Security-Typed Programming within Dependently Typed Programming

Jamie Morgenstern* Daniel R. Licata *
Carnegie Mellon University
{jamiemmt,drl}@cs.cmu.edu

Abstract

Several recent security-typed programming languages, such as Aura, PCML5, and Fine, allow programmers to express and enforce access control and information flow policies. In this paper, we show that security-typed programming can be embedded as a library within a general-purpose dependently typed programming language, Agda. Our library, Aglet, accounts for the major features of existing security-typed programming languages, such as decentralized access control, typed proof-carrying authorization, ephemeral and dynamic policies, authentication, spatial distribution, and information flow. The implementation of Aglet consists of the following ingredients: First, we represent the syntax and proofs of an authorization logic, Garg and Pfenning's BL_0, using dependent types. Second, we implement a proof search procedure, based on a focused sequent calculus, to ease the burden of constructing proofs. Third, we represent computations using a monad indexed by pre- and post-conditions drawn from the authorization logic, which permits ephemeral policies that change during execution. We describe the implementation of our library and illustrate its use on a number of the benchmark examples considered in the literature.

Categories and Subject Descriptors F.3.3 [*Logics and Meanings Of Programs*]: Studies of Program Constructs—Type structure; F.3.1 [*Logics and Meanings Of Programs*]: Specifying and Verifying and Reasoning about Programs

1. Introduction

Security-typed programming languages allow programmers to specify and enforce security policies, which describe both *access control*—who is permitted to access sensitive resources?—and *information flow*—what are they permitted to do with these resources once they get them? Aura [24] and PCML5 [9] enforce access control using dependently typed proof-carrying authorization (PCA): the run-time system requires every access to a sensitive resource

* This research was sponsored in part by the National Science Foundation under grants CCF-0702381 and CNS-0716469, and by the Pradeep Sindhu Computer Science Fellowship. The views and conclusions contained in this document are those of the author and should not be interpreted as representing the official policies, either expressed or implied, of any sponsoring institution, the U.S. government or any other entity.

ICFP'10, September 27–29, 2010, Baltimore, Maryland, USA.
Copyright © 2010 ACM 978-1-60558-794-3/10/09…$5.00

be accompanied by a proof of authorization [7], while the type system aids programmers in constructing correct proofs. Fable [37] and Jif [14] enforce information flow properties using type systems that restrict the use of values that depend on private information. Fine [38] combines these techniques to enforce both. These languages' type systems employ a number of advanced techniques, such as dependently typed authorization proofs, indexed monads of computations at a place and on behalf of a principal [8], information flow types, and affine types for ephemeral security policies.

Dependently typed programming languages provide a rich language of type-level data and computation. One promising application of dependent types is constructing domain-specific type systems as libraries, rather than new language designs—this allows the language designer to exploit the implementation, metatheory, and tools of the host language. In this paper, we apply this methodology to security typed programming, and show that security-typed programming can be embedded within a general-purpose dependently typed programming language, Agda [32]. We implement a library, Aglet, which accounts for the major features of existing security-typed programming languages, such as Aura, PCML5, and Fine:

Decentralized Access Control: Access control policies are expressed as propositions in an *authorization logic*, Garg and Pfenning's BL_0 [21]. This permits *decentralized* access control policies, expressed as the aggregate of statements made by different principals about the resources they control. In our embedding, we represent BL_0's propositions and proofs using dependent types, and exploit Agda's type checker to validate the correctness of proofs.

Dependently Typed PCA: Primitives that access resources, such as file system operations, require programmers to provide a proof of authorization, which is guaranteed by the type system to be a well-formed proof of the correct proposition.

Ephemeral and Dynamic Policies: Whether or not one may access a resource is often dependent upon the state of a system. For example, in a conference management server, authors may submit a paper, but only before the submission deadline. Fine accounts for ephemeral policies using a technique called affine types, which requires a substructural notion of variables. Because Agda does not currently provide substructurality, we show that one can instead account for ephemeral policies using an indexed monad. Following Hoare Type Theory [31], we define a type $\bigcirc\ \Gamma\ A\ \Gamma'$, which represents a computation that, given precondition Γ, returns a value of type A, with postcondition Γ'. Here, Γ and Γ' are propositions from the authorization logic, describing the state of resources in the system. For example, consider the operation in a conference management server that closes submissions and begins reviewing. We represent this by a computation of type

```
○ (InPhase Submission) Unit (InPhase Reviewing)
```

Given the conference is in phase `Submission`, this computation returns a value of type `Unit`, and the state of the conference has been changed to `Reviewing`. For comparison between the approaches, we adapt Fine's conference management example to our indexed monad. Aglet also permits dynamic acquisition and generation of policies—e.g., generating a policy based on reading the state of the conference management server from a database on startup.

Authentication: Following previous work by Avijit and Harper [8], we model authentication with an indexed monad of computation on behalf of a principal, which tracks the currently authenticated user. This monad is equipped with a `sudo` operation for switching users, given appropriate credentials. We show that computation on behalf of a principal is a special case of our policy-indexed monad `○ Γ A Γ'`.

Spatial distribution: We also show that our policy-indexed monad can be used to model spatial distribution as in PCML5.

Information Flow: Information flow policies constrain the use of values based on what went into computing them, e.g. tainting user input to avoid SQL injection attacks. We represent information flow using well-established techniques, such as indexed monads [36] and applicative functors [38].

Compile-time and Run-time Theorem Proving: Dependently typed PCA admits a sliding scale between static and dynamic verification. At the static end, one can verify, at compile-time, that a program complies with a statically-given authorization policy. This verification consists of annotating each access to a resource with an authorization proof, whose correctness is ensured by type checking. However, in many programs, the policy is not known at compile time—e.g., the policy may depend upon a system's state. Such programs may dynamically test whether each operation is permitted before performing it, in which case dependently typed PCA ensures that the correct dynamic checks are made and that failure cases are handled. A program may also mix static and dynamic verification: for example, a program may dynamically check that an expected policy is in effect, and then, in the scope of that check, deduce consequences statically. Security-typed languages use theorem provers to reduce the burden of static proofs (as in Fine) and to implement dynamic checks (as in PCML5). We have implemented a certified theorem prover for BL_0, based on a focused sequent calculus. Our theorem prover can be run at compile-time and at run-time, fulfilling both of these roles. The theorem prover also saves programmers from having to understand the details of the authorization logic, as they often do not need to write proofs manually.

The remainder of this paper is organized as follows: In Section 2, we show a variety of examples adapted from the literature, which demonstrate that Aglet accounts for programming in the style of Aura, PCML5, and Fine. In Section 3, we describe the implementation of Aglet, including the representation of the logic and the implementation of the theorem prover. We discuss related work in Section 4 and future work in Section 5. The Agda code for this paper is available from `http://www.cs.cmu.edu/~drl`.

2. Examples

In this section, we show that Aglet supports security-typed programming in the style of Aura, PCML5, and Fine by implementing a number of the benchmark examples considered in the literature. We briefly review Agda's syntax, referring the reader to the Agda Wiki(`wiki.portal.chalmers.se/agda/`). Dependent function types are written as `(x : A) → B`. An implicit dependent function space is written `{x : A} → B` or `∀ {x} → B` and arguments to implicit functions are inferred. Non-dependent functions are written `A → B`. Anonymous functions are written `λ x → e`. Named functions are defined clausally by pattern matching. Lists are constructed by `[]` and `::` (note that `:` is used for type annotations). `Set` is the classifier of classifiers in Agda.

$$
\begin{array}{l}
\mathsf{Admin\ says}\ (\forall r.\forall o.\forall f.\\
\qquad (\mathsf{HR\ says\ employee}(r)\\
\qquad \wedge\ \mathsf{System\ says\ owns}(o, f)\\
\qquad \wedge\ o\ \mathsf{says\ mayread}(r, f))\\
\quad\ \supset\ \mathsf{mayread}(r, f))\\
\mathsf{System\ says\ owns(Jamie, secret.txt)}\\
\mathsf{HR\ says\ employee(Dan)}\\
\mathsf{HR\ says\ employee(Jamie)}\\
\mathsf{Jamie\ says\ mayread(Dan, secret.txt)}\\
\mathsf{Jamie\ says\ mayread(Jamie, secret.txt)}
\end{array}
$$

Figure 1. Sample access control policy

2.1 File IO with Access Control

First, we show a dependently typed file system interface, a standard example of security typed programming [8, 38, 39].

2.1.1 Policy

To begin, we specify an authorization policy for file system operations in BL_0 (Figure 1): First, the principal Admin says that for any reader, owner, and file, if human resources says the reader is an employee, and the system administrator says the owner owns the file, and the owner says the reader may read a file, then the reader may read the file. Admin is a distinguished principal whose statements will be used to govern file system operations. Second, the system administrator says Jamie owns secret.txt. Third, human resources says both Dan and Jamie are employees. Fourth, Jamie says Dan and Jamie may read the file. This policy illustrates decentralized access control using the says modality: the policy is the aggregate of statements by different principals about resources they control.

