# Typed Language Support for Proof-Carrying Authorization
(Lecture Notes)

Deepak Garg

November 28, 2009

## 1 Introduction

Previous lectures in this class covered logic-based representation of access policies as well as their enforcement with proof-carrying authorization (PCA). The idea behind PCA is two-fold. First, access policies and credentials of principals are expressed as logical formulas, which are published in digitally signed certificates. Second, the reference monitor protecting sensitive resources allows an access only if the access request is accompanied by enough certificates to authorize the access and a logical proof which shows why the access is authorized by the formulas in those certificates.

PCA is a general architecture which may be used with any logic for representing policies so long as the logic has a proof theory. In class we have seen a logic ICL [4, 5] and a logic-based language SecPAL [2], both of which can represent access policies, and both of which support inference. Either of these could be used as a basis for enforcement with PCA.

This lecture looks at the problem of designing a programming language whose resource access interfaces (file system calls, network calls, etc.) are protected using proof-carrying authorization. As an illustrative example, consider the interface function `fsRead` that allows a program to read the contents of a file – $\texttt{fsRead}(f)$ returns the contents of file $f$. If the interface is proof-carrying, then the function would also take as argument a proof $M$ that authorizes the access; its call would take the form $\texttt{fsRead}(f)[M]$. (As a convention, we write proofs passed as arguments in square brackets $[\cdot]$ to distinguish them from regular arguments.) In this case, $M$ may establish a formula of the form $\mathsf{fs}\ \mathsf{says}\ \texttt{Read}(A, f)$, where $A$ is the user on behalf of whom the program is running, and $\mathsf{fs}$ denotes the administrator of the file system. The file system would verify the proof $M$, and if it does indeed establish $\mathsf{fs}\ \mathsf{says}\ \texttt{Read}(A, f)$, it would return the contents of the file, else it would return an error.

In this lecture we will discuss a simple language, IPCA, that supports proof-carrying interfaces such as `fsRead`. Relevant questions for the design of such a language are:

1. Where are certificates (policy rules and credentials) represented? For simplicity, in this lecture we will assume that the relevant certificates, denoted $\Gamma$, are available in a central store to the user, the type-checker, the language's run time, and the system interface that checks proofs. This eliminates the need to pass certificates in interface

calls. A language that passes such proofs is not much harder to design, but may make the presentation unclear.

2. What is the syntax of proofs $M$? So far in class, we have seen proofs written as trees of inference rules, but of course, for a language we need a textual representation. To this end, we will look at a representation of proofs called *proof-terms* which are textual and compact, and represent $M$ in that form (see Section 2).

3. Which logic or policy language is used to express policies and inference? As should become clear, the design of IPCA is largely independent of any particular logic or policy language. For illustration we will use SecPAL, but the metatheory of IPCA relies only on a few properties of the policy formalism, all of which should hold for most authorization logics and policy languages.

4. What guarantees does IPCA provide? As part of type checking, proofs embedded in IPCA programs are also checked. The type checker will accept a program only if it is certain that the program will not pass an incorrect proof to an interface when it executes. This is a *type-safety* property, which we prove formally as a theorem.

Note that it is also possible to provide proof-carrying interfaces through libraries in a language like C or Java. In that case proofs would be represented as data structures, and checked by library code implementing the interfaces. However, a standard type checker cannot check logical correctness of data structures representing proofs. Consequently, the type-safety (point 4 above) may not hold in such a setting.

In the next section we discuss proof-terms (point 2 above) using SecPAL as an illustration. In further sections, we discuss the syntax, semantics, type system, and type-safety of IPCA. Like Mini-C from an earlier lecture, IPCA is a bare-bones language with minimal constructs that are just sufficient to illustrate the relevant points and work out an example. Other more sophisticated languages which build on this idea are PCML$_5$ [1], Aura [6], and PCAL [3]. Of these, IPCA is most closely related to PCAL. It should be noted that although these languages have more constructs and features than IPCA, they are still experimental. At present, there are no widely used languages that support proof-carrying interfaces.

## 2 Proof-Terms for SecPAL

Recall from class that SecPAL is a language for representing and drawing authorizations from access control policies.[1] The inference rules of SecPAL establish judgments of the form $\Gamma \vdash a$, where $a$ is an assertion (a fact stated by a principal), and $\Gamma$ is a set of assumed assertions (called the hypotheses or policy) which are obtained from certificates. The judgment $\Gamma \vdash a$ means that from the assumptions $\Gamma$, assertion $a$ can be deduced.

[1]What we considered in class, and also what we consider in this document, is only a fragment of the full language SecPAL. For a description of the full language, see [2]. SecPAL also contains its own representation of proofs. Proof-terms presented here are different from that.

| | | | |
|---|---|---|---|
| Logic Variables | $X, Y, O, F$ | | |
| Constants | $c$ | ::= | $\mathsf{Alice} \mid \mathsf{Bob} \mid \ldots \mid$ "a.txt" $\mid \ldots$ |
| Terms | $A, t$ | ::= | $X \mid c$ |
| Predicates | $p$ | ::= | $\mathtt{Read} \mid \mathtt{Write} \mid \mathtt{Owns} \mid \ldots$ |
| Facts | $f$ | ::= | $p(t_1, \ldots, t_n) \mid A\ \mathsf{cansay}\ f$ |
| Statements | $s$ | ::= | $f\ \mathsf{if}\ f_1, \ldots, f_n$ |
| Assertions | $a$ | ::= | $A\ \mathsf{says}\ s$ |
| Hypotheses/Policies | $\Gamma$ | ::= | $a_1, \ldots, a_n$ |

