Access Control in a Core Calculus of Dependency

Martín Abadi

Computer Science Department
University of California, Santa Cruz
and
Microsoft Research, Silicon Valley

Abstract

The Dependency Core Calculus (DCC) is an extension of the computational lambda calculus that was designed in order to capture the notion of dependency that arises in information-flow control, partial evaluation, and other programming-language settings. We show that, unexpectedly, DCC can also be used as a calculus for access control in distributed systems. Initiating the study of DCC from this perspective, we explore some of its appealing properties.

Keywords: Authorization, Types

1 Introduction

The Dependency Core Calculus (DCC) [3] is a small extension of Moggi's computational lambda calculus [22]. DCC was designed in order to capture the common, central notion of dependency that arises in information-flow control, partial evaluation, call-tracking, and other programming-language settings. The paper that introduces DCC includes six translations from type-based dependency analyses into DCC. As explained there, this use of the computational lambda calculus in describing dependency was somewhat surprising.

Usually, the computational lambda calculus describes languages with side effects [22], or forms the basis of adding side effects like I/O to pure functional languages [14]. Dependency analyses, in contrast, do not fundamentally change the values being computed. Nevertheless, there is one common idea underlying both uses of the computational lambda calculus. In the case of Haskell, there is no way to compute a value using the I/O type constructor and pass that value to an expression of non-I/O type. Similarly, in information-flow systems, the test of a high-security boolean in an "if-then-else" requires that the branches of

the conditional return high-security values. In both cases, the type rules of the computational lambda calculus enforce the necessary restriction.

In this paper we show that, also unexpectedly, DCC can be used as a calculus for access control in distributed systems. In this application, the restriction that DCC imposes implies that there is no way to take evidence that a principal A has made a statement s and use it non-trivially in evidence that an unrelated principal B has made a statement t.

Access control basically consists in determining whether the principal that issues a request should be trusted on this request. Accordingly, logics for access control enable reasoning about principals, their requests, and other statements [4,16,19,9,6]. (See [1] for further discussion of several of those logics and additional references.) Often, the logics include formulas such as A `says` s, where A represents a principal, s represents a statement (a request, a delegation of authority, or some other utterance), and `says` is an operator. The use of `says` abstracts from details of authentication and authorization. One may have that A `says` s even when A does not directly utter s. For example, when A is a user and one of its programs includes s in a message, it may be convenient and appropriate to state that A `says` s. In this case, A `says` s means that A has caused s to be said, that s has been said on A's behalf, or that A supports s.

DCC can be seen as an alternative to the core of those previous logics for access control. DCC promises some appealing features in this respect: simple rules with useful consequences, a precise semantics, and good fundamental properties.

- The rules imply the "hand-off axiom" [16], which has had an unclear status until now despite its central role in many access control situations. This axiom basically asserts that, for two principals A and B, if A says that B speaks for A then B does speak for A. It was often treated as an add-on, not as a consequence of more fundamental rules or semantic definitions.

  Secondarily, the rules also imply that if s is true then A says s, for every principal A and statement s. This inference may be desirable but it can give rise to unexpected conclusions, such as

  $$A \texttt{ says } s \to (s \vee (A \texttt{ says } s'))$$

  for every s and s' [1]. Here, those conclusions are blocked by the use of constructive reasoning. (Garg and Pfenning [11] found this benefit of constructivity in the context of a different formal system; see Section 2.)

  Of course, constructivity has a price, and it can be hard to motivate. Why would one abandon the law of the excluded middle? is it not true, for instance, that a principal A says "delete the file" or that it does not? DCC addresses this difficulty, in a sense. It is not for reasoning about classical truth, but rather serves as a type system for evidence, and as such it should not be too surprising that it is constructive—since one does not always have evidence for a proposition or for its negation.
- The semantics is the sort of explanation of the calculus provided by the theory of programming languages (in contrast with classical possible-world semantics

for modal logics [4]). The semantics does not directly explain concepts such as responsibility and authority, at least in its present form. In the semantics, the meaning of a statement can be seen roughly as the set of arguments that can lead to concluding the statement. The arguments might represent digital certificates and other sorts of evidence, pieced together with logical reasoning.

- The fundamental properties include some of those traditionally proved for type systems. In particular, one can formulate various consistency, type soundness, parametricity, and noninterference results.

  These results sometimes take new forms and have new applications. For instance, exploring noninterference, we prove a theorem (Theorem 7.6) that limits the possible influence of the statements of one principal on other principals.

  Some of the properties of DCC are already considered in previous work [3,27]. Although we do not attempt a full development of the metatheory of DCC, we advance it with some simple but important proof-theoretic results.

While these features may not all be unique to DCC, we believe that DCC offers a rather attractive package, worth investigating. In addition, it seems both interesting and valuable that DCC is not an ad hoc calculus invented purely for the purposes of reasoning about access control.

Beyond specific technical results, the main contributions of this paper are the reconsideration of DCC as a system for access control and, more broadly, the use of the Curry-Howard isomorphism ("propositions-as-types, proofs-as-programs") in access control. The Curry-Howard isomorphism had not, it seems, ever been exploited in this context, and it appears to have great potential. Throughout, we focus on the rules of DCC and their interpretation. We leave several opportunities for further work on extensions and on the relevant theory.

The next section (Section 2) discusses some of the sources of this work. Section 3 revisits DCC, specifically defining a system that we call Simply Typed DCC. This is a terminating fragment of DCC as it was previously defined, with only superficial changes of notation. Section 4 considers how to use Simply Typed DCC for access control. Going further, Section 5 adds polymorphic types, in the style of Girard's System F [13,7]; the result is a system that we call Polymorphic DCC. Section 6 shows how this extension enables a new treatment of the "speaks for" relation and also outlines examples. Section 7 concerns the metatheory of Polymorphic DCC. Section 8 presents an attractive fragment of Polymorphic DCC. Section 9 concludes with a brief discussion of the prospects for DCC.

This paper is a revised version of a conference paper [2].

2 Related Work

The main sources of this work include, of course, the original papers on calculi for access control and on DCC (e.g., [1,3]). Via DCC and the computational lambda calculus, this work is also related to a large body of research on programming languages and logics (e.g., [24,10]).

The other important and more immediate influences on this work are an in-

sightful suggestion from Langworthy [18] and an intriguing manuscript by Garg and Pfenning [11]. In that manuscript, they develop a new logic for access control. That logic is an intuitionistic variant of previous logics, with an apparently pleasant proof theory. It is reminiscent of information-flow type systems, and we discovered that it has similarities with DCC (of which Garg and Pfenning were not aware). On the other hand, unlike DCC, that logic is not defined as a type system for a programming language. Furthermore, it lacks a "speaks for" relation with a hand-off property.

DCC, as we use it here, is not literally as it was originally defined. Technically, the changes are important but routine. Specifically, we remove non-terminating programs and the corresponding notion of "pointed types"; non-termination is attractive in a general-purpose type system but not usually in a proof system. On the other hand, we introduce type quantification in the style of System F. This type quantification enables us to model the "speaks for" relation. Finally, we change notations, for example writing A `says` s instead of $T_\ell(s)$, with A instead of ℓ. Curiously, this superficial syntactic change may be regarded as a more profound innovation than removing non-terminating programs and adding type quantification.

3 Simply Typed DCC

In this section we define Simply Typed DCC and indicate how to read it in logical terms.

3.1 The Calculus

Three features distinguish Simply Typed DCC from the standard computational lambda calculus [22]. First, Simply Typed DCC includes sum types. Second, instead of having one type constructor T semantically associated with a monad, the calculus incorporates multiple type constructors T_A (here written A `says` ...), one for every element $A \in \mathcal{L}$ of a lattice $\mathcal{L}$. (Wadler [28] considered this idea as well.) In our use of this lattice, the elements of $\mathcal{L}$ will represent principals, with smaller elements as more trusted principals and greater elements as less trusted ones. Third, the monadic operation "bind" has a special typing rule that is explained below.

3.1.1 Types

The types of Simply Typed DCC are given by the grammar:

$$s ::= \texttt{true} \mid (s \vee s) \mid (s \wedge s) \mid (s \to s) \mid A \texttt{ says } s$$

where A ranges over elements of a lattice $\mathcal{L}$, equipped with a partial order $\sqsubseteq$.

Thus, we depart from the original DCC notations: we write

- `true` instead of `unit`,
- $s_1 \vee s_2$ instead of $s_1 + s_2$,
- $s_1 \wedge s_2$ instead of $s_1 \times s_2$, and
- A `says` s instead of $T_A(s)$.

