Submitted to ICFP 2010

Security-Typed Programming within Dependently-Typed Programming

Jamie Morgenstern
University of Chicago
jamiemmt@uchicago.edu

Daniel R. Licata
Carnegie Mellon University
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. Most of these security-typed languages have been presented as new, stand-alone language designs. In this paper, we instead show how to embed a security-typed programming language within an existing dependently typed programming language, Agda. This language design strategy allows us to inherit both the metatheoretic properties, such as type safety, and the implementation of the host language. Our embedded language accounts for the major features of these existing security-typed programming languages, including decentralized access control, stateful and dynamic policies, spatially distributed and principaled computation, and information flow. Our embedding consists of the following ingredients: we represent the syntax and proofs of an authorization logic, Garg and Pfenning's BL_0, using dependent types, and implement a proof search procedure, based on a focused sequent calculus, to ease the burden of constructing proofs. Additionally, we represent computations as a monad indexed by pre- and post-conditions drawn from the authorization logic, which permits stateful policies that change during execution. Our work shows that a dependently typed language can be used to prototype a security-typed language with all of these features.

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? These languages enforce adherence to policies using a combination of compile-time and run-time techniques. For example, Aura [27] and PCML5 [8], enforce access control using proof-carrying authorization [7]: the run-time system requires every access to a sensitive resource be accompanied by a proof of authorization, while the type system aids programmers in constructing correct proofs. Other languages, such as Fable[38] and Jif[18], enforce information flow properties using type systems that restrict the use of values that depend on private information. Yet other languages, such as Fine[39], combine these techniques to enforce both.

However, the security-typed languages described in the literature are presented as new, stand-alone language designs. This has costs to the language designer, who must define and study a new language (e.g. the metatheoretic properties of Aura take 12,000 lines of Coq code to establish), to the language implementer, who must implement a new compiler (e.g., the prototype Fine compiler is comprised of approximately 15,000 of F# code), and to programmers, who must learn new languages and their infrastructures.

In this paper, we instead show how to embed a security-typed programming language within an existing dependently typed programming language, Agda [33]. This language design/implementation technique makes our implementation more concise, consisting of only 1400 lines of Agda code. Moreover, our language inherits standard metatheoretic results, such as type safety, from the host language. Finally, programmers may exploit existing tools, such as Agda's interactive proof editor.

Our embedded language accounts for the major features of existing security-typed programming languages:

Decentralized Access Control Access control policies are expressed as propositions in an *authorization logic* and enforced via a type system that mandates the existence of certain proofs. The authorization logic, Garg and Pfenning's BL_0 [21], 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.

Stateful 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 submint a paper, but only before the submission deadline. Fine accounts for stateful policies using a technique called affine types. However, affine types require a special substructural notion of variables, which Agda does not currently provide.

In this paper, we show that one can instead account for stateful policies using an indexed monad. Following Hoare Type Theory [32], 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. This can be represented by a computation that hase type $\bigcirc$ `(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.

Our language 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.

Spatially-Distributed and Principaled Computation Our indexed monad `◯ Γ A Γ'` subsumes other indexed notions of computation that have been used for security typed languages, such as spatially distributed computation as in PCML5, and computation on behalf of a principal, as in Avijit and Harper [8].

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 [39].

Compile-time and Run-time Theorem Proving Security-typed languages require a combination of static and dynamic verification. For example, when the access control policy of a system is known at compile-time, it is possible to statically verify that a program complies with it. In our language, programs can be verified against a statically-given policy by 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 depends upon the system state, or information in a database). These programs must dynamically test whether an operation is permitted by the policy before performing that operation.

Thus, relative to typical ML or Haskell code, the added cost of verifying security properties consists of (a) static proof annotations and (b) dynamic checks against the policy. To reduce this burden, we have implemented a certified theorem prover for our authorization logic, based on a focused sequent calculus. Our theorem prover can be run at compile-time to compute proof annotations—in which case failure to find a proof is a compile-time error. The same theorem prover can also be run at run-time to dynamically test whether an operation is allowed by a policy—in which case failure to find a proof is a possibility that must be handled at run-time. Our 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.