For the principal Dan to read secret.txt, it will be sufficient to deduce the goal $\mathsf{Admin}\ \mathsf{says}\ \mathsf{mayread}(\mathsf{Dan}, \mathsf{secret.txt})$. This proposition is provable from the above policy because of three properties of says: First, says is closed under instantiation of universal quantifiers (that is, $k\ \mathsf{says}\ \forall x.A(x)$ entails $\forall x.k\ \mathsf{says}\ A(x)$). Second, says distributes over implications ($k\ \mathsf{says}\ (A \supset B)$ entails $((k\ \mathsf{says}\ A) \supset (k\ \mathsf{says}\ B))$. Third, every principal believes that every statement of every other principal has been made ($k\ \mathsf{says}\ A$ entails $k'\ \mathsf{says}\ (k\ \mathsf{says}\ A)$)—though it is not the case that every principal believes that every statement of every other principal is *true*. Thus, the goal can be proved by using the first clause of the policy ($\mathsf{Admin}\ \mathsf{says}\ \ldots$), instantiating the quantifiers, and using the other statements in the policy to satisfy the preconditions.

In Agda, we represent this first clause as the first element of the following context (list of propositions):

```
Γpolicy =
 (Prin "Admin" says
  (∀e principal · ∀e principal · ∀e filename ·
      let owner = ▷ (iS (iS i0))
          reader = ▷ (iS i0)
          file = ▷ i0 in
   ( (  (Prin "HR"  says  (a- (Employee · reader)))
      ∧ (Prin "System" says (a- (Owner · (owner , file))))
      ∧ (owner says (a- (Mayread · (reader , file)))))
    ⊃
     (a- (Mayread · (reader , file)))))  ::
 (Prin "Admin" says
  (∀e principal ·
   ∀e filename ·
    (Prin "System" says (a- (Owner · (▷ iS i0 , ▷ i0))))
   ⊃
    (a- (MayChown · (▷ iS i0 , ▷ i0)))))  ::
 []
```

The second element of the list expresses an additional policy clause, not discussed above, which states that an owner of a file may change its ownership. Variables are represented as de Bruijn indices (i0, iS), constants are represented as injections of strings (Prin "Admin"), and atomic propositions are tagged with a *polarity* (a+ or a-), which can be thought of as a hint to the theorem prover. Quantifiers are written ∀e τ · A, where τ is the domain of quantification and A is the body of the quantifier. Atomic propositions are written p · t, where p is a proposition constant such as Mayread and t is a term (see Section 3.1 for details).

Next, we define a context representing a particular file system state. This context includes all the employee, ownership, and mayread facts mentioned above, with one additional clause saying that Dan may su as Jamie.

```
Γstate =
    (Prin "System" says
       (a- (Owner · (Prin "Jamie" , File "secret·txt"))))
  :: (Prin "HR" says  (a- (Employee · (Prin "Dan"))))
  :: (Prin "HR" says  (a- (Employee · (Prin "Jamie"))))
  :: (Prin "Jamie"  says
       (a-(Mayread · (Prin "Dan" , File "secret·txt"))))
  :: (Prin "Jamie"  says
       (a-(Mayread · (Prin "Jamie" , File "secret·txt"))))
  :: (Prin "Admin" says
       (a- (MaySu · (Prin "Dan" , Prin "Jamie"))))
  :: []
Γall = Γpolicy ++ Γstate
```

Finally, we let Γall stand for the append of Γpolicy and Γstate.

2.1.2 Compile-time Theorem Proving

We now explain the use of our theorem prover:

```
goal = a-(Mayread · (Prin "Dan" , File "secret·txt"))

proof? : Maybe (Proof Γall goal)
proof?  = prove 15

theProof : Proof Γall goal
theProof = solve proof?
```

The term proof? sets up a call to the theorem prover, attempting to prove mayread(Dan, secret.txt) using the policy specified by Γall. Sequent calculus proofs are represented by an Agda type family $(\Omega\ ;\ \Delta\ ;\ \Gamma\ ;\ \mathtt{k}) \vdash \mathtt{A}$, where Ω binds individual variables, Δ is a context of *claims* assumptions, Γ is context of *truth* assumptions, and k, the *view*, is a principal from whose point of view the judgement is made. Informally, the role of the view is that, in a sequent whose view is `k`, `k says A` entails `A`; see Section 3.1 for details about the logic. In this example, Ω and Δ will always be empty, Γ will represent a policy, as above, and the view `k` will be `Prin "Admin"`—we abbreviate such a sequent by `Proof Γ A`. The context and proposition arguments to `prove` can be inferred by Agda, and so are left as implicit arguments. The term `theProof` checks that the theorem prover succeeds *at compile-time* in this instance. The function `solve` has type:

```
solve : ∀ {A} (s : Maybe A) → {p : Check (isSome s)} → A
```

The argument `p`, of type `Check (isSome s)`, is a proof that `s` is equal to `Some s'` for some `s'`. Because this argument is implicit, Agda will attempt to fill it in by unification, which will succeed when `s` is definitionally equal to a term of the form `Some s'`. In this example, the call to the theorem prover in the term `proof?` proves the goal, computing definitionally to `Some s'` for a proof `s'` of mayread(Dan, secret.txt). Thus, we can use `solve` to extract this proof `s'`. In general, a call to the theorem prover on a context and a proposition that have no free Agda variables will always be equal to either `Some p` or to `None`.

Generic operations:

```
○ : TCtx+ [] → (A : Set) → (A → TCtx+ []) → Set

return : ∀ { Γ A} → A → ○  Γ A (\ _ → Γ)

_>>=_   : ∀ {A B Γ Γ' Γ''}
        → (○  Γ A Γ')
        → ((x : A)  →  ○ (Γ' x) B Γ'')
        → ○  Γ B Γ''

weakenPre : ∀ {A Γ Γ' Γn }
          → (Good Γn → Good Γ)
          → ◯ Γ A Γ' → Γ ⊆ Γn → ◯  Γn A Γ'

weakenPost : ∀ {A Γ Γ' Γn}
          → ◯ Γ A Γ'
          → ((x : A) → (Γn x ⊆ Γ' x))
          → ((x : A) -> (Good (Γ' x) → Good (Γn x)))
          → ◯ Γ A Γn

getLine : ∀ {Γ} → ◯ Γ String (\ _ → Γ)

print : ∀ {Γ}  → String → ◯ Γ Unit (\ _ → Γ)

error : ∀ {A Γ Γ'} → String → ◯ Γ A Γ'

acquire : ∀ {A Γ Γ'} → (Γn : TCtx+ [])
        → (Good Γ → Good (Γn ++ Γ))
        → ◯ (Γn ++ Γ) A Γ' →  ◯ Γ A Γ'
        → ◯ Γ A Γ'
```

File-specific operations:

```
sudo : ∀ { Γ A Γ' Δ Δ'} → (k1 k2 : _)
 →  Replace (a+ (As · k1)) (a+ (As · k2)) Γ Δ
 → ((x : A) → Replace (a+ (As · k2)) (a+ (As · k1))
                        (Δ' x) (Γ' x))
 → (Proof Γ (a- (MaySu · (k1 ,  k2))))
 → ◯ Δ A Δ'
 → ◯ Γ A Γ'

read : ∀ {Γ} (k : _) (file : _)
      → Proof Γ (  (a- (Mayread · (k ,  file)))
                  ∧ (a+ (As · k)))
      → ◯ Γ String (λ _ → Γ)

create : ∀ {Γ} (k : _)
 → Proof Γ (   (a- (User · k))
             ∧ (a+ (As · k)))
 → ◯ Γ String
    (λ new → (Prin "System" says
                 (a-(Owner · (k , File new)))) :: Γ)

chown : ∀ { Γ Δ} → (k k1 k2 : _) → (f : _)
 → Replace (Prin "System" says (a-(Owner · (k1 , f))))
           (Prin "System" says (a-(Owner · (k2 , f))))
           Γ Δ
 → (Proof Γ (  (a+ (As · k))
             ∧ (a- (MayChown · (k , f)))))
 → ◯ Γ Unit (\ _ → Δ)
```

Figure 2. File IO with Authorization

2.1.3 Computations

We present a monadic interface for file operations in Figure 2. This figure shows both the generic IO operations, as well as three file-specific operations for reading, creating, and changing the owner of a file. The type ◯ Γ A Γ' represents a computation with precondition Γ and postcondition Γ'. The Agda type of a context is TCtx+ [] (a context of positive truth assumptions, with no free individual variables—see Section 3.1). The postcondition is a function from A's to contexts, so the postcondition may depend on the computation's result (see create below). The generic operations are typed as follows: Because return is not effectful, its postcondition is its precondition. Bind (>>=) chains together two computations, where the postcondition of the first is the precondition of the second. Both pre- and postconditions can be weakened to larger and smaller contexts, respectively; the Good predicate can be ignored until Section 2.1.4 below. Primitives like `getLine` (reading a line of input) and `print` do not change the state and do not require proofs. The postcondition of `error` is arbitrary, as it never terminates successfully. The remaining computations are defined as follows:

Read The function `read` takes a principal `k`, a file `f`, and a proof argument. The proof ensures that the principal `k` is authorized to access the file (`Mayread(k,f)`) and that the principal `k` is the currently authenticated user (`As(k)`). We use the proposition `As` to model computation on behalf of a principal [8]. The proof is checked in the context Γ that is the precondition of the computation, ensuring that it is valid in the current state of the world. `read` delivers the contents of the file and leaves the state unchanged.

An example call to `read` looks like this:

```
Γj = Γall as "Jamie"

jread : ○ Γj String (λ _ → Γj)
jread = read (Prin "Jamie") (File "secret·txt")
             (solve (prove 17))

jreadprint : ○ Γj Unit (λ _ → Γj)
jreadprint = jread >>= λ x →
             print ("the secret is: " ^ x)
```

The function call `Γall as k` is shorthand for adding the proposition `As(k)` to the context `Γall`. The computation `jread` reads the file `secret.txt` as principal `Jamie`; the proof argument is supplied by a call to the theorem prover, which statically verifies that the required fact is derivable from the policy given by `Γall`. The computation `jreadprint` reads the file and then prints the result.