As a convention, terms representing principals are written $A$ or $B$. All other uppercase letters are logic variables. Constants and predicates in SecPAL are open-ended; here we have chosen representatives based on access control in file systems, which will be our running example. SecPAL contains two inference rules that are shown below. In the first rule, $\theta$ is a substitution that maps all logic variables in $f$ $\mathsf{if}$ $f_1, \ldots, f_n$ to ground terms. $f\theta$ denotes the application of $\theta$ to $f$.

$$\frac{(A\ \mathsf{says}\ (f\ \mathsf{if}\ f_1, \ldots, f_n)) \in \Gamma \qquad \Gamma \vdash A\ \mathsf{says}\ f_i\theta}{\Gamma \vdash A\ \mathsf{says}\ f\theta}$$

$$\frac{\Gamma \vdash A\ \mathsf{says}\ (B\ \mathsf{cansay}\ f) \qquad \Gamma \vdash B\ \mathsf{says}\ f}{\Gamma \vdash A\ \mathsf{says}\ f}$$

SecPAL proofs are trees of inference rules (as in logic or type systems). A simplified, but equivalent representation of proofs is as proof-terms. A proof-term, $M$, is obtained by flattening a proof tree with a pre-order traversal. In order to do define proof-terms, we must first associate unique names with all assumptions in $\Gamma$. Accordingly, we revise the syntax of $\Gamma$:

$$\text{Hypotheses/Policies} \quad \Gamma \quad ::= \quad \alpha_1 : a_1, \ldots, \alpha_n : a_n$$

$\alpha_1, \ldots, \alpha_n$ are names for assumptions. It is implicitly assumed that these names are distinct. Proof-terms have the following syntax.

$$\text{Proof-terms} \quad M \quad ::= \quad \mathtt{pf\_app}\ \alpha\ \theta\ (M_1, \ldots, M_n) \mid \mathtt{pf\_cansay}\ M_1\ M_2$$

Next, we revise the judgment $\Gamma \vdash a$ to include a proof-term. We write $\Gamma \vdash M : a$ to mean that $M$ is a proof-term which establishes that $a$ follows from the assumptions $\Gamma$. As in the syntax above, $M$ may contain names $\alpha$; those then refer to identically named assumptions in $\Gamma$. The rules for establishing $\Gamma \vdash M : a$ are obtained by modifying SecPAL's inference rules as follows.

$$\frac{(\alpha : A\ \mathsf{says}\ (f\ \mathsf{if}\ f_1, \ldots, f_n)) \in \Gamma \qquad \Gamma \vdash M_i : A\ \mathsf{says}\ f_i\theta}{\Gamma \vdash \mathtt{pf\_app}\ \alpha\ \theta\ (M_1, \ldots, M_n) : A\ \mathsf{says}\ f\theta}$$

$$\frac{\Gamma \vdash M_1 : A\ \mathsf{says}\ (B\ \mathsf{cansay}\ f) \qquad \Gamma \vdash M_2 : B\ \mathsf{says}\ f}{\Gamma \vdash \mathtt{pf\_cansay}\ M_1\ M_2 : A\ \mathsf{says}\ f}$$

Proof-terms are a complete representation of proofs in the following sense.

**Lemma 2.1.** *There is a derivation of $\Gamma \vdash a$ in SecPAL (without proof-terms) if and only if for every $\Gamma'$ obtained by giving unique names to assumptions in $\Gamma$, there is a proof term $M$ such that there is a derivation of $\Gamma' \vdash M : a$ in SecPAL with proof-terms.*

Another important property of proof-terms is that they can be verified easily. There is an algorithm, linear in the size of $M$, which given $\Gamma$, $M$, and $a$, checks whether $\Gamma \vdash M : a$. Owing to Lemma 2.1 and this easy verifiability, proof-terms are an appropriate representation of proofs in many settings, including languages like IPCA. It should also be noted that proof-terms are not unique to SecPAL; similar proof-terms exist for all inference systems including all logics. In the rest of this document, we always represent proofs through proof-terms and often use the word "proof" to mean a "proof-term".

**Example 2.2.** Consider the following policy $\Gamma$ with five assertions. The predicate $\mathtt{Read}(A, F)$ means that principal $A$ can read file $F$, whereas $\mathtt{Owns}(A, F)$ means that principal $A$ owns file $F$. (Following standard SecPAL convention, all logic variables are implicitly universally quantified.)

$$\begin{aligned}
\Gamma \quad = \quad & \alpha_1 : \mathtt{fs}\ \mathsf{says}\ ((O\ \mathsf{cansay}\ \mathtt{Read}(X, F))\ \mathsf{if}\ \mathtt{Owns}(O, F)),\\
& \alpha_2 : \mathtt{fs}\ \mathsf{says}\ \mathtt{Owns}(\mathsf{Alice}, \text{"a.txt"}),\\
& \alpha_3 : \mathtt{fs}\ \mathsf{says}\ \mathtt{Owns}(\mathsf{Alice}, \text{"b.txt"}),\\
& \alpha_4 : \mathtt{Alice}\ \mathsf{says}\ \mathtt{Read}(\mathsf{Bob}, \text{"a.txt"}),\\
& \alpha_5 : \mathtt{Alice}\ \mathsf{says}\ \mathtt{Read}(\mathsf{Bob}, \text{"b.txt"})
\end{aligned}$$

