A language for access control

Kumar Avijit **Robert Harper**

July 2007
CMU-CS-07-140

School of Computer Science
Carnegie Mellon University
Pittsburgh, PA 15213

Abstract

We present a language for access control. The language is organized around the notion of execution *on behalf of* a principal. This is characterized using an indexed lax modality. Central to the language is the idea of manifest security – accessing a resource requires presenting a proof of accessibility to the resource monitor. Proofs are generated at runtime by actions such as typing in password, looking up an access-control list or by composing other proofs etc. In the present work, we consider a simplified setting in which the access-control theory is static. In such a case proofs can be regarded as static entities. Proof generation can be hoisted away from resource access since proofs become permanent. Also, the actual proofs are irrelevant. The results of runtime checks can therefore be reflected as types and the program can be verified statically to ensure that relevant runtime checks would be passed before accessing any resource. We prove a theorem stating that the language is safe in terms of how all a principal can get to access a resource.

Keywords: Access control, modality, effectful computations, phase separation.

1 Introduction

This paper presents a programming language for access control. The language is based on the idea of *manifest security* wherein access to resources is granted subject to the demonstration of a proof of accessibility. Operations concerned with accessing resources are parameterized by suitable proofs that vouch for the safety of execution of the operation.

Access-control deals with controlling access to sensitive resources/data in adherence to a policy. In order to talk about access, we first need to have a notion of a principal trying to access a resource. Our language allows directly expressing the principal *on behalf of which* a computation is to be executed. Principals are introduced at a sufficiently high level of abstraction. The principals are treated like nominals and the language does not associate any property with them except that they be different from other syntactic entities in the language and the resources in the system. For example, in an operating system setting, principals might correspond to users or programs running on behalf of users. In a π-calculus, the channels might be thought of as principals. Abstractly, a principal is any agent that is responsible for the execution of certain programs or that receives the results of such programs.

We assume that the system consists of a fixed number of principals and a fixed number of resources, both of which are known statically. Principals interact with the resources using resource-specific operations. In our prototype language, we have considered reference cells as an example of resources. The concept of manifest security can be extended to other kinds of resources.

The language distinguishes effectful computations, such as reading or writing a reference cell, from pure lambda terms. This distinction is made because access control is primarily concerned with controlling the effects produced by principals, e.g. writing to a file. Pure terms can be executed alike by all principals. Effectful computations (or simply, computations) are always executed on behalf of a principal. The particular principal executing a computation affects the results of the computation. Pure terms on the other hand, evaluate to the same value, no matter which principal executes them.

The language allows a principal to assume the identity of another principal. Such a capability is often useful in scenarios where a principal wishes to execute a computation with downgraded permissions, for instance when following the principle of least privileges. Principals may, for example, be roles and a user should be allowed to switch between them by providing their respective passwords. Entering the correct password generates a proof of reachability from the principal, say $\mathbf{w}_1$, who typed in the password to the principal , say $\mathbf{w}_2$, whose password has been typed in. This proof can then be used by $\mathbf{w}_1$ to switch to $\mathbf{w}_2$.

Now we give a high level overview of the manifest security architecture. This provides the scheme of things this language is supposed to fit in.

1.1 The overall architecture

Manifest security [CHP$^+$07] proposes a new architecture for building secure extensible systems. The proposal targets software that can be customized using third party extensions. Issues like access-control and information flow control feature predominantly in such applications. The proposal addresses these issues in two major phases: (a) building a logic in which to specify security policies; (b) building a runtime system and a programming language that can be used to write secure programs. These are described below in more detail:

1. **Access-control policy** The language assumes the existence of an access-control theory specified as a set of axioms and rules of derivation. The rules of derivation come from a logic of access control that describes the access-control policy. We do not intend to build such a logic in this paper. Instead, we work with a very simple logic that contains only the hypothesis rule of inference. This logic essentially mimics access-control lists, which can be viewed as binary predicates on principals and resources, specifying for each tuple, whether an access is allowed. Important issues arise while considering richer logics in connection with our language because the proofs from the logic need to be reflected as types in the language. Section 2.1 discusses these issues with the help of an example logic.

2. **Proof-carrying runtime system** The runtime system acts as a bridge between the programming language and the access-control logic. It is responsible for handling requests for proving a proposition using the access-control theory and for requests to access a resource. Each resource is associated with a reference monitor. All accesses

to the resource pass through the reference monitor which validates the proofs of accessibility before allowing access to the resource. The accessibility proofs are provided along with the request for access.