Create The type of create is similar to `read`, in that it takes a principal and a proof that the principal can create a file (in this case, the fact that the principal is a registered user is deemed sufficient). It returns a String, the name of the created file, and illustrates why postconditions must be allowed to depend on the return value of the computation: the postcondition says that the principal is the owner of the newly created file. Thus, after a call to create(k), the postconditions signify System says Owner(k,f), where f is the name of the new file.

Chown To specify chown, we use a type Replace x y Γ Δ, which means that Δ is the result of replacing exactly one occurrence of x in Γ with y. Replace (whose definition is not shown) is defined by saying that (1) there is a de Bruijn index i showing that x is in Γ and (2) Δ is equal to the output of the function replace y i, which recurs on the index i and replaces the indicated element by y. The type of chown should be read as follows: if the principal k as whom the computation is running has the authority to change the owner of a file, and replacing owns(k,f) with owns(k’, f) in Γ produces Δ, then we can produce a computation which changes the owner of f from k to k’, leaving the remaining context unchanged.

Next, we show an example call to chown, using a context Γstate’ that is the result of replacing the fact that Jamie owns secret.txt with Dan owning that file. The computation dchown runs as Dan; it changes the owner of the file from Dan to Jamie, and then runs a computation drdprnt, defined below, that reads the file. proveReplace is a tactic used to prove that Γall’ is Γall with the ownership of secret.txt changed. solve (prove 15) calls the theorem prover to statically verify that Dan has permission to chown secret.txt.

```
Γstate’ =  replace {_} {Γstate}
  (Prin "System" says
    (a- (Owner · (Prin "Dan" , File "secret·txt"))))
```

```
Γall’ = Γpolicy ++ Γstate’

dchown : ○ (Γall’ as "Dan") Unit (λ _ → Γall as "Dan")
dchown = chown (Prin "Dan") (Prin "Dan") (Prin "Jamie")
           (File "secret·txt")
           (solve proveReplace) (solve (prove 15))
          >> drdprnt
```

Sudo Following Avijit and Harper [8], we now give a well-typed version of the Unix command sudo, which allows switching principals during execution. A first cut for the type of sudo is as follows:

```
sudo1 :  ∀ { Γ A Γ’} → (k1 k2 : _)
 → (Proof Γ (a- (MaySu · (k1 ,  k2))))
 → ○ ((a+ (As · k2)) :: Γ) A (λ _ → (a+ (As · k2)) :: Γ’)
 → ○ ((a+ (As · k1)) :: Γ) A (λ _ → (a+ (As · k1)) :: Γ’)
```

If there is a proof that k1 may sudo as k2 (e.g., a password was provided), and As(k1) is in the precondition, then it is permissible to run a subcomputation as k2. This subcomputation has a postcondition saying that it terminates running as k2, and then the overall computation returns to running as k1. Because our contexts are ordered (represented as lists rather than sets), sudo has the type in Figure 2, which allows the As facts to occur anywhere in the context. sudo’s type may be read: if replacing As(k1) with As(k2) in Γ equals Δ, and if replacing As(k2) with As(k1) in Δ’ equals Γ’, and k2 has permission to su as k1, then a computation with preconditions Δ and postconditions Δ’ can produce a computation with preconditions Γ and postconditions Γ’.

The following example call to sudo defines a computation as Dan that su’s as Jamie to run the computation jreadprint defined above:

```
drdprnt : ○ (Γall as "Dan") Unit (λ _ → Γall as "Dan")
drdprnt = sudo (Prin "Dan" ) (Prin "Jamie")
               (solve proveReplace)
               (λ _ → solve proveReplace)
               (solve (prove 15))
               jreadprint
```

This requires proving that Γstate as "Jamie" and Γstate as "Dan" are related by replacing As(Prin "Jamie") with As(Prin "Dan") (in both directions). Our tactic proveReplace proves all of these equalities. Additionally, the theorem prover statically verifies Dan may su as Jamie under the policy Γall as "Dan".

Acquire The function acquire allows a program to check whether a proposition is true in the state of the world. This construct is inspired by acquire in PCML5, but there are slight differences: in PCML5, acquire does theorem proving to prove an arbitrary proposition from the policy, whereas here acquire only verifies the truth of state-dependent atomic facts (which have no evidence) and statements of principals (whose only evidence is a digital signature [9, 24]). The function acquire takes two continuations: one to run if the check is successful, whose precondition is extended with the proposition, and an error handler, whose precondition is the current context, to run if the check fails. In fact, we allow acquire to test an entire context at once: given a context Γn, a computation with preconditions Γ extended with Γn (the success continuation), and a computation with preconditions Γ (the error continuation), acquire returns a computation with preconditions Γ. We use the notation acquire Γn / _ no⇒ s yes⇒ f to write a call to acquire in a pattern-matching style. The _ elides a Good argument, which is explained below.

```
main : ○ [] Unit (λ _ → [])
main = acquire (Γall as "Jamie") / _
```

```
        no⇒ error "acquiring policy failed"
        yes⇒ weakenPost jreadprint (λ _ ()) _
```

This example call begins and ends in the empty context. The call to acquire examines the system state to check the truth of each of the propositions in Γall as "Jamie". If all of these are true, then we run jreadprint and use weakening to forget the postconditions. If some proposition cannot be verified, then main calls error.

2.1.4 Verifying Policy Invariants

When authoring the above monadic signature for file IO, the programmer may have in mind some invariants to which policies Γ must adhere. For example, a call to chown (above) would have unexpected consequences if there ever were more than one copy of System says owns(k,f) in Γ (only one copy would be replaced, leaving a file with two owners in the postcondition). Our interface permits programmers to specify context invariants using a predicate Good Γ. The intended invariant of our interface is that a monadic computation ◯ Γ A Γ' should have the property that Γ' satisfies Good if Γ does. To achieve this, the weakening operations and acquire require preconditions Γ be accompanied by a proof of Good Γ, and the programmer must verify that operations such as read, chown, and sudo preserve the invariant. Because of this invariant, it is not necessary to make each monadic operation require a proof that the precondition is Good. This means, that when writing a client program, the programmer needs only to verify that the initial policy and those in calls to weakening and acquire satisfy the invariants.

In the above examples, we took Good to be the trivially true invariant, so the proofs could be elided with an _. As mentioned above, a useful invariant to enforce is that for every file f there is at most one statement of the form System says Owner(_ , f) in the context. This is defined in Agda as follows:

```
Good : TCtx+ [] → Set
Good Γ = ∀ {k k' f}
 → (a : (Prin "System" says (a- (Owner · (k , f)))) ∈ Γ)
 → (b : (Prin "System" says (a- (Owner · (k' , f)))) ∈ Γ)
 → Equal a b
```

Then we may prove that the postcondition of each operation is Good if the precondition is; e.g.

```
ChownPreservesGood : ∀ {Γ Δ k1 k2 f}
 → Replace (Prin "System" says (a-(Owner · (k1 , f))))
          (Prin "System" says (a-(Owner · (k2 , f))))
          Γ Δ
 → Good Γ → Good Δ
```

In the companion code, we revise the above examples so that they maintain this invariant, using a tactic to generate the proofs.

2.2 File IO with Access Control and Information Flow

Next, we extend the above file signature with information flow, adapting an example from Fine [38]. First, we define a type Tracked A L which represents a value of type A tracked with security level L, where L is a list of filenames and ⊔ appends two lists. Following Fine, we define Tracked as an abstract functor that distributes over functions (though different type structures for information flow, such as an indexed monad [36], can be used in other examples):

```
Tracked : Set → Label → Set
fmap : ∀ {A B L} → (A → B) → Tracked A L → Tracked B L
_⊙_ : ∀ {A B L1 L2} → Tracked (A → B) L1
      → Tracked A L2 → Tracked B (L1 ⊔ L2)
```

An application f ⊙ x joins the security levels of the function and the argument.

Next, we give flow-sensitive types to read and write: read tags the value with the file it was read from, and write requires a proof of `MayAllFlow provs file`, representing the fact that all of the files upon which the written string depends may flow into `file`.

```
read  : ∀ {Γ} (k : _) (file : _)
 → Proof Γ ((a- (Mayread · (k ,  file))) ∧ (a+ (As · k)))
 → ◯ Γ (Tracked String [ file ]) (λ _ → Γ)

write  : ∀ {Γ provs} (k : _) (file : _)
 → Tracked String provs
 → Proof Γ ( (a- (Maywrite · (k ,  file)))
           ∧ (a+ (As · k))
           ∧ (MayAllFlow provs file))
 → ◯ Γ Unit (λ _ → Γ)
```

For example, we can read two files and write their concatenation to `secret.txt`:

```
go : ◯ (Γ as "Jamie") Unit (\ _ → (Γ as "Jamie"))
go = read (Prin "Jamie") (File "file1.txt")
          (solve (prove 15)) >>= \ s →
     read (Prin "Jamie") (File "file2·txt")
          (solve (prove 15)) >>= \ s' →
     write (Prin "Jamie") (File "secret·txt")
           ((fmap String.string-append s) ⊙ s')
           (solve (prove 15))
```

Here the theorem prover shows that both `file1.txt` and `file2.txt` may flow into `secret.txt`, according to the policy. This proof obligation results from the fact that
`(fmap String.string-append s) ⊙ s'`
has type
`Tracked String [ "file1.txt" , "file2.txt" ]`.