The assertion $\alpha_1$ is a statement by the file system administrator $\mathsf{fs}$. It means that the principal $O$ who owns a file $F$ may state that another principal $X$ may read $F$. $\alpha_2$ and $\alpha_3$ mean that $\mathsf{Alice}$ owns files "a.txt" and "b.txt" respectively. $\alpha_4$ and $\alpha_5$ are statements made by $\mathsf{Alice}$. By way of these assertions $\mathsf{Alice}$ conveys her intention to allow $\mathsf{Bob}$ to read files "a.txt" and "b.txt" respectively.

We may intuitively expect that $\alpha_1$, $\alpha_2$, and $\alpha_4$ would entail $\mathsf{fs}\ \mathsf{says}\ \mathtt{Read}(\mathsf{Bob}, \text{"a.txt"})$ and similarly that $\alpha_1$, $\alpha_3$, and $\alpha_5$ would entail $\mathsf{fs}\ \mathsf{says}\ \mathtt{Read}(\mathsf{Bob}, \text{"b.txt"})$. Both these intuitions are correct – there are proof-terms $M_a$ and $M_b$ such that $\Gamma \vdash M_a : \mathsf{fs}\ \mathsf{says}\ \mathtt{Read}(\mathsf{Bob}, \text{"a.txt"})$ and $\Gamma \vdash M_b : \mathsf{fs}\ \mathsf{says}\ \mathtt{Read}(\mathsf{Bob}, \text{"b.txt"})$. These proof-terms are shown below.

$$\begin{aligned}
M_a \quad &= \quad \mathtt{pf\_cansay}\ (\mathtt{pf\_app}\ \alpha_1\ [\mathsf{Alice}/O, \mathsf{Bob}/X, \text{"a.txt"}/F]\ (\alpha_2))\ \alpha_4\\
M_b \quad &= \quad \mathtt{pf\_cansay}\ (\mathtt{pf\_app}\ \alpha_1\ [\mathsf{Alice}/O, \mathsf{Bob}/X, \text{"b.txt"}/F]\ (\alpha_3))\ \alpha_5
\end{aligned}$$

In summary, it is possible to represent proofs using a compact notation called proof-terms. In order to understand IPCA, it is not important to understand how proof-terms represent proofs, but only that they do and that proof-terms can be checked easily.

| | | | |
|---|---|---|---|
| Types | $T$ | $::=$ | $\mathsf{int} \mid \mathsf{bool} \mid \mathsf{principal} \mid \mathsf{string}$ |
| Variables | $x, y$ | | |
| Values | $v$ | $::=$ | $\mathsf{true} \mid \mathsf{false} \mid -1 \mid 0 \mid 1 \mid \mathsf{Alice} \mid \mathsf{Bob} \mid \text{“a.txt”} \mid \ldots$ |
| Expressions | $e$ | $::=$ | $x \mid v \mid e_1 + e_2 \mid e_1 * e_2 \mid e_1 <= e_2 \mid e_1 == e_2 \mid \mathtt{concat}(e_1, e_2) \mid \ldots$ |
| Restricted expressions | $w$ | $::=$ | $x \mid v$ |
| Commands | $c$ | $::=$ | $\mathsf{noop} \mid x = e \mid c_1; c_2 \mid \mathsf{if}\ e\ \mathsf{then}\ c_1\ \mathsf{else}\ c_2 \mid \mathsf{while}\ e\ \mathsf{do}\ c \mid$ |
| | | | $x = \mathsf{syscall}\ i\ (w_1, \ldots, w_n)\ [M] \mid \mathsf{locate}\ a\ (\alpha \Rightarrow c_1 \mid c_2)$ |
| Declarations | $\Delta$ | $::=$ | $x_1 : T_1; \ldots; x_n : T_n$ |
| Programs | $P$ | $::=$ | $\mathsf{decl}\ \Delta\ \mathsf{begin}\ c\ \mathsf{end}$ |
| Policies | $\Gamma$ | $::=$ | $\alpha_1 : a_1, \ldots, \alpha_n : a_n$ |
| Stores | $\sigma$ | $::=$ | $x_1 \mapsto v_1, \ldots, x_n \mapsto v_n$ |

**Figure 1:** Syntax of IPCA

## 3 IPCA: Syntax

IPCA extends Mini-C with interfaces that are protected with proof-carrying authorization. To the syntax of the language we add a new command $x = \mathsf{syscall}\ i\ (w_1, \ldots, w_n)\ [M]$, which calls the interface function $i$ (such as `fsRead`) with arguments $w_1, \ldots, w_n$ and an authorization proof $M$. The proof $M$ is actually a proof-term of SecPAL, as described in Section 2 although we could have used another policy formalism as well. In the operational semantics of $x = \mathsf{syscall}\ i\ (w_1, \ldots, w_n)\ [M]$, the proof $M$ is checked before any results are returned from the interface call. The letter $w$ denotes a restricted class of expressions that may either be values or variables, but not operations like $e_1 + e_2$.