Our work shows that a dependently-typed language can be used to prototype a security-typed language, and provides an example of using a dependently-typed language to construct a domain-specific type system. We exploit the expressiveness of dependent types in many ways: we use indexed inductive definitions to statically check proofs, datatype-generic programming to implement syntactic operations such as weakening and substitution, type-level recursion over values to improve the conciseness of specifications, and reflective theorem proving to automatically discharge proof obligations.

The remainder of this paper is organized as follows: In Section 2, we show a variety of examples, including a file system interface, an example of spatially distributed programming with information flow, and a conference management server example adapted from Fine. In Section 3, we describe the we discuss the Agda embedding of our language, including the representation of the logic and the implementation of the theorem prover. In Section 4, we discuss some implementation details. Finally, we discuss related work in Section 5 and conclude in Section 6. The Agda code for this paper is available from `http://www.cs.cmu.edu/~drl`.

Agda We briefly review Agda's syntax, referring the reader to the Agda Wiki(`wiki.portal.chalmers.se/agda/`) for more introductory materials. Dependent function types are written with parentheses as `(x : A) → B`. An implicit dependent function space is written as `{x : A} → B` or `∀ {x} → B` — arguments to implicit functions are inferred. Non-dependent functions are written as `A → B`. Anonymous functions are written as `λ x → e`. Named functions are defined by clausal pattern-matching definitions. Let bindings are written as `let x = e1 y = e2 ... in e`, and lists are constructed by `[]` and `::` (note that `:` is used for type annotations). Agda supports infix and mix-fix operators, which we will use heavily in our embedded logic. `Set` is the classifier of classifiers in Agda, like the kind `type` in ML or Haskell.

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

Figure 1. Sample access control policy

2. Examples

Security-typed languages provide a variety of mechanisms for enforcing security policies, and the best methodologies for applying these tools in building larger applications are not yet understood. In this section, we consider several small examples, which demonstrate that our language provides the type-theoretic tools necessary to implement the benchmark examples considered in the literature. We leave a careful study of the best ways of deploying these tools in practice to future work.

2.1 File IO with Access Control

2.1.1 Policy

We introduce BL_0 access control policies with the example in Figure 1. This policy should be read as follows: 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 calls to operations like reading a file. Second, the system administrator says Jamie owns secret.txt. Second, 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, where policies are comprised of the statements by a variety of different principals about the resources they control. Various definitions of says have been studied [1, 2, 3, 7, 8, 16, 21, 23, 24, 26, 27, 29].

For the principal Dan to read secret.txt, it will be sufficient to deduce the goal $\mathsf{Admin\ \ says\ \ mayread(Dan, secret.txt)}$. 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 certainly not the case that every principal believes that every statement of every other principal is *true*. Thus, the goal can be proved by taking the policy stated by Admin $(\forall r.\forall o.\forall f. \ldots)$, instantiating the quantifiers, and using the other statements in the policy to satisfy the preconditions.

The above policy is rendered in Agda 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))))) ::
    []
```

Similarly, the second clause expresses a policy 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 for details).

Next, we define a context representing a particular file system state. This context includes all the employee, ownership, and read 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 (pfCtx Γall ⊢ goal)
proof?  = prove 15

theProof : pfCtx 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`. Proofs are represented by the Agda type θ `⊢ A`, where θ is a context and `A` is a proposition. The function `pfCtx : TCtx+ [] → Ctx` coerces a list of propositions (abbreviated `TCtx+ []`) such as `Γall` into a context (see Section 3 for more on the context structure of the logic). 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 `Maybe A → A`, provided that the `Maybe A` is known to be of the form `Some x`. The term `proof?` is (definitionally) equal to `Some p` in Agda; in general, a call to the theorem prover on a context and a proposition that has no free Agda variables will always be equal to either `Some p` or to `None`.

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 postcondition is a function from `A`'s to contexts, modelling the fact that the postcondition can depend on the result of the computation (see `create` below). The generic operations are typed as follows: The postcondition of `return` is the same as its precondition, because it does not modify the state of the world. Bind (>>=) chains together two computations, where the postcondition of the first is the precondition of the second, for all results of the computation. 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. In this interface, primitives like `getLine` (reading a line of input) and `print` do not change the state and do not require proofs. The postcondition of `error` can be arbitrary, as it will never be reached. The remaining computations are defined as follows:

Read The function `read` takes a principal `k` and a file `f`, along with a proof that the principal may read the file (`Mayread(k,f)`) and that the computation is running as the principal (`As(k)`). the proof must be given in the context Γ that is the precondition of the computation, ensuring that the proof is valid in the current state of the world. `read` returns a `String`, and leaves the state unchanged. The proof argument ensures that well-typed programs adhere to the authorization policy. Additionally, these proofs may be logged for later audit by system administrators [27].

An example call to `read` looks like this:

```
jread : ◯ (Γall as "Jamie") String (λ _ → (Γall as "Jamie"))
jread = read (Prin "Jamie") (File "secret·txt")
             (solve (prove 17))

jreadprint : ◯ (Γall as "Jamie") Unit (λ _ → (Γall as "Jamie"))
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 ensures at compile-time that the required fact is derivable from the policy. The computation `jreadprint` first 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.

For an example call to `chown`, we define a context `Γstate'` that is the result of replacing the fact that `Jamie` owns `secret.txt` with `Dan` owning that file. Next, we do a call to `chown` that, running as `Dan`, changes the owner of the file from `Dan` to `Jamie`, and then we do a computation as `Dan` named `dreadprint` that reads the file. `dreadprint` is defined below. `proveReplace` is a tactic for proving that Γ' is Γ with the ownership of `secret.txt` changed. `solve (prove 15)` calls the theorem prover (at compile-time) to prove the current user has permission to `chown secret.txt`.

```
Γstate' =  replace {_} {Γstate}
  (Prin "System" says
    (a- (Owner · (Prin "Dan" , File "secret·txt"))))
  i0

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))
         >> dreadprint
```

Sudo We now give a well-typed version of the Unix command `sudo`, which allows a principal to run a computation as another principal. 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 have a post-condition saying that it leaves the state running as `k2`, and then the overall computation returns to running as `k1`. However, since 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:

```
dreadprint : ◯ (Γall as "Dan") Unit (λ _ → Γall as "Dan")
dreadprint = 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 replacing `As(Prin "Jamie")` with `As(Prin "Dan")` (in both directions). Our tactic `proveReplace` proves of these equalities. Additionally, a call to the theorem prover (`solve(prove 15)`) proves `MaySu · (Prin "Dan" , Prin "Jamie")`.

Acquire The function `acquire` allows a program to check whether a proposition is true in the state of the world. For example, a call to `acquire` might check to see if `System says owns(Prin "Jamie" , File "secret.txt")`. 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⇒ print "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 we would know to be true after running `jreadprint`. 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 about which contexts Γ should be allowed. For example, the above type for `chown` would have unexpected consequences if there ever were more than one owner of a file, or more than one copy of `System says owns(k,f)` in the 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 the predicate `Good` Γ. The intended invariant of the interface is that a monadic computation `◯ Γ A Γ'` should have the property that Γ' is good if Γ is. To achieve this, the weakening operations and `acquire` require preconditions that certain contexts are `Good`, and the programmer must verify that operations such as `read`, `chown`, and `sudo` preserve goodness. This establishes that all computations provided by the interface preserve goodness, so it is not necessary to make each monadic operation require a proof that the precondition is `Good`—when writing a particular program, the programmer needs only to verify that the initial policy and those in calls to weakening and `acquite` satisfy the invariants.

In the above examples, we took `Good` to be the trivially true invariant, so the proofs could be elided with an `_`. A simple but nontrivial invariant that one may wish 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 : _)
  (pfRep : (Replace (Prin "System" says (a-(Owner · (k1 , f))))
                    (Prin "System" says (a-(Owner · (k2 , f))))
                    Γ Δ))
  → Good Γ → Good Δ
```

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

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

2.2 File IO with Access Control and Information Flow

Next, we extend the above file signature with information flow, adapting an example from Swamy et al. [39]. First, we define a type `Tracked A L` which represents a value of type `A` tracked with security level `L`. 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 of course 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)
```

Note that 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 the string to be written 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 `file1.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

Next, we show how to account for spatial distribution as in ML5 [31], which 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 [30], we have shown how to build an ML5-like type system on type 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 presented 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. Note that 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.

Instead of (or in addition to) using information flow for `get`, we could use the logic to restrict `get`, so that e.g. only certain values may flow from one world to another.

2.4 Continue: A Conference Management System

Next, we adapt a formalization of the Continue conference management server [28] from Dougherty et al. [17], Swamy et al. [39] We show an excerpt of an authorization policy for Continue, which specifies who may perform *actions*—submit a paper, submit a review, change the phase of the conference (e.g., submission, notification, etc). We give a proof-carrying monadic interface to the computations which perform actions, and we show the main event loop of the server.

2.4.1 Policy

We formalize the policy for Continue 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 Continue's policy as a collection of Horn clauses, which are simple to translate to our logic; e.g.