2.3 Spatial Distribution with Information Flow

PCML5 investigates PCA for the spatially distributed programming language ML5 [29]. Here, we show how to embed an ML5-style type system, which can be combined with the above techniques for access control and information flow. PCML5 considers additional aspects of distributed authorization, such as treating the policy itself as a distributed resource, which we leave to future work.

ML tracks where resources and computations are located using modal types of the form A @ w. For example, database.read : (key → value) @ server says that a function that reads from the database must be run at the server, while javascript.alert : (string → unit) @ client says that a computation that pops up a browser alert box must be run at the client. Network communication is expressed in ML5 using an operation get : (unit → A) @ w → A @ w' that (under some conditions which we elide here) goes to w to run the given computation and brings the resulting value back to w'. In other work [27], we have shown how to build an ML5-like type system on top of an indexed monad of computations at a place, ◯ w A, with a rule get : ◯ w' A → ◯ w A. Here, observe that this monad indexing can be represented using a proposition At(w), where get is given a type analogous to sudo:

```
get : (w1 w2 : _) → ∀ {Γ A Γ' Δ Δ'}
      → Replace (a+ (At · w1)) (a+ (At · w2)) Γ Δ
      → Replace (a+ (At · w2)) (a+ (At · w1)) Δ' Γ'
      → ◯ Δ A (\ _ → Δ')
      → ◯ Γ (Tracked A w2) (\ _ → Γ')
```

Additionally, we combine spatial distribution with information flow, tagging the return value of the computation with the world it is from. The postcondition must be independent of the return value, as there is in general no coercion either way between A and Tracked A L.

Information flow can be used in this setting to force strings to be escaped before they are sent back to the client—e.g. to prevent SQL injection attacks:

```
sanitize : Tracked String (client) → HTML
str : Tracked String (server) → HTML
```

Strings from the client must be escaped before they can be included in an HTML document, whereas strings from the server are assumed to be non-malicious, and can be included directly.

In our technical report [28], we extend this example with a simple database interface that enforces both authorization and spatial distribution—database handles are only used at the server.

2.4 ConfRM: A Conference Management System

Swamy et al. [38] present an example of a conference management server, ConfRM, adapted from CONTINUE [26] and its access control policy [18]. Here, we show an excerpt of an authorization policy for ConfRM, a proof-carrying monadic interface to the computations which perform actions, and the main event loop of the server. This example uses ephemeral policies: authorization to perform actions, such as submitting a paper or a review, depend on the phase of the conference (submission, notification,. . .).

2.4.1 Policy

We formalize ConfRM's policy using terms of various types: `actions` represent requests to the web server; `principals` represent users; `papers` and `strings` are used to specify actions; `roles` define whether a user is an `Author, PCMember`, and so on. The policy is also dependent on the `phase` of the conference (e.g., an `Author` may submit a paper during the `submission` phase). The proposition `May · (k , a)` states that `k` may perform action `a`. Each action is a first-order term constructed from some arguments (e.g., `Submit, Review, Readscore, Read` all have papers, while Progress has two phases, the phase the conference is in before and after it is progressed).

Fine specifies the policy as a collection of Horn clauses, which are simple to translate to our logic, as in the following clause:

```
clause1 =
 ((∀e principal · ∀e string ·
     let
       author = ▷ iS i0
       papername = ▷ i0
     in
      (((a- (InPhase · (Submission))) ∧
        ((a- ( InRole · (author  , Author))))
        ⊃ (a- (May · (author , (Submit · papername))))))
```

This proposition reads: for all authors and paper names, if the conference is in the submission phase, and the principal is an author, then the principal may submit a paper. We have also begun to reformulate the policy using the says modality, e.g. to allow authors to share their paper scores with their coauthors.

```
saysClause =
 ((∀e principal · ∀e paper · ∀e principal ·
   let primary = ▷ i0
       paper = ▷ (iS i0)
       coauthor = ▷ (iS (iS i0)) in
   (( ((a- (InPhase · (Notification))) ∧
      ((a- (Author · (primary , paper) ))) ∧
      (primary says (a- (May · (coauthor ,
                                (Readscore · paper))))))
    ⊃ (a- (May · (coauthor , (Readscore · paper)))))))
```

This rule states that, for any principal author, paper paper, and principal coauthor, if the conference is in notification phase, and author is the author of paper, and author says coauthor may read the scores for paper, then coauthor may read the scores for paper. Similarly, using says, it is straightforward to specify a policy allowing PC members to delegate reviewing assignments to subreviewers.

2.4.2 Actions

Rather than defining a command for each action—doRead, doSubmit, etc.— we use type-level computation to write one command for processing all actions; this simplifies the code for the main loop presented below and allows for straightforward addition of actions. The generic command for processing an action, doaction, has the following type:

```
doaction : ∀ {Γ} (k : _) (a : _) → (e : ExtraArgs Γ a)
→ Proof Γ  (a- (May · (k , a))) ∧ (a+ (As · k))
→ ◯ Γ (Result a) (λ r → PostCondition a Γ e k r)
```

doaction takes a principal k, an action a to perform, and some ExtraArgs for a, along with a proof that the computation is running as k, and that k may perform a. In this example, a Proof abbreviates a sequent whose view is PCChair, rather than Admin. It returns a Result, and has a PostCondition, both of which are dependent upon the action being performed. In Agda, ExtraArgs, Result, and PostConditions are functions defined by recursion on actions, which compute a Set, a Set, and a context, respectively.

Several actions, such as Submitting a paper, require extra data that is not part of the logical specification (e.g., the contents of the paper should not be part of the proposition which authorizes it to be submitted). ExtraArgs produces the set of additional arguments each action requires.

```
  ExtraArgs : TCtx+ [] → Term [] (action) → Set
  ExtraArgs Γ (Review · _) = Term [] (string)
  ExtraArgs Γ (Submit · _) = Term [] (string)
  ExtraArgs Γ (Progress · (p1 , p2)) = Σ (λ Δ →
    Replace (a- (InPhase ·  p1))
```

```
        (a- (InPhase ·  p2)) Γ Δ)
  ExtraArgs Γ _ = Unit
```

Reviews and paper submissions require their contents, represented as terms of type string (the Agda type Term [] (string) is an injection of strings into the language of first-order terms that we use to represent propositions, as described in Section 3 below). Progressing the phase of the conference requires a proof that the conference is in the first phase, along with a new context in the resulting phase, which we represent by a pair of a new context Δ and a proof of Replace.

Next, we specify the result type of an action:

```
  Result : Term [] (action) → Set
  Result (Submit · _) = Term [] (paper)
  Result (Review · _) = Unit
  Result (BeAssigned · _) = Unit
  Result (Readscore · _) = String
  Result (Read · _) = String
  Result (Progress · _) = Unit
```

Readscore and Read return a paper's reviews and contents, while submit produces a Term [] paper, a unique id for the paper.

Finally, we define the PostCondition of each action, which is dependent upon the action itself, the precondition, the extra arguments for the action, the principal performing the action, and the Result of the action. Submitting a paper extends the preconditions with two propositions: one saying the paper has been submitted, and one saying the submitting principal is its author. Reviewing and Assigning a paper add that the paper is reviewed

```
 fix : ∀ {A Γ'}
  → ( (∀ {Γ} → ◯ Γ A Γ') → (∀ {Γ} → ◯ Γ A Γ') )
  → (∀ {Γ} →  ◯ Γ A Γ')}

 main : ∀ {Γ} → ◯ Γ Unit (λ _ → [])
 main  = fix loop where
```

```
  loop : (∀ {Γ} → ◯ Γ Unit (λ _ → [])) →
         (∀ {Γ} → ◯ Γ Unit (λ _ → []))
  loop rec {Γ} =
{-1-} prompt "Enter an action:" >>= λ astr →
       case (parseAction astr)
        None⇒ error "Unknown action"
        Some⇒ λ actionArgs →
         let a = (fst actionArgs)
             args = (snd actionArgs) in
{-2-}     prompt "Who are you?" >>= λ ustring →
           let u = parsePrin ustring in
{-3-}       acquire [ ((a- (MaySu · (Prin "Admin" , u)))) ]
                   / _
             no⇒ error "Unable to su"
{-4-}        yes⇒ case make-replace
              None⇒ error "oops, not running as admin"
              Some⇒ λ asadmin →
{-5-}          case (inputToE a _ args)
                None⇒ error "Bad input (e.g. not in phase)"
                Some⇒ λ args →
{-6-}            (sudo (Prin "Admin") u
                       (snd asadmin)
                       (\x → (snd (repAsPost (snd asadmin)
                                              {a} x)))
                       (lfoc i0 init-)
{-7-}                  (prove/dyn 15 _ _ >>=
                         none⇒ error "Unauthorized action"
                         some⇒ λ canDoAction →
{-8-}                     doaction u a args canDoAction) )
{-9-}            >>= λ _ → rec
```

Figure 3. ConfRM Main Loop

by or assigned to the principal, respectively. Readscore and Read leave the conditions unchanged. The postcondition of Progress is the first component of its ExtraArgs, i.e. the context determined by replacing the current phase with the resulting one.

```
PostCondition : (a : Term [] (action)) (Γ : TCtx+ [])
  → ExtraArgs Γ a → (k : Term [] (principal))
  → Result a  → TCtx+ []
PostCondition (Submit · y)             Γ e k r    =
 (a- (Submitted · r )) :: (a- (Author · (k , r))) :: Γ
PostCondition (Review · y)             Γ e k r    =
 (a- (Reviewed · (k , y))) :: Γ
PostCondition (BeAssigned · y)         Γ e k r    =
 (a- (Assigned · (k , y))) :: Γ
PostCondition (Readscore · y)          Γ e k r    = Γ
PostCondition (Read · y)               Γ e k r    = Γ
PostCondition (Progress · (ph1 , ph2)) Γ e k r    =
 (fst e)
```