Another command added to the language is $\mathsf{locate}\ a\ (\alpha \Rightarrow c_1 \mid c_2)$. This command directs the language run time to locate a certificate which establishes assertion $a$. If such a certificate is found, its name is bound to the variable $\alpha$ and command $c_1$ is executed. If such a certificate is not found, then command $c_2$ is executed. The scope of $\alpha$ includes $c_1$ but not $c_2$. $\mathsf{locate}\ a\ (\alpha \Rightarrow c_1 \mid c_2)$ can be used for finding certificates to construct proofs.

Besides these two new commands, we also extend Mini-C's types to include $\mathsf{principal}$ and $\mathsf{string}$ in order to write a meaningful example. Values (constants) of the two types are also added. Addition of the new types and values poses no new difficulty in the type system or semantics. The syntax of IPCA is shown in Figure 1. The syntax of assertions $a$ and proof-terms $M$ is drawn from SecPAL (Section 2). An important point: we assume that SecPAL constants coincide with IPCA values, and allow language variables $x, y$ to appear as terms in SecPAL. These variables have no specific meaning in SecPAL. In particular, unlike logic variables, they cannot be instantiated or substituted in proofs. Instead they are treated like constants (in logic jargon, such constants are called parameters).

An illustrative IPCA program is shown in Figure 2. This program concatenates the

```
 1. decl
 2.    c: int; out: string; z: string; x: string;
 3. begin
 4.    c = 0;
 5.    while (c < 2) do
 6.       if (c == 0) then x = "a.txt" else x = "b.txt";
 7.       c = c + 1;
 8.       locate (fs says ((O cansay Read(X,F)) if Owns(O,F)))
 9.         (alphaA =>
10.          locate (fs says Owns(Alice, x))
11.             (alphaB =>
12.              locate (Alice says Read(Bob, x))
13.                 (alphaC =>
14.                  z = syscall fsRead (x) [M]; //See note below
15.                  out = concat(out,z)
16.             | noop)
17.          | noop)
18.       | noop)
20. end
```

**Note.** `M` is an abbreviation for:

```
pf_cansay (pf_app alphaA [Alice/O,Bob/X,x/F] (alphaB)) alphaC
```

**Figure 2:** IPCA program to concatenate “a.txt” and “b.txt”

contents of the two files “a.txt” and “b.txt” and stores the result in the variable `out`. The main body of the program is a loop that runs twice. In the first iteration it assigns “a.txt” to the variable `x` (Line 6), constructs a proof that allows the program to read the file named `x` (Lines 8–13), then reads the file (Line 14), and concatenates its contents to `out` (Line 15). The loop then repeats in a similar way, except that “b.txt” is assigned to `x`.

The part of the program most relevant to our discussion is construction of the proof-term `M` and the syscall to `fsRead`. For proof construction, the program assumes that the policy in effect is the one from Example 2.2 and that the program will execute on behalf of Bob. Using three locate statements in each iteration (Lines 8, 10, 12) the program expects to find three relevant certificates to authorize the call to `fsRead`. The first locate statement tries to find a certificate for the assertion `(fs says ((O cansay Read(X,F)) if Owns(O,F)))`. This corresponds to assumption $\alpha_1$ from Example 2.2. Consequently, if the policy from Example 2.2 is in effect, `alphaA` will be bound to $\alpha_1$ at run time. If, on the other hand, such a certificate does not exist, the program will jump to noop at the end and loop.

The second locate statement (Line 10) tries to find a certificate which contains the assertion `(fs says Owns(Alice, x))`. During the first and second iterations, `x` will be "a.txt" and "b.txt", so this call will try to locate certificates $\alpha_2$ and $\alpha_3$ of Example 2.2 respectively in the two iterations. Accordingly, `alphaB` will be bound to $\alpha_2$ in the loop's first iteration and to $\alpha_3$ in the second iteration. In a similar manner, `alphaC` will be bound to $\alpha_4$ in the loop's first iteration and to $\alpha_5$ in the second iteration. The proof-term `M` is similar to $M_a$ and $M_b$ from Example 2.2 except that it abstracts away the difference between "a.txt" and "b.txt" using the variable `x`. Indeed at run time, the proof-term will be exactly $M_a$ in the first iteration, and $M_b$ in the second iteration. Since we already know that those proof-terms are correct, the proofs passed to the interface `fsRead` will successfully check, and the system calls will succeed.

It is important to observe that although we have argued informally that *this* program will pass correct proofs to its interface call at run time, we want to *check correctness of proofs during type-checking.* During type-checking, the two bindings "a.txt" and "b.txt" for the variable `x` will not be available (we certainly don't want the compiler to execute the program to determine what `x` will be). Consequently, the type-checker must check proofs *parametrically* in the program variables. What this means is that the proofs constructed in the program must be correct for all values of the program variables, not just those that will be attained at run time. Indeed in this case, if program control ever reaches the command `z = syscall fsRead (x) [M]`, then `alphaA`, `alphaB`, and `alphaC` must bind to certificates that establish `(fs says ((O cansay Read(X,F)) if Owns(O,F)))`, `(fs says Owns(Alice, x))`, and `(Alice says Read(Bob, x))` respectively. It is easy to show that from these assumptions, `M` (as constructed in the program) is a proof of `(fs says Read(Bob, x))`. This entire deduction is independent of the actual value(s) of the variable `x`, which is why the type checker can check the proof without executing the program to determine `x`.