```
Γpolicy : TCtx+ []
Γpolicy =
  ((∀e principal · ∀e string ·
      let
        author = ▷ iS i0
        paper = ▷ i0
      in
       (((a- (InPhase · (Submission))) ∧
         ((a- ( InRole · (author  , Author)))))
        ⊃ (a- (May · (author , (Submit · paper))))))) ::
  ((∀e principal · ∀e paper ·
     let
       reviewer = ▷ (iS i0)
       paper = ▷ i0
     in
      (((a- (InPhase · (Reviewing))) ∧
        ((a- (Assigned · (reviewer , paper)))))
       ⊃
       (a- (May · (reviewer , (Review · paper) )))))) :: []
```

The first 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. The second proposition reads: for all reviewers and papers, if the conference is in the reviewing phase and the principal is assigned to review a paper, then they may review the paper.

We have also begun to reformulate the policy using the `says` modality. For example, we may describe a policy which allows authors to share their paper scores with their coauthors:

```
Γsays : TCtx+ []
Γsays =
 ((∀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 `coathor`, 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 an individual computation 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. 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, an action to perform, and some `ExtraArgs` for the that action, along with a proof that the computation is running as the principal, and that the principal may perform that action. 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 `Submit`ing 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 must be accompanied by their contents, represented as a term 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 4 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 the papers' reviews and contents, while submit produces a `Term [] paper`, i.e. 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.

```
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)
```

`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 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.

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)
```

These dynamic tests are necessary because the policy is not known statically.

2.4.3 Server Main Loop

In Figure 3 we show the code for the main loop of the Continue server, implemented using the interface described above. The main loop serves request made by a principal who wishes to perform an action. Our proof-carrying interface ensures that this code adheres to the authorization policy described above: the action will only be executed if the principal is authorized to do so. The 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. 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

```
 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. Continue Main Loop

not worthwhile to track the postconditions). `main` is defined by taking the fixed point of the auxillary 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 runtime 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

Here we describe an example of dynamic policy acquisition: 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 (see Figure 4). This is simple in a dependently typed language because *contexts themselves are data*.

```
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 (List.fold [] (λ x → λ y →
                  parseReviewers x ++ y) asgn)

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

Figure 4. Continue Policy Acquistion

The function `getReviewerAsgn` takes a string, representing a path to the database, and returns the list of 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. The actual Continue implementation would read a variety of other propositions from the database as well (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. Embedding BL$_0$

BL$_0$ [21] extends first-order intuitionistic logic with the modality k says A. While a variety of definitions of says have been studied [1, 2, 3, 4, 7, 8, 9, 10, 16, 21, 23, 24, 25, 26, 27, 29], in BL$_0$, says is treated as a necessitation ($\Box$) modality, and *not* as a lax modality (i.e. a monad) [1, 8, 24, 27]. 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 payroll says $\forall t.(\textsf{HR says employee}(t)) \supset \textsf{MayBePaid}(t)$. 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 Terms and Types

In the above examples, we used a variety of atomic propositions (`Mayread`, `Owns`, etc.) about a variety of 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 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 Continue illustrates what programmers write to define an individual example:

```
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

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

3.2 Propositions

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

Propositions	A, B, C	::=	$P \mid A \wedge B \mid A \vee B \mid A \supset B \mid \top$
			$\mid \bot \mid \forall x : \tau.s \mid \exists x : \tau.A \mid k$ says A

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, there are the shifts $\downarrow$ and $\uparrow$, which include negative into positive and vice versa; we have supressed the shifts up to this point in the paper. The remaining datatype constructors correspond to the various ways of forming propositions in the above grammar. For example, the $\_ \wedge \_$ constructor takes two terms of type `Propo+` Ω and returns a term of type `Propo+` Ω. The constructor $\exists$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.