In writing the main server loop, we will use the following monadic wrapper of our theorem prover, in order to test at run time whether a given proposition holds in the current state of the server:

```
 prove/dyn : ∀ {Γ1} → Nat → (Γ : TCtx+ []) →
   (A : Propo- []) →
   ◯ Γ1 (Maybe (Proof Γ A)) (λ _ → Γ1)
```

2.4.3 Server Main Loop

In Figure 3 we show the code for the main loop of the ConfRM server, implemented using the interface described above. The main loop serves requests made by principals who wish to perform actions. Because the requests are not determined until run-time, and authorization depends on the system state (the phase of the conference, the role of a principal), this example uses entirely dynamic verification of security policies: the server dynamically checks that each request is authorized just before performing it, using our theorem prover at run-time. The type system ensures that the appropriate dynamic check is made. Informally, the server loop works by

(1) reading in an action and its arguments, (2) reading in a principal, (3) acquiring the credentials to su as that principal, (4) computing the precondition of the su, (5) computing the postconditions of performing the action, (6) su-ing as the principal, (7) proving the principal may perform the action, (8) performing the action, and (9) recurring. The fact that we have coalesced all of the actions into one primitive command makes this code much more concise than it would be otherwise, when we would have to repeat essentially this code as many times as there are actions.

This code is rendered in Agda as follows. `fix` permits an IO computation to be defined by general recursion. Because its type is restricted to the monad, it does not permit non-terminating elements of other types, such as `Proof`. This fixed-point combinator abstracts over the precondition, so it may vary in recursive calls, but leaves the postcondition fixed throughout the loop; we leave more general loop invariants to future work. First, `main` is given the type `∀ {Γ} → ◯ Γ Unit (λ _ → [])`: given any precondition, the computation returns unit and an empty postcondition (we do not expect to run any code following `main` so it is not worthwhile to track the postconditions). `main` is defined by taking the fixed point of the axillary function `loop`, which is abstracted over the recursive call. On line (1), the loop `prompts` the user to enter an action to perform, `parseAction` then parses the string to produce `a : action` and `args: InputArgs`, and raises an error otherwise. (2) The loop prompts for a username, parses it into a `Term [] principal`. (3) The loop attempts to acquire credentials that `"Admin"` may su as the principal (e.g., by prompting for a password). (4) The loop calls the functions `make-replace` to produce the preconditions for the `su`, by replacing `(As (Prin "Admin"))` with `a+ (As u)`. (5) The loop calls `inputToE` to produce the `ExtraArgs` for the action from the `args`; for `Progress`, this function computes the postcondition of the action from the current context. (6) The loop su-s as the principal. The first `replace` argument to `su` is the result of step (4), the proof argument is the assumption acquired in step (3), the second replace argument is discussed below. (7) The loop calls the theorem prover at run-time to prove the principal may perform the requested action. (8) The loop calls doaction and (9) recurs.

The second replace argument to su is generated using a proof that As is preserved in the PostCondition of an action:

```
postPreservesAs : ∀ {a Γ e k r k' }
                → (a+ (As · k') ∈ Γ)
                → ((a+ (As · k')) ∈ PostCondition a Γ e k r)
```

This is another example of using Agda to verify invariants of the pre- and post-conditions, as in Section 2.1.4.

2.4.4 Dynamic Policy Acquisition

Finally, we describe an example of dynamic policy acquisition in Figure 4: we read the reviewers' paper assignments from a database, parse the result into a context, acquire the context, and start the main server loop with those preconditions. This is simple in a dependently typed language because contexts themselves are data. The function getReviewerAsgn takes a string, representing a path to the database, and returns the list of reviewers for each paper. The function parseReviewers then turns each of these lists into lists of propositions, each stating the parsed reviewer is a reviewer of the paper. A more realistic ConfRM implementation would read a variety of other propositions from the database as well

```
 getReviewerAsgn : ∀ {Γ} → String →
   ◯ Γ (List (List String)) (λ _ → Γ)

parseReviewers : List String → TCtx+ []

mkPolicy : ∀ {Γ} → ◯ Γ (TCtx+ []) (λ _ → Γ)
mkPolicy =  getReviewerAsgn "papers.db" >>= λ asgn →
  return (ListM.fold [] (λ x → λ y →
                    parseReviewers x ++ y) asgn)
```

```
start = mkPolicy {[]} >>= λ ctx →
        acquire ctx / _
          no⇒ error "policy not accepted"
          yes⇒ main
```

Figure 4. ConfRM Policy Acquistion

(which papers have been submitted, reviewed, etc.) The computation `mkPolicy` calls `getReviewerAsgn` and parses the results. The computation `start` uses `mkPolicy` to generate an initial policy, acquires these preconditions, and starts the main sever loop.

3. Implementation

Our Agda implementation consists of about 1400 lines of code. We have also written about 1800 lines of example code in the embedded language, including policies, monadic interfaces to primitives, and example programs. In this section, we describe the implementation of the logic, the theorem prover, and the indexed monad.

3.1 Representing BL$_0$

BL$_0$ [21] extends first-order intuitionistic logic with the modality $k\ \mathsf{says}\ A$. While a variety of definitions of says have been studied (Abadi [2] overviews some of the approaches), in BL$_0$, says is treated as a necessitation ($\Box$) modality, and *not* as a lax modality (i.e. a monad) [1, 8, 22, 24]. The definition of says in BL$_0$ supports *exclusive delegation*, where a principal delegates responsibility for a proposition to another principal, without retaining the ability to assert that proposition himself. For example, consider a policy that $\mathsf{payroll}\ \mathsf{says}\ \forall t.(\mathsf{HR}\ \mathsf{says}\ \mathsf{employee}(t)) \supset \mathsf{MayBePaid}(t)$. Under what circumstances can we conclude payroll says MayBePaid(Alice)? The fact that HR says employee(Alice) should be sufficient. However, the fact that payroll says employee(Alice) should not, as the intention of the policy is that payroll delegates responsibility for the employee predicate to human resources, without retaining the ability to assert employee instances itself. When says is treated as a lax modality, payroll says employee(Alice) implies payroll says HR says employee(Alice), which is enough to conclude the goal. Abstractly, we wish k says A to imply k' says (k says A), but not k says (k' says A). The modality satisfies several other axioms: for example, principals say all consequences the statements they have made (k says $(p \supset q)$ entails (k says $p \supset k$ says q)) and principals believe what they say is true (k says ((k says s) $\supset s$)).

3.1.1 Terms, Types, and Atomic Propositions

In the above examples, we used a variety of atomic propositions (`Mayread`, `Owns`, etc.), which refer to several datatypes (principals, papers, conference phases, etc.). We have parametrized the representation of BL$_0$ and its theorem prover over such datatypes and atomic propositions by defining a generic datatype of first-order terms, with free variables, over a given signature. This allows us to specify the types, terms, and propositions for an example concisely, while exploiting a datatype-generic definition of weakening, substitution, etc., which are necessary to state the inference rules of the logic. The following excerpt from the signature for ConfRM illustrates what programmers write to define an individual example:

```
data BaseType : Set where
  string paper role action phase principal : BaseType

data Const : BaseType -> Set where
```

```
  Prin  : String -> Const principal
  Paper : String -> Const paper
  PCChair Reviewer Author Public : Const role
  Init Presubmission Submission ... : Const phase

data Func : BaseType -> Type -> Set where
  Review BeAssigned ...   : Func action (paper)
  Progress  : Func action  (phase ⊗ phase)

data Atom : Type -> Set where
  InPhase : Atom (phase)
  Assigned ... : Atom (principal ⊗ paper)
  May :  Atom (principal ⊗ action)
  As : Atom (principal)
```

The programmer defines a datatype of base types, a datatype giving constants of each type, a datatype of function symbols, and a datatype of atomic propositions over a given type. Additionally, the programmer must define a couple of operations on these types (equality, enumeration of all elements of a finite type) which in a future version of Agda could be generated automatically [5].

Types are BaseTypes, unit and pair types (τ1 $\otimes$ τ2). The terms over a signature are given by a datatype Term Ω τ, where Ω, an individual context (ICtx), represents the free variables of the term. An ICtx is a list of BaseTypes, and represents a context of individual variables—e.g. the context $x_1 : \tau_1, \ldots, x_n : \tau_n$ will be represented by the list τ_1 :: ... :: τ_n :: []. Variables are represented by well-scoped de Bruijn indices, which are pointers into such a list—i0 says x $\in$ (x :: l), and iS says that x $\in$ (y :: l) if x $\in$ l. Terms are either variables (▷ i), where i : $\tau \in \Omega$ is a de Bruijn index, constants, applications of function symbols (f · t), or [] and (t1 , t2) for unit and product types. Atomic propositions are represented by a datatype Aprop Ω. An atomic proposition `p · t` consists of an `Atom` paired with a term of the appropriate type. We have defined weakening and substitution generically on terms and propositions, and proved several properties of them (e.g. functoriality of weakening).