3. **The security language** The security language provides static guarantees to the effect that proper proofs would be passed to the resource monitor for accessing a resource. This serves twofold purposes: in a setting where the access-control theory is not mutable, the resource monitors no longer need to check for the validity of the proofs. Proofs can therefore be safely erased from the language. In cases where the axioms of the theory may change along the execution of the program, the resource monitors still do not need to validate the entire proof. Only the axioms appearing at the top level of a proof need to be checked. The validity of the rest of the proof is guaranteed by the language.

   Secondly, and more importantly, the type system facilitates a separation between proof generation and proof use, thereby permitting reuse of proofs. To best motivate this, consider for an analogy with array bounds checking, the following code that accesses elements of an array. Before reading off the element at a particular index, it verifies that the index is not out-of-bounds. This can be done in different ways.

   - **Explicitly comparing the index with the length of the array**

     ```
     if (Array.length array) > index
     then
        (* access the element at index *)
        let x = Array.sub (array, index)
        in
        ...
        end
     else
        ...
     ```

     Such a method is not amenable to static analysis. This is because it cannot be statically verified that the program fragment in the `if` condition does all the proper checks required in the `then` clause. In other words, what stops the programmer from writing the code of the then branch in the else branch instead?

     What is lacking in the above paradigm is the connection between checking for a condition on one hand and using the fact that the condition is true or false on the other. The `if` statement should really be looked as generating the proof/refutation of a proposition [Lof96] and passing it to the `then`/`else` branch.

   - **Combining array bounds checking with subscripting** A radical change would be to use a subscripting operation, say `safesub` instead of `sub`, that checks for the array bounds before accessing any index in the array. This would be rather wasteful in the case when the same index is used to subscript the array multiple times.

   - **Explicit proofs using dependent types** This approach remedies the shortcomings discussed in the first example code, using the richer framework of dependent types (see [LH05] for an exposition of this technique). The runtime bounds-check generates a proof that the subscript is less than the length of the array.

     $$\mathtt{nat} \;=\; \exists i :: \mathtt{NAT.Nat}(i)$$

     $$< \;\; : \; \Pi m : \mathtt{nat}.\Pi n : \mathtt{nat}.\mathtt{lt}(\mathtt{static}(m), \mathtt{static}(n))\ \mathtt{option}$$

     $\mathtt{lt}(i, j)$ is the type of proofs that the static index i is less than j. The array subscripting operation `sub` requires a proof saying that the subscript is within bounds. It would be typed as:

     $$\mathtt{sub} \;:\; \Pi m : \mathtt{nat}.\Pi n : \mathtt{nat}.\mathtt{array}[m] \rightarrow \mathtt{lt}(\mathtt{static}(n), \mathtt{static}(m)) \rightarrow \mathtt{int}$$

     The use of explicit proofs as witnesses that certain checks have been passed allows for a separation between the generation and use of these proofs. The code that does the runtime check can then be written anywhere in the program and the proof generated by it be passed to the operation that uses the proof. This facilitates reusing the same proof without having to generate it each time. Our approach borrows this idea of representing results of runtime computations as types. Checking whether a principal is allowed access to a particular resource may involve a runtime computation, such as reading an access-control list, or requesting a server for credentials. This corresponds to evaluating an integer expression and comparing it to another. The sensitive operations that require an access-control check correspond to the array subscripting operation.

To summarize, the main features of the language are:

1. The language is built around the notion of execution *on behalf of* principals. This allows a direct reference to principals in the language itself.
2. We carefully separate the programming language from the access-control logic. That is to say, the programming language does not force the use of a particular logic. The logical machinery is reflected in the language using an embedding. We hope this separation allows for modular development of the language and the logic.
3. Proof-checking and proof generation are essentially runtime activities. The results of dynamic proof checks is reflected into the types in the language. This allows us to *statically* reason about their generation and use to access resources. The separation between proof generation and their use allows for hoisting the dynamic checks away from the resource accesses.

Before presenting the details of our language, we motivate the idea of manifest security by presenting an example logic for access-control in Section 2. This section also motivates the issues regarding the embedding of logical constructs in the programming language. Section 3 presents the syntax and gives the type system for the language. We motivate our choice of typing judgments by presenting a judgmental formulation of the language in Section 4. Section 5 presents the runtime semantics. This is followed by a discussion of the access-control logic and its embedding in the language. Section 7 illustrates theorems regarding type-safety and access-control safety.

2 An example logic

Let us try formulating an example access-control policy that mimics simple ACL-based file protection in Unix. We want to be able to express a notion of ownership of resources by principals. Further, we want that a principal should be able to permit access to other principals for the resources it owns. As we shall see in this section, the manner in which this logic is formulated depends on the interface between the language and the logic.