3.3 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, 34]. 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:

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

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

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

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 the facts in it may depend on claims by the principal k_0, which are not valid when reasoning as Alice. 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 [35] 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 $k$says$A$ is as follows: A is negative, but $k$says$A$ itself is positive. As a simple check on this, observe that k 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 says $A_1, \ldots k_n$ says A_n, with a goal of the form k' says B. It is necessary to use `claimsL` to turn all propositions of the form k 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 [34] 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 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 [35] with the focusing interpretation of says described above. The interested reader can find the inference rules for these judgements in Appendix A.

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 `r`Ω. We define several helper functions on `Ctx`s: `sayCtx` clears the `Ctx` of true propositions, and changes the principal we are reasoning on behalf of to its argument. `ictx` (not shown) projects the Ω from a `Ctx`—it is shorthand for `Ctx.r`Ω. `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 remaining pieces of the context as necessary.

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 weaken terms of type `X` Ω into terms of type `X` Ω'. We have defined weakening functions for many of the types indexed by Ω: terms (`weakenTerm`), propositions, claims, true contexts (`weakenT+`), claims contexts (`weakenC`), ...

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 $\forall$`L` (for the left focus judgement), $\exists$`L` and `saysL` (for the left inversion judgement), `saysR` (for the right focus judgement), and `claimsL` (for the neutral sequent judgement). For the most part, the rules are a straightforward transcription of the rules in Appendix A. In $\forall$`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 $\exists$`L`, it is necessary to weaken the goal with the new variable, which is tacit in on-paper presentations.

```
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 k0 A}
          -> (sayCtx θ k0) ⊢ A
          -> θ ⊢R (k0 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)

3.4 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 `θ ⊢ A` with at most the given depth. The prover is *certified*: when it says a proof exists, it actually returns the 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; the prover essentially 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 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. We leave a more sophisticated approach based on unification to future work.

4. Implementation Details

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. The monadic interfaces presented in Section 2 are implemented by foreign function calls to Haskell's IO monad.

The theorem prover is fairly slow, but it suffices to get tolerable compile times on the small examples we have considered so far:

Example	Policy clauses	Prover calls	TC time
FlowReadWrite	9	5	13s
Broker	5	1	3s
ReadDelegate	11	7	13s
Continue Loop	0 (dynamic)	0	6s

The table lists the number of clauses in the policy, the number of compile-time calls to the theorem prover, and the type checking time on a 2GHz Macbook with 3GB RAM, running Agda 2.2.7.

As a larger example, we are in the process of reformulating Continue's authorization policy in our language, along the lines described above. We anticipate that running the prover at run-time in this example will require some optimization, due to the size of the policy. We plan to implement unification, which will eliminate much of the branching from quantifiers, and to do a better job of clause selection. Another possibility would be to interface with Garg's ML theorem prover for BL_0, which is quite fast, by writing a type checker for the certificates it produces [22]. A final possibility would be to optimize Agda itself—we could improve the speed of compile-time calls to the theorem prover by fixing some known inefficiencies in Agda's compile-time evaluation, such as a lack of sharing in the term representation.