Table 1
Simply Typed DCC: Typing Rules.

$$[Var] \quad \Gamma, x : s, \Gamma' \vdash x : s \qquad [Unit] \quad \Gamma \vdash () : \mathtt{true}$$

$$[Lam] \quad \frac{\Gamma, x : s_1 \vdash e : s_2}{\Gamma \vdash (\lambda x : s_1.\ e) : (s_1 \to s_2)} \qquad [App] \quad \frac{\Gamma \vdash e : (s_1 \to s_2) \quad \Gamma \vdash e' : s_1}{\Gamma \vdash (e\ e') : s_2}$$

$$[Pair] \quad \frac{\Gamma \vdash e_1 : s_1 \quad \Gamma \vdash e_2 : s_2}{\Gamma \vdash \langle e_1, e_2 \rangle : (s_1 \wedge s_2)}$$

$$[Proj\ 1] \quad \frac{\Gamma \vdash e : (s_1 \wedge s_2)}{\Gamma \vdash (\mathtt{proj}_1\ e) : s_1} \qquad [Proj\ 2] \quad \frac{\Gamma \vdash e : (s_1 \wedge s_2)}{\Gamma \vdash (\mathtt{proj}_2\ e) : s_2}$$

$$[Inj\ 1] \quad \frac{\Gamma \vdash e : s_1}{\Gamma \vdash (\mathtt{inj}_1\ e) : (s_1 \vee s_2)} \qquad [Inj\ 2] \quad \frac{\Gamma \vdash e : s_2}{\Gamma \vdash (\mathtt{inj}_2\ e) : (s_1 \vee s_2)}$$

$$[Case] \quad \frac{\Gamma \vdash e : (s_1 \vee s_2) \quad \Gamma, x : s_1 \vdash e_1 : s \quad \Gamma, x : s_2 \vdash e_2 : s}{\Gamma \vdash (\mathtt{case}\ e\ \mathtt{of}\ \mathtt{inj}_1(x).\ e_1 \mid \mathtt{inj}_2(x).\ e_2) : s}$$

$$[UnitM] \quad \frac{\Gamma \vdash e : s}{\Gamma \vdash (\eta_A\ e) : A\ \mathtt{says}\ s}$$

$$[BindM] \quad \frac{\Gamma \vdash e : A\ \mathtt{says}\ s \quad \Gamma, x : s \vdash e' : t}{\Gamma \vdash \mathtt{bind}\ x = e\ \mathtt{in}\ e' : t} \quad t \text{ is protected at level } A$$

For each $A \in \mathcal{L}$, the `says` operation induces a subset of types called the types protected at level A:

- If $A \sqsubseteq B$, then B `says` s is protected at level A;
- `true` is protected at level A;
- if s and t are protected at level A, then $(s \wedge t)$ is protected at level A;
- if t is protected at level A, then B `says` t is protected at level A; and
- if t is protected at level A, then $(s \to t)$ is protected at level A.

(Tse and Zdancewic [27] have proposed a generalization of this definition, which may be helpful in the future. The clause that says that `true` is protected at level A comes from them.)

3.1.2 Terms

The terms and the typing rules for Simply Typed DCC appear in Table 1.

These rules are for proving typing judgments of the form $\Gamma \vdash e : s$ (read "term e has type s in typing environment Γ"). Here, a typing environment Γ denotes a

list of distinct variables with types. The rules for unit, function, product, and sum types are all standard, as is the rule for the monadic unit operation. The rule for the monadic bind operation is nonstandard, using the concept of "protected at level A" for the body of bind expressions.

3.1.3 Semantics

The paper that introduces DCC provides both an operational semantics and a denotational semantics for the Simply Typed calculus (including constructs not considered here, in particular term recursion).

The operational semantics is a call-by-name semantics. The term $(\eta_A\ e)$ reduces to e, and $(\texttt{bind}\ x = e\ \texttt{in}\ e')$ reduces to $e'[e/x]$, where $e[e'/x]$ represents the result of the capture-free substitution of e' for x in e. The rest of the operational semantics of Simply Typed DCC is standard.

This operational semantics is not entirely consistent with the type system. Specifically, the term $(\eta_A\ e)$ reduces to e although these two terms have different types, and the term $(\texttt{bind}\ x = e\ \texttt{in}\ e')$ reduces to $e'[e/x]$ although e and x do not have the same type. This discrepancy is however limited. (Reduction preserves types modulo applications of the function $(\cdot)^{\mathrm{F}}$ of Section 7.) An alternative operational semantics [27] fits more closely with the type system. In this semantics, in addition to standard rules from the lambda calculus, we have that $\texttt{bind}\ x = (\eta_A\ e)\ \texttt{in}\ e'$ reduces to $e'[e/x]$. Zdancewic has recently checked progress and subject-reduction properties of this semantics, using the Twelf proof assistant [31]. For our present purposes, either operational semantics is satisfactory.

The denotational semantics of Simply Typed DCC is more intricate. We omit it here. It may however be useful in further work, in particular because it captures noninterference properties.

3.2 Logical Reading

We restate the rules of Simply Typed DCC in logical form in Table 2. Here, an environment Γ denotes a list of types. We simply omit all the terms. This move from type systems to logics is an instance of the Curry-Howard isomorphism. We show the resulting rules in Table 2 for the sake of clarity.

According to these rules, if a type t is protected at level A, then t and $A\ \texttt{says}\ t$ are logically equivalent. Therefore, up to logical equivalence, the types protected at level A are those of the form $A\ \texttt{says}\ t$ for some t.

We write $\vdash s$, and say that s is a theorem, when $\vdash s$ is derivable by the rules of Table 2. Equivalently, s is a theorem when there is a term e such that $\vdash e : s$ is derivable by the rules of Table 1. In this case, we say that e inhabits s, and e represents a proof of s.

In practice, this proof may be transmitted as evidence of s in requests for access to a resource [30,5]. For this purpose, it may be useful to decorate terms with additional type information, so that proof-checking can be as straightforward as possible. In particular, such type information may help in applying the rule [*BindM*]

Table 2
Simply Typed DCC: Logical Reading.

$$[Var] \quad \Gamma, s, \Gamma' \vdash s \qquad [Unit] \quad \Gamma \vdash \mathtt{true}$$

$$[Lam] \quad \frac{\Gamma, s_1 \vdash s_2}{\Gamma \vdash (s_1 \to s_2)} \qquad [App] \quad \frac{\Gamma \vdash (s_1 \to s_2) \quad \Gamma \vdash s_1}{\Gamma \vdash s_2}$$

$$[Pair] \quad \frac{\Gamma \vdash s_1 \quad \Gamma \vdash s_2}{\Gamma \vdash (s_1 \wedge s_2)}$$

$$[Proj\ 1] \quad \frac{\Gamma \vdash (s_1 \wedge s_2)}{\Gamma \vdash s_1} \qquad [Proj\ 2] \quad \frac{\Gamma \vdash (s_1 \wedge s_2)}{\Gamma \vdash s_2}$$

$$[Inj\ 1] \quad \frac{\Gamma \vdash s_1}{\Gamma \vdash (s_1 \vee s_2)} \qquad [Inj\ 2] \quad \frac{\Gamma \vdash s_2}{\Gamma \vdash (s_1 \vee s_2)}$$

$$[Case] \quad \frac{\Gamma \vdash (s_1 \vee s_2) \quad \Gamma, s_1 \vdash s \quad \Gamma, s_2 \vdash s}{\Gamma \vdash s}$$

$$[UnitM] \quad \frac{\Gamma \vdash s}{\Gamma \vdash A \ \mathtt{says}\ s}$$

$$[BindM] \quad \frac{\Gamma \vdash A \ \mathtt{says}\ s \quad \Gamma, s \vdash t}{\Gamma \vdash t} \quad t \text{ is protected at level } A$$

backwards—with the rule in its present form, we have to guess A and s in order to find the hypotheses that yield a conclusion. We resist this and other temptations to modify the language of proofs, in order to minimize changes to DCC and to emphasize its surprising applicability to access control.

Simply Typed DCC has the property that every type is inhabited, and therefore Simply Typed DCC may not be regarded as a very useful logic. Fortunately, it is easy to enrich Simply Typed DCC with atomic types that represent basic propositions; those are convenient in applications, and they provide uninhabited types. We do not introduce them here, though, in order to minimize deviations from the original definition of DCC and because Polymorphic DCC includes type variables that can represent basic propositions (see Section 5).

4 Access Control in Simply Typed DCC

Next we suggest how to use Simply Typed DCC for access control. We start with a basic, general discussion of the use of logics for access control, then focus on some

of the features of Simply Typed DCC.

