehat{\gamma}[x : \widehat{\tau}'\ var]) = I'(\widehat{\gamma})[x : I'(\widehat{\tau}')\ var]$, so we have $\lambda; I(\widehat{\gamma})[x : I'(\widehat{\tau}')\ var] \vdash c : I(\widehat{\tau})\ cmd$. Therefore, by rule (LETVAR), $\lambda; I(\widehat{\gamma}) \vdash \mathbf{letvar}\ x := e\ \mathbf{in}\ c : I(\widehat{\tau})\ cmd$. □

Theorem 11 (Completeness). *Suppose $\lambda; I(\widehat{\gamma}) \vdash p : \pi$ and $FTV(\widehat{\gamma}) \subseteq V$. Then $(C, \widehat{\pi}, V') = W(\lambda, \widehat{\gamma}, p, V)$ succeeds and there exists an instantiation I' such that I' extends I, except on variables in $V' - V$, $I'(C)$ is true, and $I'(\widehat{\pi}) = \pi$. Moreover, if $W(\lambda, \widehat{\gamma}, p, V)$ does not succeed, then it halts with* **fail**.

Proof. By induction on the structure of p. We show two of the more interesting cases, **while** and **proc**; the others are similar.

Suppose $\lambda; I(\widehat{\gamma}) \vdash \mathbf{while}\ e\ \mathbf{do}\ c : \tau'\ cmd$ and $FTV(\widehat{\gamma}) \subseteq V$. Then, by rule (WHILE′), there is a type τ such that $\lambda; I(\widehat{\gamma}) \vdash e : \tau$, $\lambda; I(\widehat{\gamma}) \vdash c : \tau\ cmd$, and $\tau' \leq \tau$. So, by induction, $(C, \widehat{\pi}_1, V') = W(\lambda, \widehat{\gamma}, e, V)$ succeeds, $V \subseteq V'$, and there exists an instantiation I_1 such that I_1 extends I, except on variables in $V' - V$,

$I_1(C)$ is true and $I_1(\widehat{\pi}_1) = \tau$. So $\widehat{\pi}_1$ has the form $\widehat{\tau}_1$ and $I_1(\widehat{\tau}_1) = \tau$. And so $\widehat{\pi}_1$ does not cause the first pattern match to fail.

Now $FTV(\widehat{\gamma}) \subseteq V'$, and I_1 and I agree on all variables in $\widehat{\gamma}$ since no type variable in $V' - V$ is a member of $\widehat{\gamma}$. So $\lambda; I_1(\widehat{\gamma}) \vdash c : \tau\ cmd$. By induction again, $(C', \widehat{\pi}_2, V'') = W(\lambda, \widehat{\gamma}, c, V')$ succeeds, $V' \subseteq V''$, and there is an instantiation I_2 such that I_2 extends I_1, except on type variables in $V'' - V'$, $I_2(C')$ is true and $I_2(\widehat{\pi}_2) = \tau\ cmd$. So $\widehat{\pi}_2$ has the form $\widehat{\tau}_2\ cmd$ and $I_2(\widehat{\tau}_2) = \tau$. Thus, the second pattern match succeeds and so does $W(\lambda, \widehat{\gamma}, \textbf{while}\ e\ \textbf{do}\ c, V)$, returning

$$(C \cup C' \cup \{\widehat{\tau}_1 = \widehat{\tau}_2, \alpha \leq \widehat{\tau}_1\}, \alpha\ cmd, V'' \cup \{\alpha\})$$

where $\alpha \notin V''$. Now I_2 extends I, except on variables in $(V'' - V') \cup (V' - V)$ which is $V'' - V$ since $V \subseteq V' \subseteq V''$ by Lemma 9. Let $I' = I_2[\alpha := \tau']$. Then I' extends I except on variables in $(V'' - V) \cup \{\alpha\}$, or $(V'' \cup \{\alpha\}) - V$ since $\alpha \notin V$.