5. Related Work

The languages most closely related to ours are the security-typed programming languages Aura [27], PCML5 [9], Fine [39], 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. Necessitation, which manipulates the context, makes BL_0 a more challenging logic to represent than lax logic. 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 : ∀ A. Proof 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 subsumes the notion of computation on behalf of a principal in 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 indexed monad permits more precise credentials: rather than acquiring the credentials to be k, one acquires only the ability to `su` from a given k' to k. Fourth, in Aura, the authorization policy must be encoded statically as constants in the program, except for statements by the principal `self`, which can be proved dynamically by a command `say`. In Fine, a policy can refer to dynamic data, but the policy itself must be given statically. Our language follows AH and PCML5 in only requiring the authorization policy to be known at run-time, providing a function `acquire` for querying the policy. Fifth, in PCML5, `acquire` uses theorem proving to deduce consequences of the policy, whereas in our language `acquire` only tests whether a fact is literally in the policy, and a separate theorem prover deduces consequences from the policy. We take this approach 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. Sixth, 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. The operational semantics of both PCML5 and AH include a proof-checking reference monitor; we have not yet considered such an implementation.

Abadi [1] discusses some examples of security-typed programming within DCC. Whereas Abadi [1] uses the same λ-calculus both as the programming language and as the proof terms for the authorization logic, we separate a programming language (Agda) from an embedded authorization logic (BL_0), which provides the freedom to use, e.g., a modal or linear authorization logic within an unrestricted programming language.

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. [12] for F#, both of which can be used to verify authorization policies and cryptographic protocols. 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 [25]; proof construction is entirely automated, but sometimes inserts run-time checks.

Many security-typed languages address the problem of enforcing information flow policies (see Abadi et al. [4], Chothia et al. [14] for but a couple of examples). We follow Russo et al. [36], Swamy et al. [39] 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. In this work, our correctness criterion is the more modest goal of simply ensuring that all accesses to a protected resources are accompanied by a proof of authorization, which can be accomplished simply using dependent types to represent proofs.

DeYoung and Pfenning [15] 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, 3, 4, 7, 8, 16, 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 [11, 25, 40]. Security-typed languages instead check authorization proofs *at compile-time* through type checking, and thus can be used to write code that will definitely pass the reference monitor of a PCA system.

6. Conclusion

In this paper, we have embedded a security-typed programming language in a dependently-typed programming language. Our language accounts for the major features of existing security-typed programming languages such as decentralized access control, information flow, spatially distributed and principaled computation, stateful and dynamic policies, and compile-time and run-time theorem proving. Our embedding consists of: a representation of the authorization logic BL_0, which permits decentralized access control policies, using dependent types, so that Agda's type checker can be reused to statically check the correctness of proofs; a proof-producing theorem prover, which can be used to discharge proof obligations both at compile-time and at run-time; and an indexed monad of computations, used to type effectful, proof-carrying operations that may modify stateful policies.