## 4 Type System

The type system for IPCA is similar to that for Mini-C, except that it checks proofs passed to system interfaces. To check proofs, we must know at the least the assertions that must be proved at each system interface. We do this using interface specifications, which we assume are provided and fixed throughout. An interface specification has the form $i : (x_1 : T_1, \ldots, x_n : T_n)[a] \rightarrow T_r$, which means that interface $i$ expects $n$ arguments of types $T_1, \ldots, T_n$ as well as a proof of the assertion $a$, and returns a value of type $T_r$. The argument names $x_1, \ldots, x_n$ can be used to refer to the arguments in the assertion $a$. (In type-system jargon, $a$ is said to be *dependent* on the variables $x_1, \ldots, x_n$.) In addition, $a$ may contain the special constant $\mu$ to refer to the principal executing the program. As an example, the specification of the interface `fsRead` would be.

$$\texttt{fsRead} : (x : \mathsf{string})[\mathsf{fs\ says\ Read}(\mu, x)] \rightarrow \mathsf{string}$$

The type system of IPCA uses two judgments: $\Delta \vdash e : T$, which means that expression $e$ has type $T$, and $\Delta; \Gamma \vdash^A c$, which means that command $c$ is well-formed for execution by principal $A$. The principal $A$ must be known during type-checking to find the binding for

$\boxed{\text{Types for commands}}$

$$\frac{}{\Delta;\Gamma \vdash^A \mathsf{noop}} \qquad \frac{(x:T)\in\Delta \qquad \Delta\vdash e:T}{\Delta;\Gamma\vdash^A x = e} \qquad \frac{\Delta;\Gamma\vdash^A c_1 \qquad \Delta;\Gamma\backslash\mathtt{modifies}(c_1)\vdash^A c_2}{\Delta;\Gamma\vdash^A c_1;c_2}$$

$$\frac{\Delta\vdash e:\mathsf{bool} \qquad \Delta;\Gamma\vdash^A c_1 \qquad \Delta;\Gamma\vdash^A c_2}{\Delta;\Gamma\vdash^A \mathsf{if}\ e\ \mathsf{then}\ c_1\ \mathsf{else}\ c_2} \qquad \frac{\Delta\vdash e:\mathsf{bool} \qquad \Delta;\Gamma\backslash\mathtt{modifies}(c)\vdash^A c}{\Delta;\Gamma\vdash^A \mathsf{while}\ e\ \mathsf{do}\ c}$$

$$\frac{\begin{array}{c} i:(x_1:T_1,\ldots,x_n:T_n)[a]\rightarrow T_r \\ \Delta\vdash w_j:T_j \qquad (x:T_r)\in\Delta \qquad \Gamma\vdash M: a\{w_1/x_1\}\ldots\{w_n/x_n\}\{A/\mu\}\end{array}}{\Delta;\Gamma\vdash^A x = \mathsf{syscall}\ i\ (w_1,\ldots,w_n)\ [M]}$$

$$\frac{\Delta;\Gamma,\alpha:a\vdash^A c_1 \qquad \Delta;\Gamma\vdash^A c_2}{\Delta;\Gamma\vdash^A \mathsf{locate}\ a\ (\alpha\Rightarrow c_1 \mid c_2)}$$

**Figure 3:** Type system of IPCA (commands)

$\mu$ in interface specifications. Our type-safety theorem assumes explicitly that the principal $A$ used for type-checking matches the principal actually executing the program (else a proof passed to an interface at run time may authorize an incorrect principal, causing the program to fail).[2] $\Gamma$ is a list of certificates that would necessarily have been collected when $c$ executes. For example, when type-checking the program of Figure 2, $\Gamma$ would be empty when we start type-checking, and `alphaA: (fs says ((O cansay Read(X,F)) if Owns(O,F)))` when checking the command starting at Line 10.

Rules of the judgment $\Delta\vdash e:T$ are similar to those of Mini-C, except that we account for the new types and constants. These are presented in Appendix A. The rules of the judgment $\Delta;\Gamma\vdash^A c$ are shown in Figure 3. Rules for commands included in Mini-C are similar to corresponding rules of Mini-C. A subtle difference arises in the rules for $c_1;c_2$ and $\mathsf{while}\ e\ \mathsf{do}\ c$. In the case of the former, it is possible that $c_1$ may modify some variables, and accordingly, assertions in $\Gamma$ that contain such variables must be removed from $\Gamma$ when checking $c_2$. We write $\Gamma\backslash\mathtt{modifies}(c_1)$ to denote the subset of $\Gamma$ containing only those assertions that do not mention any $x$ for which $x=\cdot$ appears in $c_1$. Similarly, since the loop $\mathsf{while}\ e\ \mathsf{do}\ c$ unrolls to $(c;\mathsf{while}\ e\ \mathsf{do}\ c)$, when type-checking $c$ in the body of $\mathsf{while}\ e\ \mathsf{do}\ c$, we must remove from $\Gamma$ any assertions that contain variables potentially modified by $c$.