4.1 Basics

In logical approaches to access control, the problem of deciding whether an operation should be granted is formulated in logical terms, as the problem of constructing or checking a proof. For instance, a logical formula s may represent that a particular operation o should be performed. In this case, s may be written as a proposition of the form $\mathrm{Do}(o)$ (or $\mathrm{Ok}(o)$ [29]). The reference monitor in charge of making access control decisions for o may have the policy that a particular principal A is authorized to perform o. This policy may be represented by the formula:

$$(A \texttt{ says } \mathrm{Do}(o)) \rightarrow \mathrm{Do}(o)$$

Similarly, a request for the operation o from a principal B may be represented by the formula:

$$B \texttt{ says } \mathrm{Do}(o)$$

The reference monitor may attempt to prove that these two formulas imply $\mathrm{Do}(o)$, and grant access if it succeeds. In general, the proof may exploit relations between A and B and other facts known to the reference monitor. Alternatively, the reference monitor may simply check a proof presented by B. Thus, when the proof is a DCC term, the reference monitor may simply do type-checking.

4.2 Properties of `says`

In Simply Typed DCC, the operators $\vee$, $\wedge$, and $\rightarrow$ obey the standard intuitionistic propositional rules. In addition, we have

$$\vdash s \rightarrow A \texttt{ says } s$$

with proof term

$$\lambda x : s.\ (\eta_A\ x)$$

and

$$\vdash (A \texttt{ says } (s \rightarrow s')) \rightarrow ((A \texttt{ says } s) \rightarrow (A \texttt{ says } s'))$$

with proof term

$$\lambda x : A \texttt{ says } (s \rightarrow s').\ \texttt{bind } x' = x \texttt{ in } \lambda y : A \texttt{ says } s.\ \texttt{bind } y' = y \texttt{ in } (\eta_A\ (x'y'))$$

These theorems correspond to the main axioms and rules adopted in previous calculi (despite the switch to intuitionistic reasoning). Therefore, we have enough to perform much of the typical, basic reasoning about access control.

In fact, the theorem $\vdash s \rightarrow A \texttt{ says } s$ is even stronger than what we usually have. Usually, we have only the necessitation rule from modal logic, which says only that if $\vdash s$ then $\vdash A \texttt{ says } s$. The necessitation rule avoids the unexpected consequences of $\vdash s \rightarrow A \texttt{ says } s$ in a classical-logic context [1].

The theorem $\vdash s \rightarrow A \texttt{ says } s$ suggests that $A \texttt{ says } s$ does not mean that A actually utters s or something that implies it. Rather, we should see all reasoning from the point of view of a reference monitor that is in charge of making access

control decisions. Informally, we have A `says` s when, combining the statements that the reference monitor believes with those that A contributes, the reference monitor can conclude s. Thus, the reference monitor's participation is left implicit.

Further, we have

$$\vdash (A \texttt{ says } A \texttt{ says } s) \to (A \texttt{ says } s)$$

with proof term

$$\lambda x : A \texttt{ says } A \texttt{ says } s.\, \texttt{bind } y = x \texttt{ in } y$$

(cf. [4, Section 3.4]), and also

$$\vdash (A \texttt{ says } B \texttt{ says } s) \to (B \texttt{ says } A \texttt{ says } s)$$

with proof term

$$\lambda x : A \texttt{ says } B \texttt{ says } s.\, \texttt{bind } y = x \texttt{ in bind } z = y \texttt{ in } (\eta_B\ (\eta_A\ z))$$

These properties sometimes simplify reasoning about chains of `says`: they imply that when

$$A_1 \texttt{ says } A_2 \ldots \texttt{ says } A_n \texttt{ says } s$$

the order of the principals $A_1, \ldots, A_n$ and how often they occur in the chain do not matter. In this respect, saying is much like tainting in systems that aim to guarantee data integrity: the order of taints and how often they happen typically do not affect the resulting integrity level. However, the commutativity property $(A \texttt{ says } B \texttt{ says } s) \to (B \texttt{ says } A \texttt{ says } s)$ is not central to our approach, and in Section 8 we show how to omit it.

4.3 Using the Partial Order, and its Connection to "Speaks for"

In our application of DCC, the least element of the lattice of levels stands for the most trusted principal, and greater elements represent less trusted principals, as indicated in Section 3. The opposite ordering is typically used for obtaining secrecy guarantees with information-flow type systems, and with DCC in particular (e.g., [3]). The present ordering is fairly common in models for integrity. Perhaps this coincidence should be expected, since reliable access control requires the integrity of requests and policies.

We have the following theorem:

$$\vdash (A \texttt{ says } s) \to (B \texttt{ says } s) \qquad \text{for } A \sqsubseteq B$$

Therefore, we could regard $\sqsubseteq$ as a representation of the "speaks for" relation. Indeed, with that reading, we obtain that if A speaks for B and A `says` s then B `says` s, for every s. This property is characteristic of the "speaks for" relation.

However, Simply Typed DCC is not a rich setting for reasoning about the "speaks for" relation. In particular, we cannot use the symbol $\sqsubseteq$ in expressions such as A `says` $(B \sqsubseteq A)$: this expression is not even a syntactically legal formula. A possible solution may be to extend the syntax so that $(B \sqsubseteq A)$ is a type, thus internalizing $\sqsubseteq$. While this approach may be viable, it is somewhat ad hoc. Below we develop a more principled approach, relying on polymorphism.

The lattice structure also provides operators $\sqcap$ and $\sqcup$, respectively the meet and the join operators of the lattice $\mathcal{L}$.

Since $A \sqcap B \sqsubseteq A$ and $A \sqcap B \sqsubseteq B$, we obtain:

$$\vdash (A \sqcap B \texttt{ says } s) \rightarrow (A \texttt{ says } s) \wedge (B \texttt{ says } s) \tag{1}$$

for every s. However, the converse does not automatically hold for every s.[1] Intuitively, $A \sqcap B$ may be more trustworthy than A and B, even when A and B happen to agree. In this sense, a joint signature of a statement may carry more weight than two separate signatures of the same statement.

Similarly, since $A \sqsubseteq A \sqcup B$ and $B \sqsubseteq A \sqcup B$, we obtain:

$$\vdash (A \texttt{ says } s) \vee (B \texttt{ says } s) \rightarrow (A \sqcup B \texttt{ says } s) \tag{2}$$

for every s, but not the converse. Intuitively, $A \sqcup B$ may be less trustworthy than each of A and B in isolation. Because saying is closed under logical consequence, a statement by $A \sqcup B$ may be derived from separate statements by A and B, with results that neither A nor B would have produced alone.

These properties suggest that we may interpret lattice elements as groups, $\sqcap$ as group intersection, $\sqcup$ as group union, and $\sqsubseteq$ as group inclusion. Thus, an individual element of the lattice $\mathcal{L}$ may not (always) correspond to an atomic subject, but rather to a group of subjects. It then seems quite reasonable that statements by $A \sqcap B$ may be more trusted than statements by both A and B. For example, a security policy may refer to two groups, "club members" and "adults", and require a certain authorization by an adult club member; a statement by an adult club member cannot be replaced with identical statements by an arbitrary club member and an arbitrary adult.

Alternatively, the lattice $\mathcal{L}$ may be a lattice of abstract security levels, of the kind that may be used in multilevel security (e.g., [8,12]). In such a lattice, the converses of (1) and (2) can have strong, questionable consequences. For example, let us consider a five-element lattice with a bottom, a top, and three elements A, B, and C unrelated to one another. In this lattice, $A \sqcap B$ is the bottom element, $A \sqcup B$ is the top element, and hence $A \sqcap B \sqsubseteq C \sqsubseteq A \sqcup B$. If A says s and B says s implied $A \sqcap B$ says s, then it would also follow that C says s. Similarly, if $A \sqcup B$ says s implied that A says s or B says s, then C says s would also imply that A says s or B says s.

In many of these scenarios, and in particular when $\mathcal{L}$ is a lattice of groups, the lattice $\mathcal{L}$ is distributive. In general, the lattice $\mathcal{L}$ is not required to be distributive, but there is no contradiction in adding that condition; the rules of DCC apply without change.

---

[1] More precisely, the converse cannot be derived as soon as we introduce basic propositions, for example in the form of type variables. As long as every type is inhabited, the converse can of course be derived, but not for interesting reasons.
Similar caveats apply to discussions of some other formulas in the rest of Section 4.