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[25], in which case our type system would ensure the reference monitor would never reject a proof. Third, 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 [15]). 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. On the more practical side, we are currently in the process of implementing a more significant portion of the Continue access policy and server. In order to utilize the larger policy, we plan to improve the efficiency of the theorem prover, as described above.

Acknowledgements We thank Frank Pfenning for advice about this project and feedback on a previous draft of this article. We thank many other members of Carnegie Mellon's PoP Group, especially Kumar Avijit, Deepak Garg, Rob Simmons, and Bob Harper, for helpful discussions about this work. We thank several anonymous referees for feedback on a previous version 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, 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. URL `citeseer.ist.psu.edu/abadi93calculus.html`.

[4] M. Abadi, A. Banerjee, N. Heintze, and J. G. Riecke. A core calculus of dependency. In *In Proc. 26th ACM Symp. on Principles of Programming Languages (POPL*, 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. Available from `http://www.cs.cmu.edu/∼rwh`, 2009.

[10] L. Bauer, M. A. Schneider, E. W. Felten, and A. W. Appel. Access control on the web using proof-carrying authorization. In *3rd DARPA Information Survivability Conference and Exposition*, pages 117–119. IEEE Computer Society, 2003.

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

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

[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 (ESORICS '09)*, September 2009.

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

[15] H. DeYoung and F. Pfenning. Reasoning about the consequences of authorization policies in a linear epistemic logic. Technical report, CMU, 2009.

[16] H. DeYoung, D. Garg, and F. Pfenning. An authorization logic with explicit time. Technical Report CMU-CS-07-166, Computer Science Department, Carnegie Mellon University, December 2007.

[17] D. J. Dougherty, K. Fisler, and S. Krishnamurthi. Specifying and reasoning about dynamic access-control policies. In *of Lecture Notes in Computer Science*, pages 632–646. Springer, 2006.

[18] A. M. et. al. Jif reference manual. http://www.cs.cornell.edu/jif/doc/jif-3.0.0/manual.html, June 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. Proof search in an authorization logic. Technical report, CMU, 2009.

[23] D. Garg and M. Abadi. A modal deconstruction of access control logics. In *Software Science and Computation Structures (FoSSaCS 2008)*. Springer Verlag, April 2008. URL `http://www.cs.cmu.edu/ dg/publications.html`.

[24] D. Garg and F. Pfenning. Non-interference in constructive authorization logic. In *Proceedings of the 19th IEEE Computer Security Foundations Workshop (CSFW 19)*, 2006.

[25] D. Garg and F. Pfenning. Pcfs: A proof-carrying file system. *Technical Report CMU-CS-09-123*, 2009.

[26] Y. Gurevich and I. Neeman. Dkal: Distributed-knowledge authorization language. In *Proceedings of the 21st IEEE Symposium on Computer Security Foundations (CSF-21), 2008.* Springer Verlag, 2008.

[27] 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.

[28] S. Krishnamurthi. The continue server (or, how i administered padl 2002 and 2003). In *PADL '03: Proceedings of the 5th International Symposium on Practical Aspects of Declarative Languages*, pages 2–16, London, UK, 2003. Springer-Verlag. ISBN 3-540-00389-4.

[29] B. Lampson, M. Abadi, M. Abadi, M. Burrows, and E. Wobber. Authentication in distributed systems: Theory and practice. *ACM Transactions on Computer Systems*, 10:265–310, 1992.

[30] D. R. Licata and R. Harper. A monadic formalization of ML5. Available from `http://cs.cmu.edu/~drl`, April 2010.

[31] T. Murphy, VII. *Modal Types for Mobile Code*. PhD thesis, Carnegie Mellon, January 2008. URL `http://tom7.org/papers/`. Available as technical report CMU-CS-08-126.

[32] 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.

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