Finally, we establish that $I'(C \cup C' \cup \{\widehat{\tau}_1 = \widehat{\tau}_2, \alpha \leq \widehat{\tau}_1\})$ is true. By Lemma 9, V' contains all type variables in C and in $\widehat{\pi}_1$, so neither α nor any variable in $V'' - V'$ is a member of C or $\widehat{\pi}_1$. Thus I' and I_1 agree on all type variables in C and $\widehat{\pi}_1$. So $I'(C)$ is true and $I'(\widehat{\tau}_1) = \tau$. Likewise, by Lemma 9, V'' contains all type variables in C' and $\widehat{\pi}_2$. Since $\alpha \notin V''$, I' and I_2 agree on all type variables in C' and $\widehat{\pi}_2$. So $I'(C')$ is true and $I'(\widehat{\tau}_2) = \tau$. By the third hypothesis of rule (WHILE′), $I'(\alpha) \leq I'(\widehat{\tau}_1)$ and we're done.

Now suppose that

$$\lambda; I(\widehat{\gamma}) \vdash \textbf{proc}\ (\textbf{in}\ x_1,\ \textbf{inout}\ x_2,\ \textbf{out}\ x_3)\ c : \tau\ proc(\tau_1,\ \tau_2\ var,\ \tau_3\ acc)$$

and $FTV(\widehat{\gamma}) \subseteq V$. Then by rule (PROCEDURE), we have

$$\lambda; I(\widehat{\gamma})[x_1 : \tau_1, x_2 : \tau_2\ var, x_3 : \tau_3\ acc] \vdash c : \tau\ cmd\ \ .$$

Let $I_1 = I[\alpha := \tau_1, \beta := \tau_2, \delta := \tau_3]$ where $\alpha, \beta, \delta \notin V$. Since $FTV(\widehat{\gamma}) \subseteq V$, then α, β, and δ do not occur in $\widehat{\gamma}$. So $\lambda; I_1(\widehat{\gamma}[x_1 : \alpha, x_2 : \beta\ var, x_3 : \delta\ acc]) \vdash c : \tau\ cmd$. Hence, by induction, $W(\lambda, \widehat{\gamma}[x_1 : \alpha, x_2 : \beta\ var, x_3 : \delta\ acc], c, V \cup \{\alpha, \beta, \delta\})$ succeeds, returning $(C, \widehat{\pi}, V')$, $V \cup \{\alpha, \beta, \delta\} \subseteq V'$, and there exists an instantiation I' such that I' extends I_1, except on variables in $V' - (V \cup \{\alpha, \beta, \delta\})$, $I'(C)$ is true, and $I'(\widehat{\pi}) = \tau\ cmd$. So $\widehat{\pi}$ has the form $\widehat{\tau}\ cmd$ and $I'(\widehat{\tau}) = \tau$. Thus the pattern match succeeds and so does

$$W(\lambda; \widehat{\gamma}, \textbf{proc}\ (\textbf{in}\ x_1,\ \textbf{inout}\ x_2,\ \textbf{out}\ x_3)\ c, V)$$

returning $(C, \widehat{\tau}\ proc(\alpha,\ \beta\ var,\ \delta\ acc), V')$. Now I_1 extends I except on variables α, β and δ. So I' extends I except on variables in $(V' - (V \cup \{\alpha, \beta, \delta\})) \cup \{\alpha, \beta, \delta\}$ which is $V' - V$ since α, β, and δ are in V' but not V. □

It follows from these theorems that we can check whether p is typable with respect to λ and γ by first running $W(\lambda, \gamma, p, \emptyset)$, and, if it succeeds with $(C, \widehat{\pi}, V)$, then checking whether C is satisfiable with respect to the partial ordering of security levels. Checking the satisfiability of a flat set of subtyping inequalities with respect to a partial order has been studied previously [15, 18]. It is NP-complete, in general, but can sometimes be done efficiently, for example, if the partial order is a disjoint union of lattices.

6.1 Principal Types

In addition to checking typability, type inference gives us the ability to compute *principal types*, that document all possible types of a program. We use constrained quantification [13] for our principal types:

$$\sigma ::= \forall \bar{\alpha} \textbf{ with } C \,.\, \widehat{\pi}$$

In such a type scheme, the type variables $\bar{\alpha}$ can be instantiated only in ways that satisfy the subtype inequalities in C.

The *instances* of a type scheme are defined as follows:

Definition 12 (Instance). $\forall \bar{\alpha} \textbf{ with } C \,.\, \widehat{\pi} \succ \pi$ if there exists an instantiation I whose domain is $\bar{\alpha}$ such that $I(C)$ is true and $\vdash I(\widehat{\pi}) \subseteq \pi$. In this case we say that π is an *instance* of $\forall \bar{\alpha} \textbf{ with } C \,.\, \widehat{\pi}$.