3.1.2 Propositions

BL$_0$ propositions include conjunction, disjunction, implication, universal and existential quantification, and the says modality:

$$\begin{array}{lcl} A, B, C & ::= & P \mid A \wedge B \mid A \vee B \mid A \supset B \mid \top \\ & & \mid \bot \mid \forall x : \tau . s \mid \exists x : \tau . A \mid k \ \mathsf{says} \ A \end{array}$$

In Figure 5, we represent this syntax in Agda. Propositions (`Propo`) are indexed by a context of free variables, and additionally by a *polarity* (+ or -), which will be helpful in defining a focused sequent calculus below. Because the syntax of propositions is polarized, there are two injections `a-` and `a+` from atomic propositions `Aprop` to negative and positive propositions, respectively. Additionally, the shifts ↓ and ↑ include negative into positive and vice versa. The remaining datatype constructors correspond to the various ways of forming propositions in the above grammar. For example, the _∧_ constructor takes two terms of type `Propo+` Ω and returns a term of type `Propo+` Ω. The constructor ∃i (existential quantification over individuals), takes a positive proposition, in a context with one new free variable of type τ, and returns a positive proposition in the original context Ω.

We have suppressed the shifts up to this point in the paper for readability. We could suppress shifts in our Agda code by implementing a simple translation that, given an unpolarized proposition

```
data Propo : Polarity → ICtx → Set where
  _⊃_ : ∀ {Ω} → Propo+ Ω → Propo- Ω → Propo- Ω
  ∀i_ : ∀ {Ω τ} → Propo- (τ :: Ω) → Propo- Ω
  a-  : ∀ {Ω} → Aprop Ω → Propo- Ω
  ↑   : ∀ {Ω} → Propo+ Ω → Propo- Ω
```

```
_∨_    : ∀ {Ω} → Propo+ Ω → Propo+ Ω → Propo+ Ω
_∧_    : ∀ {Ω} → Propo+ Ω → Propo+ Ω → Propo+ Ω
⊥      : ∀ {Ω} → Propo+ Ω
⊤      : ∀ {Ω} → Propo+ Ω
∃i_    : ∀ {Ω τ} → Propo+ (τ :: Ω) → Propo+ Ω
_says_ : ∀ {Ω} → Term Ω principal →
          Propo- Ω → Propo+ Ω
a+     : ∀ {Ω} → Aprop Ω → Propo+ Ω
↓      : ∀ {Ω} → Propo- Ω → Propo+ Ω
```

Figure 5. Agda Representation of BL_0 Propositions

and an intended polarization of each atom, computes a polarized proposition with minimal shifts.

3.1.3 Proofs

Sequent calculus. Sequents in BL_0 have the form $\Omega; \Delta; \Gamma \xrightarrow{k} A$. The context Ω gives types to individual variables (e.g. it is extended by $\forall$), and the context Γ contains propositions that are assumed to be true (e.g. it is extended by $\supset$)—these are the standard contexts of first-order logic. The context Δ contains *claims*, assumptions of the form k' claims A; claims is the judgement underlying the says connective [21, 33]. Finally, k, the *view* of the sequent, is the principal on behalf of whom the inference is made.

The rules for says are as follows:

$$\frac{\Omega; \Delta; [] \xrightarrow{k} A}{\Omega; \Delta; \Gamma \xrightarrow{k0} k \ \mathsf{says}\ A}\ \text{SAYSR}$$

$$\frac{\Omega; \Delta, (k \text{ claims } A); \Gamma, (k \text{ says } A) \xrightarrow{k0} C}{\Omega; \Delta; \Gamma, (k \text{ says } A) \xrightarrow{k0} C} \text{ SAYSL}$$

$$\frac{\Omega; (\Delta, k \text{ claims } A); (\Gamma, A) \xrightarrow{k0} C \qquad k \geq k0}{\Omega; (\Delta, k \text{ claims } A); \Gamma \xrightarrow{k0} C} \text{ CLAIMSL}$$

In order to show k says A, one empties the context Γ of true assumptions, and reasons on behalf of k with the goal A (rule `saysR`). It is necessary to empty Γ because the facts in it may depend on claims by the principal k_0, which are not valid when reasoning as k. The rule `saysL` says that if one is reasoning from an assumption k says A, one may proceed using a new assumption that k claims A. Claims are used by the rule `claimsL`, which allows passage from a claim k claims A to an assumption that A is actually true. This rule makes use of a preorder on principals, and asserts that any statements made by a greater principal are accepted as true by lesser principals.

Focused sequent calculus. To help with defining a proof search procedure, we present BL_0 as a weakly-focused sequent calculus. Garg [21] describes both an unfocused sequent calculus and a focused proof system for FHH, a fragment of BL_0; here we give a focused sequent calculus for all of BL_0. Focusing [6] is a proof-theoretic technique for reducing inessential non-determinism in proof search, by exploiting the fact that one can chain together certain proof steps into larger steps. In the Agda code above, we polarized the syntax of propositions, dividing them into positive and negative classes. Positive propositions, such as disjunction, require choices on the right, but are invertible on the left: a goal C is provable under assumption A^+ if and only if it is provable under the left rule's premises. Dually, negative propositions involve choices on the left but are invertible on the right. Weak focusing [34] forces focus (choice) steps of like-polarity connectives to be chained together, but does not force inversion (pattern-matching) steps to be chained together. We use weak, rather than full, focusing because it is slightly easier to represent in Agda, and because it can sometimes lead to shorter proofs if one internalizes the identity principles (which say that A entails A)—though we do not exploit this fact in our current prover.

The polarity of $k \;\mathsf{says}\; A$ is as follows: A is negative, but $k \;\mathsf{says}\; A$ itself is positive. As a simple check on this, observe that $k \;\mathsf{says}\; A$ is invertible on the left—one can always immediately make the claims assumption—but not on the right—because `saysR` clears the true assumptions. For example, a policy is often of the form $k_1 \;\mathsf{says}\; A_1, \ldots k_n \;\mathsf{says}\; A_n$, with a goal of the form $k' \;\mathsf{says}\; B$. It is necessary to use `claimsL` to turn all propositions of the form $k \;\mathsf{says}\; A$ in Γ into claims in Δ before using `saysR` on the goal—if one uses `saysR` first, the policy would be discarded. This polarization is analogous to $\Box$ in Pfenning and Davies [33] and to ! in linear logic [6], which is reasonable given that says is a necessitation modality.

Our sequent calculus has three main judgements:

- Right focus: $\Omega; \Delta; \Gamma \xrightarrow{k} [A^{+}]$
- Left focus: $\Omega; \Delta; \Gamma \xrightarrow{k} [A^{-}] > C$
- Neutral sequent: $\Omega; \Delta; \Gamma \xrightarrow{k} C^{-}$

Here Δ consists of claims $k \;\mathsf{claims}\; A^{-}$ and Γ consists of positive propositions. For convenience in the Agda implementation, we break out a one-step left-inversion judgement $\Omega; \Delta; \Gamma \xrightarrow{k} A^{+} >_{I} C$, which applies a left rule to the distinguished proposition A^{+} and then reverts to a neutral sequent. The rules are a fairly simple integration of the idea of weak focusing [34] with the focusing interpretation of says described above. The interested reader can find the inference rules for these judgements in the extended version of this paper [28].

Agda Representation In Figure 6, we show an excerpt of the Agda representation of this sequent calculus. First, we define a record type for a `Ctx`, which tuples together the Ω, Δ, Γ, and k parts of a sequent—we write Θ for such a tuple. Γ is represented as a list of propositions; Δ is represented as a list of pairs of a principal and a proposition, written `p claims A`; k is a term of type principal. Record fields are selected by writing `R.x`, where the type of the record is `R` and the desired field is `x` (e.g., `Ctx.rk` selects the principal from a `Ctx` record). Note that `Ctx` is a dependent record: the true context, the claims context, and the view can mention the variables bound in the individual context `rΩ`. We write `TCtx+ Ω` for `List (Propo+ Ω)`. We define several helper functions on `Ctx`s: `sayCtx` clears the `Ctx` of true propositions, and changes the view of the context to its second argument. `ictx` (not shown) is shorthand for `Ctx.rΩ`. `addTrue` and `addClaim` (not shown) add a true proposition onto Γ or a claim onto Δ, respectively. `addVar` adds a variable to Ω, and *weakens* the rest of the context.

When writing down the calculus on paper, it is obvious that extending Ω does not affect Γ or Δ; any variables bound in Ω will be bound in $\Omega' \supseteq \Omega$. However, in Agda, it is necessary to explicitly coerce `F Ω` to `F Ω′` for type families `F` dependent on Ω. We have defined weakening functions for many of the types indexed by Ω: terms (`weakenTerm`), propositions, claims, true contexts (`weakenT+`), claims contexts (`weakenC`), and so on.