Table 3
Polymorphic DCC: Additional Typing Rules.

$$[TLam]\ \frac{\Gamma, X \vdash e : s}{\Gamma \vdash (\Lambda X.\, e) : \forall X.\, s} \quad [TApp]\ \frac{\Gamma \vdash e : \forall X.\, s}{\Gamma \vdash (et) : s[t/X]}\ (t \text{ well-formed in } \Gamma)$$

Table 4
Polymorphic DCC: Logical Form of the Additional Typing Rules.

$$[TLam]\ \frac{\Gamma \vdash s}{\Gamma \vdash \forall X.\, s}\ (X \text{ not free in } \Gamma) \quad [TApp]\ \frac{\Gamma \vdash \forall X.\, s}{\Gamma \vdash s[t/X]}$$

4.5 Conjunctions and Disjunctions on Principals

Despite important differences, the operators $\sqcap$ and $\sqcup$ may be reminiscent of the operators $\wedge$ and $\vee$ that have in the past been applied to principals. We could introduce those operators as abbreviations:

$$A \wedge B\ \mathtt{says}\ s \quad \text{stands for} \quad (A\ \mathtt{says}\ s) \wedge (B\ \mathtt{says}\ s)$$

$$A \vee B\ \mathtt{says}\ s \quad \text{stands for} \quad (A\ \mathtt{says}\ s) \vee (B\ \mathtt{says}\ s)$$

While these abbreviations are convenient, it is worth noting that $A \wedge B$ and $A \vee B$ need not behave exactly like lattice elements. For instance, it should not be taken for granted that

$$(A \vee B\ \mathtt{says}\ (s \to s')) \to ((A \vee B\ \mathtt{says}\ s) \to (A \vee B\ \mathtt{says}\ s'))$$

holds for every s and s': when $A\ \mathtt{says}\ s$ and $B\ \mathtt{says}\ (s \to s')$, we have $A \vee B\ \mathtt{says}\ s$ and $A \vee B\ \mathtt{says}\ (s \to s')$, but not necessarily $A\ \mathtt{says}\ s'$ or $B\ \mathtt{says}\ s'$.

4.6 Additional Operations

Going further, we could add other abbreviations and other operators on the lattice, in order to represent useful compound principals. Such extensions have been explored in previous work (e.g., [4,16]). There, the operator | ("quoting") plays a central role, so we revisit it briefly in this section. We leave to further work a more systematic study of additional operators.

When A and B are two principals, $A \mid B$ is a principal that represents A quoting B. A fundamental property of $A \mid B$ (sometimes regarded as its definition) is the equivalence of $A \mid B\ \mathtt{says}\ s$ with $A\ \mathtt{says}\ B\ \mathtt{says}\ s$, for every s.

In the context of Simply Typed DCC, the properties of says listed in Section 4.2 suggest strong axioms for the operator |. Specifically, we may take | to be associative,

commutative, and idempotent. Furthermore, the theorem

$$\vdash (A \texttt{ says } s) \to (A' \texttt{ says } s) \qquad \text{for } A \sqsubseteq A'$$

(see Section 4.3) suggests that | could be monotone with respect to $\sqsubseteq$ in its first argument, and by commutativity also in its second argument. It follows that

$$(A \mid B) \sqsubseteq ((A \sqcup B) \mid (A \sqcup B)) = (A \sqcup B)$$

Conversely, for all s, if either A `says` s or B `says` s then $A \mid B$ `says` s. It is therefore attractive to let $(A \sqcup B) \sqsubseteq (A \mid B)$, since one may think of $A \sqcup B$ as the least principal that says everything that either A or B says. This property requires a substantial strengthening of Simply Typed DCC, since it implies

$$(A \sqcup B) \texttt{ says } s \to A \texttt{ says } B \texttt{ says } s$$

The strengthening might be achieved by relaxing the type system in such a way that A `says` B `says` s is protected at level $A \sqcup B$.

In summary, in the context of Simply Typed DCC, it is tempting to equate the quoting operator | with the join operator $\sqcup$. While this equation seems reasonable and avoids the introduction of new syntax for |, it may lead to significant changes in Simply Typed DCC.

5 Polymorphic DCC

We extend DCC with polymorphism in the style of System F. We rely on System F because of its conciseness and power; as usual, one may also consider more restricted forms of polymorphism.

This extension addresses the shortcoming described in Section 4.3: in Section 6 we show that we can use polymorphism in order to represent and reason about the "speaks for" relation. We can also use polymorphism in other ways, of course. In particular, we consider how to use polymorphism for encoding `says`.

A common practical criticism of polymorphic types is that they introduce serious complications in type inference. We note that this criticism does not apply for the present application. Another practical criticism of polymorphic types is that they can be hard to write and to understand. This criticism does apply. We would not recommend writing security policies with many quantifiers. In practice, security policies could often be written in terms of a few idioms whose definitions may rely on quantifiers, but without explicit quantifiers. In this direction, it is attractive to explore the development of languages for writing security policies, perhaps in a logic-programming style (as in SD3 [15], Binder [9], and RT [20]).

5.1 The Calculus

The extension to Polymorphic DCC relies only on standard rules for polymorphism, introducing universal quantification over types. We review those rules in this section.

5.1.1 *Types*

With polymorphism, the types of DCC are given by the grammar:

$$s ::= \texttt{true} \mid (s \vee s) \mid (s \wedge s) \mid (s \to s) \mid A \texttt{ says } s \mid X \mid \forall X.\, s$$

where A ranges over elements of a lattice $\mathcal{L}$, and X ranges over a set of type variables. The variable X is bound in $\forall X.\, s$, and subject to renaming.

Again, the `says` operation induces a subset of types called the types protected at level A. We add one clause to the definition of Section 3.1.1:

- if t is protected at level A, then $\forall X.\, t$ is protected at level A.

This simple clause is motivated by a straightforward analogy between $\forall$ and $\wedge$.

5.1.2 *Terms*

The additional forms of terms and the additional typing rules for Polymorphic DCC appear in Table 3. These are in addition to those of Table 1.

Here, a typing environment Γ denotes a list where each element is either a distinct type variable or a distinct variable with a type. We require that if the type of a variable mentions a type variable X, then X occurs further to the left. Thus, for example, $x : X$ and $x : X, X$ are not well-formed typing environments, while $X, x : X$ and

$$X, Y, x : X, y : A \texttt{ says } (X \to Y)$$

are well-formed typing environments.

In rule [*TApp*], we require that t be a well-formed type in Γ. This condition means that any type variables in t should be declared in Γ. Also in rule [*TApp*], the expression $s[t/X]$ represents the result of the capture-free substitution of t for X in s.

5.1.3 *Semantics*

It is straightforward to extend the operational semantics to Polymorphic DCC, along standard lines (e.g., [21]). Zdancewic's proofs, mentioned in Section 3.1.3, apply to this extension as well.

Extending the denotational semantics is somewhat more challenging. We leave this extension for further work.

5.2 *Logical Reading*

The logical reading of Simply Typed DCC, outlined in Section 3.2, generalizes to Polymorphic DCC.

The additional rules are given in logical form in Table 4. Here, a typing environment Γ denotes a list of types; type variables may occur free in those types. We simply omit all the terms, as well as declarations for type variables. For example, the typing environment

$$X, Y, x : Y \to X, y : A \texttt{ says } Y$$

yields the environment

$$Y \to X, A \texttt{ says } Y$$

and the typing judgment

$$X, Y, x : Y \to X, y : A \texttt{ says } Y \vdash \texttt{bind } y' = y \texttt{ in } (\eta_A\ (xy')) : A \texttt{ says } X$$

yields

$$Y \to X, A \texttt{ says } Y \vdash A \texttt{ says } X$$

In judgments such as this one, type variables that are not quantified can be read and used as basic propositions.

In rule [*TLam*] of Table 4, we make explicit that X should not occur free in Γ. In Table 3, the corresponding requirement is implied by the fact that Γ, X is a well-formed typing environment.

Much as in Section 3.2, we write $\vdash s$, and say that s is a theorem, when $\vdash s$ is derivable by the rules of Tables 2 and 4. Equivalently, s is a theorem when there is a list of type variables $X_1, \ldots, X_n$ and a term e such that $X_1, \ldots, X_n \vdash e : s$ is derivable by the rules of Tables 1 and 3.

In particular, we immediately obtain:

$$\vdash (A \texttt{ says } \forall X.\ s) \to (\forall X.\ A \texttt{ says } s)$$

with proof term

$$\lambda x : (A \texttt{ says } \forall X.\ s).\ \Lambda X.\ \texttt{bind } y = x \texttt{ in } (\eta_A\ (yX))$$

but not the converse implication—which may be harmless, but which we do not seem to need.

5.3 Translating Access Control into Parametricity

Tse and Zdancewic [27] have shown an encoding of Simply Typed DCC into System F. Crucially, they map a type of the form $A \texttt{ says } s$ to $\alpha_A \to s$, where α_A is a distinct type variable. The translations of other DCC constructs are mostly straightforward. Tse and Zdancewic do not treat non-termination and related notions, which fortunately we do not need for our present purposes. It seems possible to extend their encoding to Polymorphic DCC. The end result would be an interesting representation of access control purely in System F.

We leave the study of this representation for further work. Theorems beyond those of Tse and Zdancewic may be required for justifying this use of their technique. In particular, although their encoding is type-preserving, typing in System F is somewhat more liberal than in DCC, and one would want to ensure that the flexibility of System F does not yield unexpected results for access control.