[34] F. Pfenning and R. Davies. A judgmental reconstruction of modal logic. In *Mathematical Structures in Computer Science*, page 2001, 1999.

[35] 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.

[36] A. Russo, K. Claessen, and J. Hughes. A library for light-weight information-flow security in haskell. In *Haskell '08: Proceedings of the first ACM SIGPLAN symposium on Haskell*, pages 13–24, New York, NY, USA, 2008. ACM. ISBN 978-1-60558-064-7. doi: http://doi.acm.org/10.1145/1411286.1411289.

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

[38] N. Swamy, B. J. Corcoran, and M. Hicks. Fable: A language for enforcing user-defined security policies. In *SP '08: Proceedings of the 2008 IEEE Symposium on Security and Privacy*, pages 369–383, Washington, DC, USA, 2008. IEEE Computer Society. ISBN 978-0-7695-3168-7. doi: http://dx.doi.org/10.1109/SP.2008.29.

[39] N. Swamy, J. Chen, and R. Chugh. Enforcing stateful authorization and information flow policies in fine. In *In Proceedings of the European Symposium on Programming (ESOP)*, 03 2010.

[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.

A. Appendix: Weakly focused sequent calculus for $\mathbf{BL}_0$

$$\dfrac{}{\Omega;\Delta;\Gamma;P^{+}\xrightarrow{k}[P^{+}]}\ \textsc{init+}\qquad\dfrac{}{\Omega;\Delta;\Gamma,[P^{-}]\xrightarrow{k}P^{-}}\ \textsc{init-}$$

$$\dfrac{}{\Omega;\Delta;\Gamma\xrightarrow{k}[\top]}\ \top\qquad\dfrac{\Omega;\Delta;\Gamma\xrightarrow{k}C}{\Omega;\Delta;\Gamma\xrightarrow{k}\top>_I C}\ \top L\qquad\dfrac{}{\Omega;\Delta;\Gamma,\xrightarrow{k}\bot>_I C}\ \bot$$

$$\dfrac{\Omega;\Delta;\Gamma\xrightarrow{k}[A]\qquad\Omega;\Delta;\Gamma\xrightarrow{k}[B]}{\Omega;\Delta;\Gamma\xrightarrow{k}[A\wedge B]}\ \wedge\mathrm{R}\qquad\dfrac{\Omega;\Delta;\Gamma,A,B\xrightarrow{k}C}{\Omega;\Delta;\Gamma\xrightarrow{k}(A\wedge B)>_I C}\ \wedge\mathrm{L}$$

$$\dfrac{\Omega;\Delta;\Gamma\xrightarrow{k}[A]}{\Omega;\Delta;\Gamma\xrightarrow{k}[A\vee B]}\ \vee\mathrm{R1}\qquad\dfrac{\Omega;\Delta;\Gamma\xrightarrow{k}[B]}{\Omega;\Delta;\Gamma\xrightarrow{k}[A\vee B]}\ \vee\mathrm{R2}\qquad\dfrac{\Omega;\Delta;\Gamma,A,\xrightarrow{k}C\qquad\Omega;\Delta;\Gamma,B,\xrightarrow{k}C}{\Omega;\Delta;\Gamma\xrightarrow{k}(A\vee B)>_I C}\ \vee\mathrm{L}$$

$$\dfrac{\Omega\vdash t:\tau\qquad\Omega;\Delta;\Gamma\xrightarrow{k}[[t/x]A]}{\Omega;\Delta;\Gamma\xrightarrow{k}[\exists x:\tau.A]}\ \exists\mathrm{R}\qquad\dfrac{\Omega,x:\tau;\Delta;\Gamma,A\xrightarrow{k}C}{\Omega;\Delta;\Gamma\xrightarrow{k}(\exists x:\tau.A)>_I C}\ \exists\mathrm{L}$$

$$\dfrac{\Omega;\Delta;\Gamma\xrightarrow{k}A^{-}}{\Omega;\Delta;\Gamma\xrightarrow{k}[\downarrow A^{-}]}\ \textsc{blur}\mathrm{R}\qquad\dfrac{\Omega;\Delta;\Gamma,\downarrow A^{-}\xrightarrow{k}[A^{-}]>C}{\Omega;\Delta;\Gamma,\downarrow A^{-}\xrightarrow{k}C}\ \textsc{lfoc}$$

$$\dfrac{\Omega;\Delta;[]\xrightarrow{k}A^{-}}{\Omega;\Delta;\Gamma\xrightarrow{k0}[k\ \mathsf{says}\ A^{-}]}\ \textsc{says}\mathrm{R}\qquad\dfrac{\Omega;\Delta,(k\ \mathsf{claims}\ A^{-});\Gamma\xrightarrow{k0}C}{\Omega;\Delta;\Gamma\xrightarrow{k0}(k\ \mathsf{says}\ A^{-})>_I C}\ \textsc{says}\mathrm{L}$$

$$\dfrac{\Omega;\Delta;\Gamma,A\xrightarrow{k}B}{\Omega;\Delta;\Gamma\xrightarrow{k}A\supset B}\ \supset\mathrm{R}\qquad\dfrac{\Omega;\Delta;\Gamma\xrightarrow{k}[A]\qquad\Omega;\Delta;\Gamma\xrightarrow{k}[B]>C}{\Omega;\Delta;\Gamma\xrightarrow{k}[A\supset B]>C}\ \supset\mathrm{L}$$

$$\dfrac{\Omega;\Delta;\Gamma\xrightarrow{k}[t/x]A}{\Omega;\Delta;\Gamma\xrightarrow{k}\forall x:\tau.A}\ \forall\mathrm{R}\qquad\dfrac{\Omega;\Delta;\Gamma\xrightarrow{k}[[t/x]A]>C}{\Omega;\Delta;\Gamma\xrightarrow{k}[\forall x:\tau.A]>C}\ \forall\mathrm{L}$$

$$\dfrac{\Omega;\Delta;\Gamma\xrightarrow{k}[A^{+}]}{\Omega;\Delta;\Gamma\xrightarrow{k}\uparrow A^{+}}\ \textsc{rfoc}\qquad\dfrac{\Omega;\Delta;\Gamma,A^{+}\xrightarrow{k}C}{\Omega;\Delta;\Gamma\xrightarrow{k}[\uparrow A^{+}]>C}\ \textsc{blur}\mathrm{L}$$

$$\dfrac{\Omega;\Delta;(\Gamma,A^{+})\xrightarrow{k}[A^{+}]>_I C}{\Omega;\Delta;(\Gamma,A^{+})\xrightarrow{k}C}\ \textsc{linv}\qquad\dfrac{\Omega;(\Delta,k\ \mathsf{claims}\ A^{-});\Gamma\xrightarrow{k0}[A^{-}]>C\qquad k0\geq k}{\Omega;(\Delta,k\ \mathsf{claims}\ A^{-});\Gamma\xrightarrow{k0}C}\ \textsc{claims}\mathrm{L}$$