There are 4 judgments in our weakly-focused sequent calculus; analogously, there are 4 mutually recursive datatype declarations representing these judgements in Agda, with one datatype constructor for each inference rule. We show the constructors `∀L` (for the left focus judgement), ∃L and saysL (for the left inversion judgement), saysR (for the right focus judgement), and claimsL (for

```
record Ctx : Set where
  field rΩ  : ICtx
        rΓ+ : List (Propo+ rΩ)
          -- pairs written (k claims A)
        rΔ  : List (Term rΩ principal × Propo- rΩ)
        rk  : Term rΩ principal

addVar : (θ : Ctx) → (A : Type) → Ctx
addVar θ τ = record {rΩ  = (τ :: Ctx.rΩ θ) ;
             rΓ+ = (weakenT+ (Ctx.rΓ+ θ) iS) ;
             rΔ  = (weakenC (Ctx.rΔ θ) iS) ;
             rk  = (weakenTerm (Ctx.rk θ) iS)}

sayCtx : (θ : Ctx) →
         (k : Term (Ctx.rΩ θ) principal) → Ctx
sayCtx θ k = (record {rΩ = Ctx.rΩ θ ;
             rΓ+ = [] ; rΔ = Ctx.rΔ θ ; rk = k})

mutual
  data _⊢L_>_ : (θ : Ctx) → Propo- (ictx θ)
              → Propo- (ictx θ) → Set where
    ∀L    : ∀ {θ τ A C} → (t : Term (ictx θ) τ) →
              θ ⊢L (substlast A t) > C →
              θ ⊢L ∀i_ {ictx θ}{τ} A > C
    ...
  data _⊢I_>_ : (θ : Ctx) → (Propo+ (ictx θ))
       → Propo- (ictx θ) → Set where
    ∃L : ∀ {θ  τ A C}
       → (addTrue (addVar θ τ) A) ⊢ (weakenP C iS)
       → θ ⊢I (∃e τ A) > C
    saysL : ∀ {θ k s B}
          → addClaim θ (k claims s) ⊢ C
          → θ ⊢I (k says s) > C
    ...
```

```
data _⊢R_ : (θ : Ctx) → Propo+ (ictx θ) → Set where
  saysR : ∀ {θ k A}
        → (sayCtx θ k) ⊢ A
        → θ ⊢R (k says A)
  ...
data _⊢_ :  (θ : Ctx) → Propo- (ictx θ) → Set where
  claimsL : ∀ {θ k A C}
          → (k claims A) ∈ Ctx.rΔ θ
          → θ ⊢L A > C  →  k ≥ Ctx.rk θ
          → θ ⊢ C
  ...
```

Figure 6. Agda representation of proofs (exceprt)

the neutral sequent judgement). For the most part, the rules are a straightforward transcription of the sequent calculus rules [28]. In ∀L, the function `substlast` substitutes a term for the last variable in a proposition; we have implemented substitution for individual variables for each of the syntactic categories. In ∃L, it is necessary to weaken the goal with the new variable, which is tacit in on-paper presentations.

Properties Because the sequent calculus is cut-free, consistency of closed proofs is immediate:

> **Consistency:** For all principals k, there is no derivation of $[]; []; [] \xrightarrow{k}\uparrow \bot$.

Proof: no rule concludes $\bot$ in right focus, and in the empty context no left focus or left inversion rules apply.

Identity and cut can be proved using the usual syntactic methods, adapting Garg's proof [21] for an unfocused sequent calculus to weak focusing, following Pfenning and Simmons [34].

3.2 Proof Search

We have implemented a simple proof-producing theorem prover for BL$_0$:

```
prove : Nat → (θ : Ctx) → (A : Propo- (ictx θ))
      → Maybe (θ ⊢ A)
```

`prove` takes a depth bound, a context, and a proposition, and attempts to find a proof of $\theta \vdash \mathtt{A}$ with at most the given depth. The prover is *certified*: when the prover succeeds, it returns a proof, which is guaranteed by type checking to be well-formed. When the prover fails, it simply returns `None`. The prover is implemented by around 200 lines of Agda code.

Our prover is quite naïve, but it suffices to prove the examples in this paper. For the most part, the prover backchains over the focusing rules. However, whereas the above sequent calculus was only weakly focused, the prover is fully focused, in that it eagerly applies invertible rules, which avoids backtracking over different applications of them. If the goal is right-invertible, the prover applies right rules. Once the goal is not right-invertible (an atom or a shift $\uparrow A^+$), the prover fully left-inverts all of the assumptions in Γ. Inverting a context Γ breaks up the positive propositions using left rules, generating a list of non-invertible contexts $\Theta_1, ..., \Theta_k$ such that, if for every i, $\Theta_i \vdash C$, then $\Theta \vdash C$. Once the sequent has been fully inverted, the prover tries right-focusing (if the goal is a shift $\uparrow A^+$) and left-focusing on all assumptions in Γ and claims in Δ, until one of these choices succeeds. The focus phases involves further backtracking over choices (e.g., which branch of a disjunction to take). The focus rules for quantifiers ($\forall E$ and $\exists I$) require guessing an instantiation of the quantifier. Our current implementation is brute-force: it simply computes all terms of a given type in a given context and tries each of them in turn—we have only considered individual types with finitely many inhabitants.

The prover achieves tolerable compile times on the small examples we have considered so far (1 to 13 seconds). If it proves too slow for some examples, we have several options: First, we can improve our implementation—e.g. by implementing unification, which will eliminate much of the branching from quantifiers, or by doing a better job of clause selection. Second, we could connect Agda with an external theorem prover, following Kariso [25]. Garg has implemented theorem prover for BL_0 in ML [21], which we could integrate soundly by writing a type checker for the certificates it produces. Third, we could optimize Agda itself, by fixing some known inefficiencies in Agda's compile-time evaluation.

3.3 Computations

The monadic interfaces presented in Section 2 are currently treated as refinement types on Haskell's IO monad, which is exposed through the Agda foreign function interface. The implementations of proof-carrying file operations simply ignore their proof arguments. `fix` is compiled using general recursion in Haskell. In this operational model, programs written in Aglet adhere to the security policies, but no guarantees are made about programs that can access, e.g., the raw file system operations. We discuss alternatives in Section 5 below.

4. Related Work

Aglet implements security-typed programming in the style of Aura [24], PCML5 [9], Fine [38], and previous work by Avijit and Harper [8] (henceforth AH), which integrate authorization logics into functional/imperative programming languages. Our main contribution relative to these languages is to show how to support security-typed programming within an existing dependently-typed language. There are also some technical differences between these languages and ours:

First, Aura, PCML5, and AH interpret `says` as a lax modality, whereas BL_0 interprets it as a necessitation modality to support exclusive delegation; Fine uses first-order classical logic and does not directly support the `says` modality. The context-clearing necessitation modality is more challenging to represent than a lax modality.

Second, unlike these four languages, our language treats propositions and proofs as inductively defined data, which has several applications: In Aura, all proof-carrying primitives log the supplied proofs for later audit; the programmer could implement logged operations on top of our existing interface by writing a function `toString : Proof Γ A -> String` by recursion over proofs. Recursion over propositions is also essential for writing our theorem prover inside of Agda.

Third, our indexed monad of computations allows us to encode computation on behalf of a principal, following AH. In Aura, all computation proceeds on behalf of a single distinguished principal `self`. In PCML5, a program can authenticate as different principals, but the credentials are less precise: in PCML5, the program authenticates as k, whereas in AH the program acquires only the ability to `su` from a given k' to k—which may be a useful restriction if the program is subsequently no longer running as k'. Fine does not track authentication as a primitive notion, though it seems likely it could be encoded using an `As` predicate and affine types.

Fourth, in PCML5, `acquire` uses theorem proving to deduce consequences of the policy, whereas in our language `acquire` only tests whether a state-dependent atom or a statement by a principal is literally in the policy, and a separate theorem prover deduces consequences from the policy. We separate theorem proving from `acquire` so that we may also use the same theorem prover at compile-time to statically discharge proof obligations. PCML5 and AH make use of a theorem prover only at run-time, whereas Fine uses theorem proving only at compile-time.

Fifth, PCML5 is a language for spatially distributed authorization, where resources and policies are located at different sites on a network. We have shown how to support ML5-style spatial distribution using our indexed monad, but we leave spatial distribution of policies to future work.

Sixth, the operational semantics of both PCML5 and AH include a proof-checking reference monitor; we have not yet considered such an implementation.

Several other languages provide support for verifying security properties by type checking. For example, Fournet et al. [19] develop a type system for a process calculus, and Bengtson et al. [11] for F#, both of which can be used to verify authorization policies and cryptographic protocols. This work addresses important issues of concurrency, which we do not consider here. A technical difference is that, in their work, proofs are kept behind the scenes (e.g., in F7, propositions are proved by the Z3 theorem prover). In contrast, our language makes the proof theory directly available to the programmer, so that propositions and proofs can be computed with (for logging or run-time theorem proving) and so that proofs can be constructed manually when a theorem prover fails. Another example of a language that does not give the programmer direct access to the proof theory is PCAL [13], an extension of BASH that constructs the proofs required by a proof-carrying file system [23]; proof construction is entirely automated, but sometimes inserts run-time checks.

Our indexed monad was inspired by HTT [30]. RIF [12] also investigates applications of indexed monads to security-typed programming, but there are some technical differences: First, RIF is a new language where refinement types (using first-order classical logic) and a refined state monad are primitive notions, whereas we embed an authorization logic and an indexed monad in an existing dependently typed language. Second, RIF's monad is indexed by predicates on an explicit representation of the system state, whereas we index by policies Γ that describe an implicit ambient state.

Many security-typed languages address the problem of enforcing information flow policies (see Abadi et al. [4], Chothia et al. [15] for but a couple of examples). We follow Russo et al. [36], Swamy et al. [38] in representing information flow using an abstract type constructor (e.g., a monad or an applicative functor). Fable [37] takes a different approach to verifying access-control, information flow, and integrity properties, by providing a type of labelled data that is treated abstractly outside of certain policy portions of the program. This mechanism facilitates checking security properties (by choosing the labels appropriately and implementing policy functions) and proving bi-simulation properties of the programs that adhere to these policies.

DeYoung and Pfenning [16] describe a technique for representing access control policies and stateful operations in a linear authorization logic. Our approach to verifying context invariants, as in Section 2.1.4, is inspired by their work.