Moreover, while encodings into System F can provide useful guidance and semantics, it would be premature to replace DCC with System F. At this point, it seems more convenient to reason about access control in DCC than in System F. Unlike System F, Simply Typed DCC provides a simple notation and setting in which to study logical questions, to develop examples and applications, and to experiment with alternative concepts and rules. Although Polymorphic DCC includes all the

complexity of System F, it still shares many of the advantages of Simply Typed DCC from our perspective, particularly since we make only a limited, disciplined use of quantifiers in this initial exploration.

6 Access Control in Polymorphic DCC

Continuing Section 4, in this section we suggest how to use Polymorphic DCC for access control.

6.1 "Speaks for"

We encode "speaks for" via polymorphism. In keeping with previous papers, we write $A \Rightarrow B$ for "A speaks for B". Here, however, we write $A \Rightarrow B$ as an abbreviation for

$$\forall X.\, (A\ \texttt{says}\ X \to B\ \texttt{says}\ X)$$

We immediately obtain a fundamental property of the "speaks for" relation:

$$\vdash (A \Rightarrow B) \to ((A\ \texttt{says}\ s) \to (B\ \texttt{says}\ s))$$

for every s, with proof term

$$\lambda x : A \Rightarrow B.\, xs$$

Remarkably, we also obtain the "hand-off axiom" as a theorem (not as an added axiom):

$$\vdash (A\ \texttt{says}\ (B \Rightarrow A)) \to (B \Rightarrow A)$$

This theorem can be derived because the definitions imply that $B \Rightarrow A$ is protected at level A. Its proof is the term:

$$\lambda x : A\ \texttt{says}\ (B \Rightarrow A).\, \texttt{bind}\ y = x\ \texttt{in}\ y$$

Clearly, $A \sqsubseteq B$ implies $A \Rightarrow B$:

$$\vdash A \Rightarrow B \qquad \text{for } A \sqsubseteq B$$

However, the converse is not true: $A \Rightarrow B$ does not imply $A \sqsubseteq B$. In fact, we can even have $A \Rightarrow B$ and $B \Rightarrow A$ while A and B are different lattice levels. In this respect, the logic keeps a certain separation between the underlying lattice and the partial order of the "speaks for" relation. Whether this separation is a feature or an unfortunate redundancy may be open to debate. It does not seem to have significant disadvantages, and it does have the advantage of allowing us to develop the Simply Typed calculus before having a "speaks for" relation, without deviating from DCC. It also enables the use of the lattice operations $\sqcap$ and $\sqcup$, which would be harder to define otherwise. Nevertheless, in Section 8 we consider a fragment of Polymorphic DCC that does not require a lattice of principals.

Restrictions of "speaks for", similar to ones previously considered (e.g., [17]), can be defined and studied just as easily. In particular, given a type $\mathcal{C}(X)$ with free type variable X, we can write the type:

$$\forall X.\, (\mathcal{C}(X) \wedge A\ \texttt{says}\ X \to B\ \texttt{says}\ X)$$

in order to express that A speaks for B on statements X that satisfy $\mathcal{C}(X)$. For instance, when $\mathcal{C}(X)$ is $s \to X$ for some s, this type means that A speaks for B on consequences of s.

6.2 Examples

Using the encoding of "speaks for", we sketch some small examples of the use of Polymorphic DCC for access control. For these examples, as a convenience, we assume standard data types (such as `int`), as well as basic propositions for authorization. We also speculate on more substantial extensions to DCC.

Since DCC is a programming language, it is attractive to consider ways of integrating traditional programming with DCC-based access control, at the language level. The examples are partly intended to suggest avenues for research in this direction. Going further, it would be interesting to consider the combination of DCC-based access control with information-flow systems for programming languages (e.g., [23,25,26]), especially since DCC can capture notions of information flow.

6.2.1 Access with a Simple Hand-off

The first of our examples is fairly basic. It could be reproduced in a variety of other logics once the appropriate instance of the "hand-off axiom" is assumed.

Suppose that we have the following formulas:

$$\begin{array}{ll} & A \ \mathtt{says}\ (B \Rightarrow A) \\ (3) & B \ \mathtt{says}\ \mathrm{Do}(o) \\ & (A \ \mathtt{says}\ \mathrm{Do}(o)) \to \mathrm{Do}(o) \end{array}$$

The first represents a hand-off from A to B. As in Section 4.1, the second formula shows a statement $\mathrm{Do}(o)$ supported by B's authority, and the third reflects that A is trusted on that statement.

Combining these formulas, we can derive $\mathrm{Do}(o)$. Formally, we have:

$$\vdash \left(\begin{array}{l} A \ \mathtt{says}\ (B \Rightarrow A) \\ \quad \wedge \\ B \ \mathtt{says}\ \mathrm{Do}(o) \\ \quad \wedge \\ (A \ \mathtt{says}\ \mathrm{Do}(o)) \to \mathrm{Do}(o) \end{array} \right) \to \mathrm{Do}(o)$$

As in Section 4.1, the conclusion $\mathrm{Do}(o)$ is intended to mean that operation o should be granted.

We assume the formulas (3) for the rest of Section 6, as we build on this example.

6.2.2 Proof-Carrying Calls

Next we consider the situation in which we are defining a security-aware interface for the operation o. This interface should reflect o's authorization requirements. For instance, if o expects an integer argument and produces an integer result, then the type of o might be:

$$\mathrm{Do}(o) \rightarrow (\texttt{int} \rightarrow \texttt{int})$$

in which we make explicit the requirement $\mathrm{Do}(o)$. Thus, when a principal C invokes o, it should pass a proof of $\mathrm{Do}(o)$.

For this purpose, we would expect that the caller C would obtain and combine proofs of the formulas B `says` $\mathrm{Do}(o)$, $(A$ `says` $\mathrm{Do}(o)) \rightarrow \mathrm{Do}(o)$, and A `says` $(B \Rightarrow A)$. The proofs of A `says` $(B \Rightarrow A)$ and of B `says` $\mathrm{Do}(o)$ may be digitally signed statements, if we adopt the sensible principle that a digitally signed statement that can be checked with a principal's public key constitutes a proof that the principal says the statement. The proof of $(A$ `says` $\mathrm{Do}(o)) \rightarrow \mathrm{Do}(o)$ may be derived from a digitally signed statement from an authority trusted on o. We leave for further work the details of the interface between standard DCC and the world of digital signatures.

Fundamentally, this example requires viewing a proposition as a type and viewing a proof as a language-level expression. Hence, despite its simplicity, it seems to be beyond the scope of other logics for access control.

6.2.3 Proof Completion and Evaluation

When $(A$ `says` $\mathrm{Do}(o)) \rightarrow \mathrm{Do}(o)$ represents the access policy for o as specified by an authority trusted on o, the reference monitor may be willing to assume this formula, but the caller C may not have a proof of $(A$ `says` $\mathrm{Do}(o)) \rightarrow \mathrm{Do}(o)$. In such situations, a more permissive type for o may be appropriate. The type might be:

$$(((A \texttt{ says } \mathrm{Do}(o)) \rightarrow \mathrm{Do}(o)) \rightarrow \mathrm{Do}(o)) \rightarrow (\texttt{int} \rightarrow \texttt{int})$$

or simply:

$$(A \texttt{ says } \mathrm{Do}(o)) \rightarrow (\texttt{int} \rightarrow \texttt{int})$$

According to this type, C should provide a proof of

$$A \texttt{ says } \mathrm{Do}(o)$$

The reference monitor can then establish $\mathrm{Do}(o)$ for itself, by applying

$$(A \texttt{ says } \mathrm{Do}(o)) \rightarrow \mathrm{Do}(o)$$

Similarly, there are situations in which the evidence for B `says` $\mathrm{Do}(o)$ cannot be reified into a proof that the caller C can present and combine with other proofs. For instance, when C is actually B, the act of invoking o may include asserting $\mathrm{Do}(o)$. The reference monitor may know what B says, but the evidence in question is not a bitstring or an expression that can be transferred. Again, a more permissive type for o may be appropriate. The type might be:

$$((B \texttt{ says } \mathrm{Do}(o)) \rightarrow \mathrm{Do}(o)) \rightarrow (\texttt{int} \rightarrow \texttt{int})$$

According to this type, C should provide a proof of

$$(B \texttt{ says } \mathrm{Do}(o)) \to \mathrm{Do}(o)$$

The reference monitor can then try to establish $\mathrm{Do}(o)$ for itself, in this case by proving B `says` $\mathrm{Do}(o)$. In particular, the reference monitor may simply assume B `says` $\mathrm{Do}(o)$ if it has authenticated the caller as B. For other callers, the reference monitor may fail in its proof of B `says` $\mathrm{Do}(o)$, at run-time. A richer type system might express that, if the caller speaks for B, then the reference monitor is guaranteed to grant access.