The important (and new) rules here are those for $\mathsf{syscall}$ and $\mathsf{locate}$. In the former, the objective is to type-check $x = \mathsf{syscall}\ i\ (w_1,\ldots,w_n)\ [M]$. First, the specification of the interface $i$ must be looked up. If it has the form $i:(x_1:T_1,\ldots,x_n:T_n)[a]\rightarrow T_r$, then we must check that the inputs and outputs have the right type. So each $w_j$ must have type $T_j$

[2]In IPCA, the principal on behalf of whom the program executes cannot change during program execution. However, a command to change this principal can be added with a moderate amount of effort.

(second premise), and the variable $x$ must have the return type $T_r$ (third premise). Finally, we check that the proof-term $M$ establishes the assertion $a$ with appropriate substitutions: $w_j$ for $x_j$ and $A$ for $\mu$ (fourth premise). Observe that $\Gamma$ and $M$ may contain program variables. The proof checking, which is done using SecPAL's rules from Section 2 here, must treat these variables as constants (parameters), so that the proof, if correct, holds no matter what values the variables take at run time.

The rule for type-checking $\mathsf{locate}\ a\ (\alpha \Rightarrow c_1 \mid c_2)$ is straightforward: $c_1$ is checked with the additional assertion assumption $\alpha : a$, and $c_2$ is checked without this assumption.

**Exercise 4.1.** Suppose that the program of Figure 2 parses as $\mathsf{decl}\ \Delta\ \mathsf{begin}\ c\ \mathsf{end}$. Show that $\Delta; \Gamma \vdash^{\mathsf{Bob}} c$ for any $\Gamma$, assuming the specification $\mathtt{fsRead} : (x : \mathsf{string})[\mathsf{fs\ says\ Read}(\mu, x)] \rightarrow \mathsf{string}$.

## 5 Operational Semantics

We formalize the operational semantics of IPCA using two judgments: (a) $\sigma \rhd e \hookrightarrow e'$ which means that under the store $\sigma$, expression $e$ simplifies to $e'$, and (b) $\sigma ; c \xrightarrow{\Gamma_0, A} \sigma' ; c'$, which means that the command $c$ in store $\sigma$ reduces to $c'$ and updates the store to $\sigma'$. $\Gamma_0$ is the set of (named) policy certificates that are available in the system. Note that this set is not the same as the set $\Gamma$ available during type-checking. The latter is based on $\mathsf{locate}$ statements in the program. $\Gamma_0$, on the other hand, represents the actual certificates available when the program executes. It is independent of any particular program, and must not contain any program variables. $A$ is the principal on behalf of whom the program is executing.

Rules of the judgment $\sigma \rhd e \hookrightarrow e'$ are similar to those of Mini-C. They are listed in Appendix B. Rules of the judgment $\sigma ; c \xrightarrow{\Gamma_0, A} \sigma' ; c'$ are shown in Figure 4. The important rules here are those for $\mathsf{syscall}$ and $\mathsf{locate}$. In the former, we wish to evaluate $x = \mathsf{syscall}\ i\ (w_1, \ldots, w_n)\ [M]$. First, each of $w_1, \ldots, w_n$ are evaluated. From the syntax, each $w_j$ is either a value or a program variable. In the former case it is already evaluated; in the latter case, it is evaluated by looking it up in $\sigma$. We write $w_j\sigma$ for the result of evaluating $w_j$. Next, we look up the specification of the interface $i$ (first premise). If the specification is $i : (x_1 : T_1, \ldots, x_n : T_n)[a] \rightarrow T_r$, then the proof $M$ must be checked against the assertion $a\{w_1\sigma/x_1\} \ldots \{w_n\sigma/x_n\}\{A/\mu\}$ in the available assertions $\Gamma_0$ (second premise). If this succeeds then the interface call is evaluated. We model this evaluation with the judgment $i(w_1\sigma, \ldots, w_n\sigma) \Downarrow v$, which means that interface $i$ when called with arguments $w_1\sigma, \ldots, w_n\sigma$ returns result $v$ (third premise). The overall effect of the command is to update the value of $x$ in the store to $v$.

There are two rules for evaluating $\mathsf{locate}\ a\ (\alpha \Rightarrow c_1 \mid c_2)$. Since the objective of the command is to find an assertion $a$ in the set of available assertions, we first evaluate $a$ fully by substituting program variables in it using their values in $\sigma$. The result is written $a\sigma$. Next, $a\sigma$ is looked up in $\Gamma_0$. If $\alpha' : a\sigma$ exists in $\Gamma_0$ for some $\alpha'$, then we evaluate $c_1$ with $\alpha'$ substituted for $\alpha$ (first rule), else we evaluate $c_2$ (second rule).

$$\boxed{\sigma\,;c \xrightarrow{\Gamma_0,A} \sigma'\,;c'}$$

$$\frac{\sigma \rhd e \hookrightarrow^* v}{\sigma\,;(x=e) \xrightarrow{\Gamma_0,A} \sigma[x\mapsto v]\,;\ \mathsf{noop}} \qquad \frac{\sigma\,;c_1 \xrightarrow{\Gamma_0,A} \sigma'\,;c_1'}{\sigma\,;(c_1;c_2) \xrightarrow{\Gamma_0,A} \sigma'\,;(c_1';c_2)} \qquad \frac{}{\sigma\,;(\mathsf{noop};c_2) \xrightarrow{\Gamma_0,A} \sigma\,;c_2}$$