The literature describes a growing body of authorization logics [1, 2, 3, 17, 20, 21]. We chose BL_0 [21], a simple logic that supports the expression of decentralized policies and whose says connective permits exclusive delegation.

Appel and Felten [7] pioneered the use of proof-carrying authorization, in which a system checks authorization proofs at run-time. Several systems have been built using PCA [10, 23, 40]. Like many security-typed languages, we use dependently typed PCA to check authorization proofs at compile-time through type checking.

5. Conclusion

In this paper, we have described Aglet, a library embedding security-typed programming in a dependently-typed programming language. There are many interesting avenues for future work: First, we may consider embedding an authorization logic such as full BL [20] that accounts for resources that change over time. Second, we have currently implemented the monadic computation interface on top of unguarded Haskell IO commands, which provides security guarantees for well-typed programs. To maintain security in the presence of ill-typed attackers, we may instead implement our interface using a proof-carrying run-time system such as PCFS [23]. Following PCML5 [9], we may then be able to prove a progress theorem showing that well-typed programs always pass the reference monitor. Another intriguing possibility is to formalize the operational behavior of computations directly within Agda—e.g. using an algebraic axiomatization [35]. Third, in this paper we have shown examples of entirely static and entirely dynamic verification; we would like to consider examples that mix the two. This will require using reflection to represent Agda judgements as data, so that our theorem prover does not get stuck on open Agda terms. Fourth, we have shown a few small examples of using Agda to reason about the class of contexts that is possible given a particular monadic interface. In future work, we would like to explore ways of systematizing this reasoning (e.g., by using linear logic to describe transformations between contexts, as in DeYoung and Pfenning [16]). We would also like to use Agda to analyze global properties of a particular monadic interface (such as proving a principal can never access a resource). Once we have circumscribed the contexts generated by a particular interface, we can prove such properties by induction on BL_0 proofs. Fifth, we would like to implement more significant examples, such as a larger portion of ConfRM.

Acknowledgements We thank Frank Pfenning, Robert Harper, Kumar Avijit, Deepak Garg, and Rob Simmons for helpful discussions about this work. We thank Frank Pfenning, Robert Harper, and several anonymous referees for feedback on previous drafts of this article.

References

[1] M. Abadi. Access control in a core calculus of dependency. In *Internatonal Conference on Functional Programming*, 2006.

[2] M. Abadi. Variations in access control logic. In *International Conference on Deontic Logic in Computer Science*, pages 96–109. Springer-Verlag, 2008.

[3] M. Abadi, M. Burrows, B. Lampson, and G. Plotkin. A calculus for access control in distributed systems. *ACM Transactions on Programming Languages and Systems*, 15(4):706–734, September 1993.

[4] M. Abadi, A. Banerjee, N. Heintze, and J. G. Riecke. A core calculus of dependency. In *ACM Symposium on Principles of Programming Languages*, pages 147–160. ACM Press, 1999.

[5] T. Altenkirch and C. McBride. Generic programming within dependently typed programming. In *IFIP TC2 Working Conference on Generic Programming, Schloss Dagstuhl*, 2003.

[6] J.-M. Andreoli. Logic programming with focusing proofs in linear logic. *Journal of Logic and Computation*, 2(3):297–347, 1992.

[7] A. W. Appel and E. W. Felten. Proof-carrying authentication. In *ACM Conference on Computer and Communications Security*, pages 52–62, 1999.

[8] K. Avijit and R. Harper. A language for access control. Technical Report CMU-CS-07-140, Carnegie Mellon University, Computer Science Department, 2007.

[9] K. Avijit, A. Datta, and R. Harper. Distributed programming with distributed authorization. In *ACM SIGPLAN-SIGACT Symposium on Types in Language Design and Implementation*, 2010.

[10] L. Bauer, S. Garriss, J. M. Mccune, M. K. Reiter, J. Rouse, and P. Rutenbar. Device-enabled authorization in the Grey System. In *Proceedings of the 8th Information Security Conference*, pages 431–445. Springer Verlag LNCS, 2005.

[11] J. Bengtson, K. Bhargavan, C. Fournet, A. Gordon, and S. Maffeis. Refinement types for secure implementations. In *Computer Science Logic*, 2008.

[12] J. Borgström, A. D. Gordon, and R. Pucella. Roles, Stacks, Histories: A Triple for Hoare. Technical Report MSR-TR-2009-97, Microsoft Research, 2009.

[13] A. Chaudhuri and D. Garg. PCAL: Language support for proof-carrying authorization systems. In *Proceedings of the 14th European Symposium on Research in Computer Security*, September 2009.

[14] S. Chong, A. C. Myers, K. Vikram, and L. Zheng. Jif reference manual. Available from `http://www.cs.cornell.edu/jif/doc/jif-3.3.0/manual.html`, February 2009.

[15] T. Chothia, D. Duggan, and J. Vitek. Type-based distributed access control (extended abstract). In *Computer Security Foundations Workshop*, 2003.

[16] H. DeYoung and F. Pfenning. Reasoning about the consequences of authorization policies in a linear epistemic logic. In *Workshop on Foundations of Computer Security*, 2009.

[17] H. DeYoung, D. Garg, and F. Pfenning. An authorization logic with explicit time. In *IEEE Computer Security Foundations Symposium*, 2008.

[18] D. J. Dougherty, K. Fisler, and S. Krishnamurthi. Specifying and reasoning about dynamic access-control policies. In *International Joint Conference on Automated Reasoning*, pages 632–646. Springer,

2006.

[19] C. Fournet, A. D. Gordon, and S. Maffeis. A type discipline for authorization in distributed systems. In *Computer Science Logic*, 2007.

[20] D. Garg. *Proof Theory for Authorization Logic and its Application to a Practical File System*. PhD thesis, Carnegie Mellon University, 2009.

[21] D. Garg. Proof search in an authorization logic. Technical Report CMU-CS-09-121, Computer Science Department, Carnegie Mellon University, April 2009.

[22] D. Garg and F. Pfenning. Non-interference in constructive authorization logic. In *Computer Security Foundations Workshop*, pages 183–293, 2006.

[23] D. Garg and F. Pfenning. PCFS: A proof-carrying file system. Technical Report CMU-CS-09-123, Carnegie Mellon University, 2009.

[24] L. Jia, J. A. Vaughan, K. Mazurak, J. Zhao, L. Zarko, J. Schorr, and S. Zdancewic. Aura: A programming language for authorization and audit. In *ACM SIGPLAN International Conference on Functional Programming*, 2008.

[25] K. Kariso. Integrating Agda and automated theorem proving techniques. Talk at Dependently Typed Programming Workshop, 2010.

[26] S. Krishnamurthi. The CONTINUE server (or, How I administered PADL 2002 and 2003). In *International Symposium on Practical Aspects of Declarative Languages*, pages 2–16. Springer-Verlag, 2003.

[27] D. R. Licata and R. Harper. A monadic formalization of ML5. In *Pre-preceedings of Workshop on Logical Frameworks and Metalanguages: Theory and Practice*, July 2010.

[28] J. Morgenstern and D. R. Licata. Security-typed programming within dependently typed programming. Technical Report CMU-CS-10-114, Carnegie Mellon University, 2010.

[29] T. Murphy, VII. *Modal Types for Mobile Code*. PhD thesis, Carnegie

Mellon, January 2008. Available as technical report CMU-CS-08-126.

[30] A. Nanevski, G. Morrisett, and L. Birkedal. Polymorphism and separation in Hoare Type Theory. In *ACM SIGPLAN International Conference on Functional Programming*, pages 62–73, Portland, Oregon, 2006.

[31] A. Nanevski, G. Morrisett, A. Shinnar, P. Govereau, and L. Birkedal. Ynot: Reasoning with the awkward squad. In *ACM SIGPLAN International Conference on Functional Programming*, 2008.

[32] U. Norell. *Towards a practical programming language based on dependent type theory*. PhD thesis, Chalmers University of Technology, 2007.

[33] F. Pfenning and R. Davies. A judgmental reconstruction of modal logic. *Mathematical Structures in Computer Science*, 11:511–540, 2001.

[34] F. Pfenning and R. J. Simmons. Substructural operational semantics as ordered logic programming. In *IEEE Symposium on Logic In Computer Science*, pages 101–110, Los Alamitos, CA, USA, September 2009. IEEE Computer Society.

[35] G. Plotkin and M. Pretnar. Handlers of algebraic effects. In *European Symposium on Programming*, pages 80–94. Springer-Verlag, 2009.

[36] A. Russo, K. Claessen, and J. Hughes. A library for light-weight information-flow security in Haskell. In *ACM SIGPLAN Symposium on Haskell*, pages 13–24. ACM, 2008.

[37] N. Swamy, B. J. Corcoran, and M. Hicks. Fable: A language for enforcing user-defined security policies. In *IEEE Symposium on Security and Privacy*, pages 369–383. IEEE Computer Society, 2008.

[38] N. Swamy, J. Chen, and R. Chugh. Enforcing stateful authorization and information flow policies in Fine. In *European Symposium on Programming*, 2010.

[39] J. A. Vaughan, L. Jia, K. Mazurak, and S. Zdancewic. Evidence-

based audit. In *IEEE Computer Security Foundations Symposium*, June 2008.

[40] E. Wobber, M. Abadi, M. Burrows, and B. Lampson. Authentication in the Taos operating system. *ACM Transactions On Computer Systems*, 12(1):3–32, 1994.