Such variations lead to the idea that, sometimes, the caller C should be allowed to provide only an incomplete proof of $\mathrm{Do}(o)$. The incomplete proof may contain commands that indicate where the reference monitor is expected to do theorem-proving, consult its local policy, or solicit additional evidence from C or from other sources. Those commands go beyond what we have studied formally in this paper; they may for example include input, output, and recursion. They are not guaranteed to succeed. For instance, recursion may allow circular reasoning, and requests for input may not be answered. Therefore, evidence should be evaluated, and the evaluation process should turn incomplete proofs into complete proofs. The resulting system leverages the idea that proofs are programs that can be executed, and thus further exploits the Curry-Howard isomorphism.

7 Metatheory

In this section we start to explore the metatheory of Polymorphic DCC.

7.1 Basic Results

The metatheory of Polymorphic DCC ought to include traditional consistency, type soundness, and parametricity results, of the kind common in the study of System F. In this paper, we start to consider those properties, relying on the metatheory of System F. More broadly, we expect many existing theorems and techniques to apply to the study of Polymorphic DCC. In this respect, it is useful that DCC and System F are not ad hoc formal systems defined solely for the purposes of reasoning about access control.

Without much effort, we can obtain some basic results by translating Polymorphic DCC into System F (specifically, into a version of System F with unit, product, sum, and function types, as well as universal quantification), even with an encoding less sophisticated than the one introduced by Tse and Zdancewic (see Section 5.3). In particular, our encoding may simply delete all occurrences of `says`. Writing $(\cdot)^{\mathrm{F}}$ for this encoding, we have that

$$(A \texttt{ says } s)^{\mathrm{F}} = (s)^{\mathrm{F}}$$

The encoding is trivial for `true`, $\vee$, $\wedge$, $\to$, $\forall$, and type variables.

We extend the function $(\cdot)^{\mathrm{F}}$ to environments, applying it component by component. Note that if s is well-formed in Γ then $(s)^{\mathrm{F}}$ is well-formed in $(\Gamma)^{\mathrm{F}}$.

The encoding maps typed terms to typed terms:

Proposition 7.1 *If* $\Gamma \vdash e : t$ *in Polymorphic DCC, then there exists* e' *such that* $(\Gamma)^{\mathrm{F}} \vdash e' : (t)^{\mathrm{F}}$ *in System F.*

Proof. We construct e' by translating e, by induction on typing derivations in Polymorphic DCC. The only non-trivial cases correspond to the rules [*UnitM*] and [*BindM*]. For those, we set:

$$(\eta_A\ e)^{\mathrm{F}} = (e)^{\mathrm{F}}$$
$$(\texttt{bind}\ x = e\ \texttt{in}\ e')^{\mathrm{F}} = ((\lambda x : t.\ (e')^{\mathrm{F}})(e)^{\mathrm{F}})$$

for an appropriate type t obtained from the typing derivation for $\texttt{bind}\ x = e\ \texttt{in}\ e'$. Specifically, if the typing of $\texttt{bind}\ x = e\ \texttt{in}\ e'$ is done in an environment Γ' and relies on the hypothesis $\Gamma' \vdash e : A\ \texttt{says}\ s$, then t is $(s)^{\mathrm{F}}$. □

It follows that Polymorphic DCC is consistent:

Proposition 7.2 *In Polymorphic DCC, it is not the case that* $\vdash \forall X.\ X$*, nor that* $\vdash A\ \texttt{says}\ \forall X.\ X$*.*

Proof. If it were the case, we would have a list of type variables $X_1, \ldots, X_n$ and a term e such that $X_1, \ldots, X_n \vdash e : \forall X.\ X$ or $X_1, \ldots, X_n \vdash e : A\ \texttt{says}\ \forall X.\ X$ in Polymorphic DCC. By Proposition 7.1, we would then have $X_1, \ldots, X_n \vdash e' : \forall X. X$ for some term e' in System F. Since $\forall X.\ X$ is not inhabited in System F, we obtain a contradiction. □

It may also be seen that the encoding maps reduction steps to reduction steps. It follows that infinite reduction sequences are not possible in Polymorphic DCC, since they are not in System F.

7.2 Noninterference

The metatheory of Polymorphic DCC should also include noninterference results in the style of previous ones for DCC [3,27] or those of Garg and Pfenning [11]. A typical noninterference result would imply that if we have a proof e of $A\ \texttt{says}\ s$ and it depends on a proof x of $B\ \texttt{says}\ t$, where A and B are unrelated levels, then, from the point of view of e, it does not matter which actual proof we substitute for x. Even more strongly, we should be able to obtain that e can be constructed without x (at least under certain hypotheses on e). We devote the rest of this section to establishing a result in this spirit.

We state a basic proposition as a preliminary:

Proposition 7.3 *In Polymorphic DCC, if* t *is protected at level* A *and* $B \sqsubseteq A$*, then* t *is protected at level* B*.*

Proof. The argument is by induction on the proof that t is protected at level A. The base case for a formula of the form $C\ \texttt{says}\ s$ with $A \sqsubseteq C$ follows from the transitivity of $\sqsubseteq$, which implies that $B \sqsubseteq C$. The base case for $\texttt{true}$ is trivial. The inductive steps are straightforward applications of the induction hypothesis. □

For a type s and $B \in \mathcal{L}$, we define $(s)^B$ as follows:

$$
\begin{aligned}
(\texttt{true})^B &= \texttt{true} \\
(s_1 \vee s_2)^B &= (s_1)^B \vee (s_2)^B \\
(s_1 \wedge s_2)^B &= (s_1)^B \wedge (s_2)^B \\
(s_1 \to s_2)^B &= (s_1)^B \to (s_2)^B \\
(A\ \texttt{says}\ s)^B &= \begin{cases} \texttt{true} & \text{if } B \sqsubseteq A \\ A\ \texttt{says}\ (s)^B & \text{otherwise} \end{cases} \\
(X)^B &= X \\
(\forall X.\ s)^B &= \forall X.\ (s)^B
\end{aligned}
$$

Note that type variables are left unchanged by the function $(\cdot)^B$. While a type variable may eventually be instantiated to a type of the form A `says` s, and we cannot predict whether $B \sqsubseteq A$, we need not be concerned with those instantiations for our purposes. The function $(\cdot)^B$ commutes with substitutions for type variables: $(s[t/X])^B = (s)^B[(t)^B/X]$.

Intuitively, $(s)^B$ is a variant of s that corresponds to the situation in which B is completely untrustworthy, so B `says` t is always true, independently of t. Theorem 7.6, below, aims to show that B's untrustworthiness has a limited effect on other principals.

If s is of the form B `says` t, then $(s)^B$ is trivially provable, since it is `true`. More generally, the following proposition shows that formulas of the form $(s)^B$ with s protected at level B are always provable (that is, that these formulas are always inhabited when viewed as types).

Proposition 7.4 *In Polymorphic DCC, if Γ is a well-formed typing environment, t is a well-formed type in Γ, and t is protected at level B, then there exists e such that $\Gamma \vdash e : (t)^B$.*

Proof. The argument is by induction on the proof that t is protected at level B.

- If t is of the form A `says` s where $B \sqsubseteq A$, then $(t)^B = \texttt{true}$. We take e to be ().
- If t is `true`, then again $(t)^B = \texttt{true}$ and we take e to be ().
- If t is of the form A `says` s where s is protected at level B, in this case with $B \not\sqsubseteq A$, then there exists e_1 such that $\Gamma \vdash e_1 : (s)^B$, by the induction hypothesis, so $\Gamma \vdash (\eta_A\ e_1) : A\ \texttt{says}\ (s)^B$, that is, $\Gamma \vdash (\eta_A\ e_1) : (t)^B$.

  (We can assume that $B \not\sqsubseteq A$ in this case, although this condition does not appear in the definition of "protected", because the first case applies when $B \sqsubseteq A$.)
- If t is of the form $(s_1 \wedge s_2)$ where s_1 and s_2 are protected at level B, then there exist e_1 and e_2 such that $\Gamma \vdash e_1 : (s_1)^B$ and $\Gamma \vdash e_2 : (s_2)^B$, by the induction hypothesis, so $\Gamma \vdash \langle e_1, e_2 \rangle : (t)^B$.
- If t is of the form $s_2 \to s_1$ where s_1 is protected at level B, then there exists e_1 such that $\Gamma, x : (s_2)^B \vdash e_1 : (s_1)^B$ for a fresh variable x, by the induction

hypothesis, and therefore $\Gamma \vdash (\lambda x : (s_2)^B.\, e_1) : (t)^B$.

- If t is of the form $\forall X.\, s$ where s is protected at level B, then there exists e_1 such that $\Gamma, X \vdash e_1 : (s)^B$ (up to renaming of X), by the induction hypothesis, and therefore $\Gamma \vdash (\Lambda X.\, e_1) : (t)^B$.