$$\frac{\sigma \rhd e \hookrightarrow^* \mathsf{true}}{\sigma\,;(\mathsf{if}\ e\ \mathsf{then}\ c_1\ \mathsf{else}\ c_2) \xrightarrow{\Gamma_0,A} \sigma\,;c_1} \qquad \frac{\sigma \rhd e \hookrightarrow^* \mathsf{false}}{\sigma\,;(\mathsf{if}\ e\ \mathsf{then}\ c_1\ \mathsf{else}\ c_2) \xrightarrow{\Gamma_0,A} \sigma\,;c_2}$$

$$\frac{\sigma \rhd e \hookrightarrow^* \mathsf{true}}{\sigma\,;(\mathsf{while}\ e\ \mathsf{do}\ c) \xrightarrow{\Gamma_0,A} \sigma\,;(c;(\mathsf{while}\ e\ \mathsf{do}\ c))} \qquad \frac{\sigma \rhd e \hookrightarrow^* \mathsf{false}}{\sigma\,;(\mathsf{while}\ e\ \mathsf{do}\ c) \xrightarrow{\Gamma_0,A} \sigma\,;\ \mathsf{noop}}$$

$$\frac{\begin{array}{c} i:(x_1:T_1,\ldots,x_n:T_n)[a]\to T_r \\ \Gamma_0 \vdash M\sigma : a\{w_1\sigma/x_1\}\ldots\{w_n\sigma/x_n\}\{A/\mu\} \qquad i(w_1\sigma,\ldots,w_n\sigma)\Downarrow v \end{array}}{\sigma\,;x = \mathsf{syscall}\ i\ (w_1,\ldots,w_n)\ [M] \xrightarrow{\Gamma_0,A} \sigma[x\mapsto v]\,;c'}$$

$$\frac{\alpha' : a\sigma \in \Gamma_0}{\sigma\,;\ \mathsf{locate}\ a\ (\alpha \Rightarrow c_1 \mid c_2) \xrightarrow{\Gamma_0,A} \sigma; c_1\{\alpha'/\alpha\}} \qquad \frac{\text{There is no } \alpha' \text{ such that } \alpha' : a\sigma \in \Gamma_0}{\sigma\,;\ \mathsf{locate}\ a\ (\alpha \Rightarrow c_1 \mid c_2) \xrightarrow{\Gamma_0,A} \sigma; c_2}$$

**Figure 4:** Operational semantics of IPCA (commands)

## 6 Type-Safety

We state the type-safety theorem of IPCA, and list assumptions used in its proof. We say that $\Delta \vdash \sigma$ if for each $(x:T)\in\Delta$, there is a $v$ such that $\cdot \vdash v : \sigma$ and $\sigma(x)=v$. Type-safety is formalized in the following theorem.

**Theorem 6.1** (Type-safety)**.** *Suppose $\Delta;\cdot \vdash^A c$ and $\Delta \vdash \sigma$. If $\sigma\,;c \xrightarrow{\Gamma_0,A}{}^{*} \sigma'\,;c'$, then either $c' =$noop or there are $\sigma''$ and $c''$ such that $\sigma'\,;c' \xrightarrow{\Gamma_0,A} \sigma''\,;c''$.*

The theorem says that if $c$ is well-typed for execution as principal $A$, then when the program is run as principal $A$ (with any set of assertions $\Gamma_0$), the program cannot get stuck, i.e. it will either evaluate to noop or keep evaluating for ever. In particular, this means that the program will never pass an incorrect proof to an interface proof, because if a program did indeed try to pass an incorrect proof, the operational semantics would block the program and cause it to not evaluate any further.

The proof of type-safety is quite involved, and we omit it. The theorem relies on the following substitution property of the inference system of the framework used to represent policies. This property is quite general and holds for most well-designed inference systems including that of SecPAL (Section 2), and as a result IPCA can be used with many policy formalisms without losing type-safety.

> (Substitution assumption) If $\Gamma \vdash M : a$ and $\rho$ is a partial substitution for program variables, then $\Gamma\rho \vdash M\rho : a\rho$.

Finally, the type-safety theorem assumes that interface calls return values that conform to their specifications.

> (Interface assumption) If $i : (x_1 : T_1, \ldots, x_n : T_n)[a] \rightarrow T_r$ and $\cdot \vdash v_j : T_j$, then there is a $v$ such that $i(v_1, \ldots, v_n) \Downarrow v$ and $\cdot \vdash v : T_r$.

## References

[1] Kumar Avijit, Anupam Datta, and Robert Harper. Distributed programming with distributed authorization. In *Proceedings of the Fifth ACM Workshop on Types in Language Design and Implementation (TLDI)*, 2009. To appear.

[2] Moritz Y. Becker, Cédric Fournet, and Andrew D. Gordon. Design and semantics of a decentralized authorization language. In *20th IEEE Computer Security Foundations Symposium*, pages 3–15, 2007.

[3] Avik Chaudhuri and Deepak Garg. PCAL: Language support for proof-carrying authorization systems. In *Proceedings of the European Symposium on Research in Computer Security (ESORICS)*, pages 184–199, 2009.

[4] Deepak Garg and Martín Abadi. A modal deconstruction of access control logics. In *Proceedings of the 11th International Conference on Foundations of Software Science and Computation Structures (FoSSaCS 2008)*, pages 216–230, April 2008.

[5] Deepak Garg and Frank Pfenning. Non-interference in constructive authorization logic. In *Proceedings of the 19th Computer Security Foundations Workshop (CSFW '06)*, pages 283–293, July 2006.