Definition 13 (Principal Type). σ is a *principal type* for p with respect to λ and γ if for all π, $\lambda; \gamma \vdash p : \pi$ iff $\sigma \succ \pi$.

By the Soundness and Completeness theorems above, we can compute a principal type for p with respect to λ and γ by running $(C, \widehat{\pi}, V) = W(\lambda, \gamma, p, \emptyset)$, verifying that C is satisfiable, and forming the type scheme $\forall \bar{\alpha} \textbf{ with } C \,.\, \widehat{\pi}$, where $\bar{\alpha}$ contains all type variables free in C or $\widehat{\pi}$. (Note that the definition of the instance relation could in fact have required that $I(\widehat{\pi}) = \pi$; the weaker definition was adopted to allow for more type simplification, as we discuss below.)

Here is an example of type inference. Calling W on the procedure given in Section 2.3 produces the principal type

$$\begin{array}{l}\forall \alpha, \gamma, \nu, o, \epsilon, \iota, \zeta, \mu, \delta, \eta, \theta, \kappa, \lambda, \beta, \xi \textbf{ with} \\ \left\{ \begin{array}{l} \alpha \leq \gamma,\ \nu = o,\ \epsilon = \iota,\ \nu \leq \epsilon,\ \epsilon = \zeta,\ \gamma \leq \epsilon,\ \iota = \mu,\ \delta = \eta,\ \iota \leq \delta, \\ \eta = \theta,\ \delta \leq \eta,\ \gamma = \kappa,\ \mu \leq \gamma,\ \kappa = \lambda,\ \gamma \leq \kappa,\ \beta = \xi,\ o \leq \beta,\ \delta \leq \xi \end{array} \right\} \\ .\, \nu\ proc(\alpha,\ \beta\ acc) \end{array}$$

Such a complex principal type obviously cannot serve as useful documentation to a programmer. For this reason, it is necessary, as a practical matter, to simplify the principal types produced by W.

6.2 Type Simplification

There is a natural notion of equivalence on type schemes: two type schemes are *equivalent* iff they have the same set of instances. The idea of type simplification is to replace a type scheme with a simpler, yet equivalent, type scheme. The type simplifications considered in [13] can be applied directly here.

Often we can make deductions about how a type scheme $\forall \bar{\alpha} \textbf{ with } C \,.\, \widehat{\pi}$ can be instantiated. For instance, suppose that C contains the inequalities $\alpha \leq \beta$ as well as $\beta \leq \alpha$. Since $\leq$ is a partial order, any instantiation that satisfies C must instantiate α and β to the same type. Thus we can unify α and β. In general, we can collapse the strongly-connected components of C. Performing this simplification on the type scheme above yields the simpler principal type

$$\forall \alpha, o, \delta, \lambda, \xi \textbf{ with } \{\delta \leq \xi, o \leq \lambda, \lambda \leq \delta, \alpha \leq \lambda\} \,.\, o\ proc(\alpha, \xi\ acc)$$

We can further simplify type schemes by exploiting the monotonicities of types. For example, $o\ proc(\alpha, \xi\ acc)$ is antimonotonic in α; that is, boosting α produces a smaller type. Since the only constraint on α is that $\alpha \leq \lambda$, we can instantiate α to λ, yielding a simpler principal type. Performing such monotonicity-based instantiations repeatedly, we finally obtain the principal type

$$\forall \xi \,.\, \xi\ proc(\xi, \xi\ acc)$$

which has no constraints at all. With type simplification, principal types become useful documentation of the security requirements of programs.

7 Related Work and Future Directions

One of the earliest efforts in the area is Denning's lattice model of secure information flow [5, 6]. Denning extended the work of Bell and LaPadula [4] by giving a secure-flow certification algorithm for programs. This early work has been followed by a variety of efforts dealing with secure information flow [2, 8, 3, 10, 11, 17].

Some of these efforts [8, 10] have been aimed at proving the soundness of Denning's analysis. These efforts, however, prove soundness relative to an *instrumented semantics* whose validity is open to question. In contrast, we show the soundness of our analysis with respect to a standard natural semantics.