□

We extend the function $(\cdot)^B$ to environments, applying it component by component. Note that if t is well-formed in Γ then $(t)^B$ is well-formed in $(\Gamma)^B$. In addition, we have:

Proposition 7.5 *In Polymorphic DCC, if t is protected at level A then so is $(t)^B$.*

Proof. The argument is by induction on the proof that t is protected at level A.

- If t is of the form C `says` s where $A \sqsubseteq C$, then $(t)^B$ is either `true` or C `says` $(s)^B$, depending on whether $B \sqsubseteq C$. In the former case, it is immediate that $(t)^B$ is protected at level A; in the latter case, it follows from $A \sqsubseteq C$.
- If t is of the form C `says` s where s is protected at level A, then $(t)^B$ is either `true` or C `says` $(s)^B$, again depending on whether $B \sqsubseteq C$. In the former case, it is immediate that $(t)^B$ is protected at level A; in the latter case, it follows from the induction hypothesis.
- The remaining cases are straightforward applications of the definitions and of the induction hypothesis.

□

We obtain the following theorem:

Theorem 7.6 *In Polymorphic DCC, for every typing environment Γ, type s, and $B \in \mathcal{L}$, if $\Gamma \vdash e : s$ then there exists e' such that $(\Gamma)^B \vdash e' : (s)^B$.*

Proof. The proof is by induction on the derivation of $\Gamma \vdash e : s$, with an argument by cases on the last rule applied in this derivation.

- [*Var*]: e' is the same variable as e.
- [*Unit*]: e' is $()$, like e.
- [*Lam*]: If e is $\lambda x : s_1.\, e_1$ then e' is $\lambda x : (s_1)^B.\, e_1'$ where e_1' is given by the induction hypothesis.
- [*App*]: If e is $e_1 e_2$ then e' is $e_1' e_2'$ where e_1' and e_2' are given by the induction hypothesis.
- [*Pair*], [*Proj 1*], [*Proj 2*], [*Inj 1*], [*Inj 2*], [*Case*]: All these cases are also straightforward applications of the induction hypothesis.
- [*UnitM*]: If e is $(\eta_A\ e_1)$ then e' is $()$ if $B \sqsubseteq A$, and otherwise it is $(\eta_A\ e_1')$ where e_1' is given by the induction hypothesis.
- [*BindM*]: If s is protected at level B, then Proposition 7.4 yields the desired result. On the other hand, if s is not protected at level B, then it is not protected at any level A such that $B \sqsubseteq A$, by Proposition 7.3. So, if rule [*BindM*] is

applied with hypotheses of the form $\Gamma \vdash e_1 : A \texttt{ says } s_1$ and $\Gamma, x : s_1 \vdash e_2 : s$, and the condition that s is protected at level A, then $B \not\sqsubseteq A$. It follows that $(A \texttt{ says } s_1)^B = A \texttt{ says } (s_1)^B$. We can therefore let e' be $\texttt{bind } x = e'_1 \texttt{ in } e'_2$, where e'_1 and e'_2 are given by the induction hypothesis. The typing of e' can be done via rule $[BindM]$ because of the form of $(A \texttt{ says } s_1)^B$ and because $(s)^B$ is protected at level A by Proposition 7.5.

- $[TLam]$: If e is $\Lambda X.\, e_1$ then e' is $\Lambda X.\, e'_1$ where e'_1 is given by the induction hypothesis.
- $[TApp]$: If e is $e_1 t_1$ then e' is $e'_1 (t_1)^B$ where e'_1 is given by the induction hypothesis. The typing of e' can be done via rule $[TApp]$ because of the commutation of $(\cdot)^B$ with substitution: the type of e_1 must be of the form $\forall X.\, t$, with $s = t[t_1/X]$ and $(t[t_1/X])^B = (t)^B[(t_1)^B/X]$.

(The statement of the theorem does not concern itself with characterizing the behavior of e' as a program. An analysis of the proof could give such additional information.) □

As a special case, we derive the following corollary from the theorem:

Corollary 7.7 *In Polymorphic DCC, for every type s and $B \in \mathcal{L}$, if $\vdash s$ then $\vdash (s)^B$.*

For example, Corollary 7.7 says that if $B \not\sqsubseteq A$, then

$$\vdash (B \texttt{ says } t) \to (A \texttt{ says } \forall X.\, X)$$

implies

$$\vdash \texttt{true} \to (A \texttt{ says } \forall X.\, X)$$

and therefore

$$\vdash A \texttt{ says } \forall X.\, X$$

Since this judgment is not derivable (by Proposition 7.2), we arrive at a contradiction, inferring that

$$\vdash (B \texttt{ says } t) \to (A \texttt{ says } \forall X.\, X)$$

is not derivable either. Thus, no matter what B says, A does not say $\forall X.\, X$.

More generally, suppose that $(s)^B = s$. This condition holds when s mentions no principal C such that $B \sqsubseteq C$. Corollary 7.7 says that if $B \not\sqsubseteq A$ then

$$\vdash (B \texttt{ says } t) \to (A \texttt{ says } s)$$

implies

$$\vdash \texttt{true} \to (A \texttt{ says } s)$$

and therefore

$$\vdash A \texttt{ says } s$$

In this case, the conclusion $\vdash A \texttt{ says } s$ may be derivable, in particular when s is a tautology such as $X \to X$. Thus, if A says s when B says t, then A says s whether or not B says t.

On the other hand, suppose that $(s)^B = s$ need not hold. Corollary 7.7 yields that if $B \not\sqsubseteq A$ then

$$\vdash (B \texttt{ says } t) \to (A \texttt{ says } s)$$

implies

$$\vdash A \texttt{ says } (s)^B$$

We cannot expect to obtain $A \texttt{ says } s$ verbatim. Consider, for instance, the case in which s is $B \texttt{ says } t$. We have that

$$\vdash (B \texttt{ says } t) \to (A \texttt{ says } (B \texttt{ says } t))$$

but in general we cannot derive $A \texttt{ says } (B \texttt{ says } t)$ without any hypotheses. Thus, our results limit the effect that B's statements can have on A's statements, but we cannot expect to show that they have no effect at all: what B says does influence what A says about B.

Corollary 7.7 applies even if B speaks for A because of a hand-off, but in that case it does not yield interesting results. Suppose, again, that $B \not\sqsubseteq A$; then

$$\vdash (A \texttt{ says } (B \Rightarrow A)) \to ((B \texttt{ says } t) \to (A \texttt{ says } X))$$

implies

$$\vdash (A \texttt{ says } \forall Y.\, (\texttt{true} \to A \texttt{ says } Y)) \to (\texttt{true} \to (A \texttt{ says } X))$$

and therefore

$$\vdash (A \texttt{ says } \forall Y.\, A \texttt{ says } Y) \to (A \texttt{ says } X)$$

However, this conclusion is fairly obvious.

8 A Fragment of Polymorphic DCC

We close the technical body of this paper by presenting an attractive fragment of Polymorphic DCC that we call Polymorphic CDD ("Cut-Down DCC"). This fragment shares many of the properties of Polymorphic DCC, but it is simpler. In particular, it does not require a notion of protected type, nor a lattice structure on principals. In these respects, it is a little closer to the computational lambda calculus [22] and to Garg and Pfenning's logic [11]. However, it still supports a "speaks for" relation with a hand-off property.

8.1 The Calculus

Types are given by the grammar:

$$s ::= \texttt{true} \mid (s \vee s) \mid (s \wedge s) \mid (s \to s) \mid A \texttt{ says } s \mid X \mid \forall X.\, s$$

much as in Section 5.1.1, but here A ranges over elements of a set, rather than a lattice.