[6] Limin Jia, Jeffrey A. Vaughan, Karl Mazurak, Jianzhou Zhao, Luke Zarko, Joseph Schorr, and Steve Zdancewic. Aura: A programming language for authorization and audit. In *Proceedings of the International Conference on Functional Programming (ICFP)*, pages 27–38, 2008.

## A Additional Details of the Type-System

The rules for type-checking expressions of IPCA are shown in Figure 5.

## B Additional Details of the Operational Semantics

The rules for evaluating expressions of IPCA are shown in Figure 6.

$\boxed{\text{Types for expressions}}$

$$\frac{(x : T) \in \Delta}{\Delta \vdash x : T} \qquad \frac{}{\Delta \vdash \mathsf{true} : \mathsf{bool}} \qquad \frac{}{\Delta \vdash \mathsf{false} : \mathsf{bool}} \qquad \frac{n \in \{\ldots, -1, 0, 1, \ldots\}}{\Delta \vdash n : \mathsf{int}}$$

$$\frac{s \in \{\text{“a.txt”}, \text{“Alice”}, \ldots\}}{\Delta \vdash s : \mathsf{string}} \qquad \frac{A \in \{\mathsf{Alice}, \mathsf{Bob}, \ldots\}}{\Delta \vdash A : \mathsf{principal}} \qquad \frac{\Delta \vdash e_1 : \mathsf{int} \qquad \Delta \vdash e_2 : \mathsf{int}}{\Delta \vdash e_1 + e_2 : \mathsf{int}}$$

$$\frac{\Delta \vdash e_1 : \mathsf{int} \qquad \Delta \vdash e_2 : \mathsf{int}}{\Delta \vdash e_1 * e_2 : \mathsf{int}} \qquad \frac{\Delta \vdash e_1 : \mathsf{int} \qquad \Delta \vdash e_2 : \mathsf{int}}{\Delta \vdash e_1 <= e_2 : \mathsf{bool}} \qquad \frac{\Delta \vdash e_1 : \mathsf{int} \qquad \Delta \vdash e_2 : \mathsf{int}}{\Delta \vdash e_1 == e_2 : \mathsf{bool}}$$

$$\frac{\Delta \vdash e_1 : \mathsf{string} \qquad \Delta \vdash e_2 : \mathsf{string}}{\Delta \vdash \mathtt{concat}(e_1, e_2) : \mathsf{string}}$$

**Figure 5:** Type system of IPCA (expressions)

$\boxed{\sigma \rhd e \hookrightarrow e'}$

$$\frac{(x \mapsto v) \in \sigma}{\sigma \rhd x \hookrightarrow v} \qquad \frac{\sigma \rhd e_1 \hookrightarrow e_1'}{\sigma \rhd e_1 + e_2 \hookrightarrow e_1' + e_2} \qquad \frac{\sigma \rhd e_2 \hookrightarrow e_2'}{\sigma \rhd v_1 + e_2 \hookrightarrow v_1 + e_2'} \qquad \frac{\mathtt{add}(n_1, n_2) = n}{\sigma \rhd n_1 + n_2 \hookrightarrow n}$$

$$\frac{\sigma \rhd e_1 \hookrightarrow e_1'}{\sigma \rhd e_1 * e_2 \hookrightarrow e_1' * e_2} \qquad \frac{\sigma \rhd e_2 \hookrightarrow e_2'}{\sigma \rhd v_1 * e_2 \hookrightarrow v_1 * e_2'} \qquad \frac{\mathtt{mult}(n_1, n_2) = n}{\sigma \rhd n_1 * n_2 \hookrightarrow n}$$

$$\frac{\sigma \rhd e_1 \hookrightarrow e_1'}{\sigma \rhd e_1 <= e_2 \hookrightarrow e_1' <= e_2} \qquad \frac{\sigma \rhd e_2 \hookrightarrow e_2'}{\sigma \rhd v_1 <= e_2 \hookrightarrow v_1 <= e_2'} \qquad \frac{\mathtt{leq}(n_1, n_2) = b}{\sigma \rhd n_1 <= n_2 \hookrightarrow b}$$

$$\frac{\sigma \rhd e_1 \hookrightarrow e_1'}{\sigma \rhd e_1 == e_2 \hookrightarrow e_1' == e_2} \qquad \frac{\sigma \rhd e_2 \hookrightarrow e_2'}{\sigma \rhd v_1 == e_2 \hookrightarrow v_1 == e_2'} \qquad \frac{\mathtt{eq}(n_1, n_2) = b}{\sigma \rhd n_1 == n_2 \hookrightarrow b}$$

$$\frac{\sigma \rhd e_1 \hookrightarrow e_1'}{\sigma \rhd \mathtt{concat}(e_1, e_2) \hookrightarrow \mathtt{concat}(e_1', e_2)} \qquad \frac{\sigma \rhd e_2 \hookrightarrow e_2'}{\sigma \rhd \mathtt{concat}(v_1, e_2) \hookrightarrow \mathtt{concat}(v_1, e_2')}$$

$$\frac{\mathtt{strcat}(v_1, v_2) = v}{\sigma \rhd \mathtt{concat}(v_1, v_2) \hookrightarrow v}$$

**Figure 6:** Operational semantics of IPCA (expressions)