The work of Banâtre et al. [3] is similar in spirit to our work. They give a compile-time algorithm for detecting information flow in sequential programs, and they justify their algorithm in terms of a noninterference property. Their algorithm works by building a final accessibility graph indicating whether the contents of one variable at some point in the program can flow into an instance of a variable at some other point. The drawback here is that the number of vertices in the final accessibility graph is at least linear in the size of the program. This means that, unlike simplified principal types, final graphs cannot serve as practical program documentation.

Palsberg and Ørbæk [11] give a type system for trust analysis in the simply-typed λ calculus with a **trust** coercion. This (unsafe) coercion permits untrusted values to be explicitly coerced to trusted values. However, subject reduction is the only soundness property shown for their type system. It is unclear what one can say about the soundness of their system in terms of secure information flow. The **trust** coercion certainly rules out our noninterference theorem.

Another recent type-based approach is Abadi's work on a version of the pi calculus, called spi, extended to express cryptographic protocols [1]. Also related is Necula and Lee's recent work on proof-carrying code [9].

In the future, it would be desirable to extend the core language considered here with a number of important features, including concurrency, networking, and exception handling. The impact of such features on the noninterference property needs to be investigated.

References

1. Abadi, M., Secrecy by Typing in Cryptographic Protocols (Draft), unpublished manuscript, DEC Systems Research Center, December 1996.
2. Andrews, G. and Reitman, R., An Axiomatic Approach to Information Flow in Programs, *ACM Trans. on Programming Languages and Systems*, 2, 1, pp. 56–76, 1980.
3. Banâtre, J., Bryce, C., and Le Métayer, D., Compile-time Detection of Information Flow in Sequential Programs, *Proc. 3rd ESORICS*, LNCS 875, pp. 55–73, 1994.
4. Bell, D. and LaPadula, L., Secure Computer System: Mathematical Foundations and Model, MITRE Corp. Tech Report M74-244, 1973.
5. Denning, D., A Lattice Model of Secure Information Flow, *Comm of the ACM*, 19, 5, pp. 236–242, 1976.
6. Denning, D. and Denning, P., Certification of Programs for Secure Information Flow, *Comm of the ACM*, 20, 7, pp. 504–513, 1977.
7. Goguen, J. and Meseguer, J., Security Policies and Security Models, *Proc. 1982 IEEE Symposium on Security and Privacy*, pp. 11–20, 1982.
8. Mizuno, M. and Schmidt, D., A Security Flow Control Algorithm and its Denotational Semantics Correctness Proof, *Formal Aspects of Computing*, 4:6A, pp. 722–754, 1992.
9. Necula, G., Proof-Carrying Code, to appear in *Proc. 24th Symp. on Principles of Programming Languages*, January 1997.
10. Ørbæk, P., Can You Trust Your Data?, *Proc. 1995 TAPSOFT*, LNCS 915, pp. 575–589, 1995.
11. Palsberg, J. and Ørbæk, P., Trust in the λ-calculus, *Proc. 1995 Static Analysis Symposium*, LNCS 983, pp. 314–329, 1995.
12. Reynolds, J. Preliminary Design of the Programming Language Forsythe, Technical Report CMU-CS-88-159, Carnegie Mellon University, June 1988.
13. Smith, G., Principal Type Schemes for Functional Programs with Overloading and Subtyping, *Science of Computer Programming*, 23, pp. 197–226, 1994.
14. Smith, G. and Volpano, D., Polymorphic Typing of Variables and References, *ACM Trans. on Programming Languages and Systems*, 18, 3, pp. 254–267, 1996.
15. Tiuryn, J., Subtype Inequalities, *Proc. 1992 IEEE Symp. on Logic in Computer Science*, pp. 308–315, 1992.
16. Tofte, M., Type Inference for Polymorphic References, *Information and Computation*, 89, pp. 1–34, 1990.
17. Volpano, D., Smith, G. and Irvine, C., A Sound Type System for Secure Flow Analysis, *J. Computer Security*, 4, 3, pp. 1–21, 1996.
18. Wand, M. and O'Keefe, P., On the Complexity of Type Inference with Coercion, *Proc. ACM Conf. on Functional Programming Languages and Computer Architecture*, pp. 293–298, 1989.
19. Wright, A., Simple Imperative Polymorphism, *Journal of Lisp and Symbolic Computing*, 8, 4, pp. 343–356, 1995.

This article was processed using the LaTeX macro package with LLNCS style