The terms and the typing rules are exactly as in Table 1, except that rule [*BindM*] is simplified (and thereby limited) so that it does not have a side condition

about "protected types" and instead it yields conclusions that always have the form A `says` t:

$$\frac{\Gamma \vdash e : A \texttt{ says } s \qquad \Gamma, x : s \vdash e' : A \texttt{ says } t}{\Gamma \vdash \texttt{bind } x = e \texttt{ in } e' : A \texttt{ says } t}$$

In logical form, this rule is:

$$\frac{\Gamma \vdash A \texttt{ says } s \qquad \Gamma, s \vdash A \texttt{ says } t}{\Gamma \vdash A \texttt{ says } t}$$

8.2 Some Theorems

By design, the modified rule [*BindM*] does not enable us to type-check the term $\lambda x : A \texttt{ says } B \texttt{ says } s.\, \texttt{bind } y = x \texttt{ in bind } z = y \texttt{ in } (\eta_B\ (\eta_A\ z))$ and to derive the commutativity property $(A \texttt{ says } B \texttt{ says } s) \to (B \texttt{ says } A \texttt{ says } s)$ (cf. Section 4.2). Other than this, Polymorphic CDD yields most of the same important theorems as Polymorphic DCC. In particular, the operators $\vee$, $\wedge$, and $\to$ obey the standard intuitionistic propositional rules, and in addition we have:

$$\vdash s \to A \texttt{ says } s$$

$$\vdash (A \texttt{ says } (s \to s')) \to ((A \texttt{ says } s) \to (A \texttt{ says } s'))$$

$$\vdash (A \texttt{ says } A \texttt{ says } s) \to (A \texttt{ says } s)$$

$$\vdash (A \texttt{ says } \forall X.\, s) \to (\forall X.\, A \texttt{ says } s)$$

$$\vdash (A \Rightarrow B) \to ((A \texttt{ says } s) \to (B \texttt{ says } s))$$

with the same proofs as above. We also obtain the "hand-off axiom":

$$\vdash (A \texttt{ says } (B \Rightarrow A)) \to (B \Rightarrow A)$$

with the proof term

$$\lambda x : A \texttt{ says } (B \Rightarrow A).\, \Lambda X.\, \lambda y : B \texttt{ says } X.\, \texttt{bind } z = x \texttt{ in } (zX)\ y$$

which is more complex than the corresponding term in Section 6.

Obviously, we cannot obtain any of the results related to the lattice $\mathcal{L}$. However, we can obtain some analogues by representing every relation $A \sqsubseteq B$ in $\mathcal{L}$ as an assumption $A \Rightarrow B$. It is not too hard to turn this idea into a precise encoding when $\mathcal{L}$ is finite. More work on the proof theory of Polymorphic DCC is required for treating the general case in which $\mathcal{L}$ may be infinite.

Like Polymorphic DCC, Polymorphic CDD is consistent: Proposition 7.2 applies immediately. Other results from the metatheory of Polymorphic DCC carry over as well. In particular, Theorem 7.6 has a straightforward analogue, for which we let $(B \texttt{ says } s)^B = \texttt{true}$ for every principal B and $(A \texttt{ says } s)^B = A \texttt{ says } (s)^B$ for every other principal A.

9 Conclusion

While the body of this paper suggests several attractive directions for further research, the main questions on this enterprise may concern the usefulness of the perspective on access control that DCC suggests. Should we use DCC (or a similar type system) as a logic for access control?

In particular, we may have doubts on the logical rules that DCC induces. Specifically, the "hand-off axiom" embodies a strong form of delegation of authority, which may seem dangerous. On the other hand, there are safe disciplines for applying the "hand-off axiom", in which it is used for justifying only more controlled forms of delegation. For example, principals may limit their authority by adopting roles before delegating [16, Section 6.1].

In addition, we may wonder whether the constructive character of DCC will not occasionally surprise and frustrate. On the other hand, regarding the logic of access control as a type system for evidence, perhaps this constructive character should be embraced.

Overall, DCC satisfies several long-standing wishes as a calculus for access control. That appears to be more than a coincidence.

Acknowledgments

This work was done at Microsoft Research. It benefited from conversations with Cédric Fournet, Butler Lampson, Dave Langworthy, Greg Morrisett, and Frank Pfenning, and from comments from anonymous reviewers.

References

[1] Martín Abadi. Logic in access control. In *Proceedings of the Eighteenth Annual IEEE Symposium on Logic in Computer Science*, pages 228–233, 2003.

[2] Martín Abadi. Access control in a core calculus of dependency. In *Proceedings of the 11th ACM International Conference on Functional Programming (ICFP 2006)*, 2006. To appear.

[3] Martín Abadi, Anindya Banerjee, Nevin Heintze, and Jon G. Riecke. A core calculus of dependency. In *Proceedings of the 26th ACM Symposium on Principles of Programming Languages*, pages 147–160, January 1999.

[4] Martín Abadi, Michael Burrows, Butler Lampson, and Gordon Plotkin. A calculus for access control in distributed systems. *ACM Transactions on Programming Languages and Systems*, 15(4):706–734, October 1993.

[5] Andrew W. Appel and Edward W. Felten. Proof-carrying authentication. In *Proceedings of the 5th ACM Conference on Computer and Communications Security*, pages 52–62, November 1999.

[6] Lujo Bauer, Scott Garriss, and Michael K. Reiter. Distributed proving in access-control systems. In *Proceedings of the 2005 IEEE Symposium on Security and Privacy*, pages 81–95, May 2005.

[7] Luca Cardelli. Type systems. In Allen B. Tucker, editor, *The Computer Science and Engineering Handbook*, chapter 103, pages 2208–2236. CRC Press, Boca Raton, FL, 1997.

[8] Dorothy E. Denning. *Cryptography and Data Security*. Addison-Wesley, Reading, Mass., 1982.

[9] John DeTreville. Binder, a logic-based security language. In *Proceedings of the 2002 IEEE Symposium on Security and Privacy*, pages 105–113, May 2002.

[10] Matt Fairtlough and Michael Mendler. Propositional lax logic. *Information and Computation*, 137(1):1–33, 1997.

[11] Deepak Garg and Frank Pfenning. Non-interference in constructive authorization logic. A revised version of this paper appears in the *Proceedings of the 19th IEEE Computer Security Foundations Workshop (CSFW 19)*. Manuscript, February 2006.

[12] Morrie Gasser. *Building a Secure Computer System*. Van Nostrand Reinhold Company Inc., New York, 1988.

[13] Jean-Yves Girard. *Interprétation Fonctionnelle et Elimination des Coupures de l'Arithmétique d'Ordre Supérieur*. Thèse de doctorat d'état, Université Paris VII, June 1972.

[14] Paul Hudak, Simon Peyton Jones, Philip Wadler, Brian Boutel, Jon Fairbairn, Joseph Fasel, María M. Guzmán, Kevin Hammond, John Hughes, Thomas Johnsson, Dick Kieburtz, Rishiyur Nikhil, Will Partain, and John Peterson. Report on the programming language Haskell: a non-strict, purely functional language. Version 1.2. *ACM SIGPLAN Notices*, 27(5):1–164, 1992.

[15] Trevor Jim. SD3: A trust management system with certified evaluation. In *Proceedings of the 2001 IEEE Symposium on Security and Privacy*, pages 106–115, May 2001.

[16] Butler Lampson, Martín Abadi, Michael Burrows, and Edward Wobber. Authentication in distributed systems: Theory and practice. *ACM Transactions on Computer Systems*, 10(4):265–310, November 1992.

[17] Butler W. Lampson. Computer security in the real world. *IEEE Computer*, 37(6):37–46, June 2004.

[18] David Langworthy. Private communication. February 2006.

[19] Ninghui Li, Benjamin N. Grosof, and Feigenbaum. Delegation logic: A logic-based approach to distributed authorization. *ACM Transactions on Information and System Security*, 6(1):128–171, February 2003.

[20] Ninghui Li and John C. Mitchell. Datalog with constraints: A foundation for trust-management languages. In *Proceedings of the Fifth International Symposium on Practical Aspects of Declarative Languages (PADL 2003)*, volume 2562 of *Lecture Notes in Computer Science*, pages 58–73. Springer-Verlag, January 2003.

[21] John C. Mitchell. *Foundations for Programming Languages*. The MIT Press, Cambridge, Mass., 1996.

[22] Eugenio Moggi. Notions of computation and monads. *Information and Control*, 93(1):55–92, 1991.

[23] Andrew C. Myers. JFlow: Practical mostly-static information flow control. In *Proceedings of the 26th ACM Symposium on Principles of Programming Languages*, pages 228–241, January 1999.

[24] Gordon D. Plotkin. Call-by-name, call-by-value and the λ-calculus. *Theoretical Computer Science*, 1:125–159, 1975.

[25] François Pottier and Sylvain Conchon. Information flow inference for free. In *Proceedings of the Fifth ACM SIGPLAN International Conference on Functional Programming*, pages 46–57, September 2000.

[26] François Pottier and Vincent Simonet. Information flow inference for ML. *ACM Transactions on Programming Languages and Systems*, 25(1):117–158, January 2003.

[27] Stephen Tse and Steve Zdancewic. Translating dependency into parametricity. *Journal of Functional Programming*. To appear.

[28] Philip Wadler. The marriage of effects and monads. In *Proceedings of the 3rd ACM SIGPLAN International Conference on Functional Programming*, pages 63–74, 1998.

[29] Dan S. Wallach, Andrew W. Appel, and Edward W. Felten. SAFKASI: A security mechanism for language-based systems. *ACM Transactions on Software Engineering and Methodology*, 9(4):341–378, 2000.

[30] Edward Wobber, Martín Abadi, Michael Burrows, and Butler Lampson. Authentication in the Taos operating system. *ACM Transactions on Computer Systems*, 12(1):3–32, February 1994.

[31] Steve Zdancewic. Private communication. August 2006.