2.1 Interfacing the policy logic with the language

The language interacts closely with the policy logic. This is because the terms in the language contain proofs of accessibility. Type-checking, therefore, involves proof checking and the rules for verifying proofs come from the policy logic. This ultimately leads to the language knowing about all the judgmental forms used by the policy logic. This leads to the question: how many types of judgments do we need in the logic? On one extreme, we might just have the truth judgment in the logic and represent all other judgmental concepts as propositions. On the other extreme, we may have a rich judgmental level in the logic. In the former case, the logic may be developed independently of the language. Type-checking would involve checking equality between propositions. The biggest drawback to such an approach is that since we do not allow any judgment other than the truth judgment, we can hope to define only very simple logics. For instance, consider formulating a logic with ownership and affirmation. We may start off with having a proposition $\mathtt{owns}(\mathbf{w}, R)$ for ownership, a proposition $\langle \mathbf{w} \rangle P$ for affirmation, and propositions $\mathtt{mayrd}(\mathbf{w}, R)$, $\mathtt{maywt}(\mathbf{w}, R)$ expressing that the principal $\mathbf{w}$ is allowed to read/write to a resource. We might also be able to have an axiom: $\forall w.\forall R.\mathtt{owns}(w, R) \supset \mathtt{mayrd}(\mathbf{w}, R)\ \mathtt{true}$. The trouble comes when we try to have axioms for affirmation. We do not know of a way to define the affirmation proposition without affirmation judgment. We might try to have the following rules instead:

$$\frac{\Gamma \vdash P\ \mathtt{true}}{\Gamma \vdash \langle \mathbf{w} \rangle P\ \mathtt{true}} \qquad \frac{\Gamma \vdash \langle \mathbf{w} \rangle P\ \mathtt{true} \qquad \Gamma, P\ \mathtt{true} \vdash \langle \mathbf{w} \rangle Q\ \mathtt{true}}{\Gamma \vdash \langle \mathbf{w} \rangle Q\ \mathtt{true}}$$

The above rules however can not function as introduction and elimination rules for the affirmation proposition as the purported elimination rule is not locally complete. In general, it might not be possible to design the required logic using only propositions and truth judgment.

The other design choice, i.e. having a rich judgmental level in the logic, looks more promising. The only downside is that the language gets tied closely to the logic since it needs to know about all the judgmental forms used in the logic.

For our example logic, we begin by having a judgment form $\mathbf{w}\ \texttt{owns}\ I$ that expresses ownership of the resource indexed I. In an implementation, evidence of this judgment would be credentials about ownership provided *a priori* in the implementation. The following rules define our example logic, where P ranges over the propositions in the logic (we use the subscript $\mathcal{L}$ to distinguish proof terms of the logic):

$$\frac{}{\Gamma, x : P \vdash x : P}(\text{hyp}) \qquad \frac{\Gamma \vdash p_1 : P_1 \qquad \Gamma \vdash p_2 : P_2}{\Gamma \vdash \langle p_1, p_2 \rangle_{\mathcal{L}} : P_1 \wedge P_2}(\wedge\text{I}) \qquad \frac{\Gamma \vdash p : P_1 \wedge P_2}{\Gamma \vdash \texttt{fst}_{\mathcal{L}} p : P_1}(\wedge\text{E}_1) \qquad \frac{\Gamma \vdash p : P_1 \wedge P_2}{\Gamma \vdash \texttt{snd}_{\mathcal{L}} p : P_2}(\wedge\text{E}_2)$$

$$\frac{\Gamma, x : P_1 \vdash p : P_2}{\Gamma \vdash \lambda_{\mathcal{L}} x.p : P_1 \supset P_2}(\supset \text{I}) \qquad \frac{\Gamma \vdash p : P_1 \supset P_2 \qquad \Gamma \vdash q : P_1}{\Gamma \vdash \texttt{app}_{\mathcal{L}} p\ q : P_2}(\supset \text{E}) \qquad \frac{\Gamma \vdash p : w\ \texttt{owns}\ R}{\Gamma \vdash \texttt{ownerrd}(p) : \texttt{mayrd}(\mathbf{w}, R)}$$

$$\frac{\Gamma \vdash p : w\ \texttt{owns}\ R}{\Gamma \vdash \texttt{ownerwt}(p) : \texttt{maywt}(\mathbf{w}, R)} \qquad \frac{\Gamma \vdash p : \langle \mathbf{w} \rangle \texttt{mayrd}(\mathbf{w}', R) \qquad \Gamma \vdash o : w\ \texttt{owns}\ R}{\Gamma \vdash \texttt{owner\_permit\_rd}(p, o) : \texttt{mayrd}(\mathbf{w}', R)}$$

$$\frac{\Gamma \vdash p : \langle \mathbf{w} \rangle \texttt{maywt}(\mathbf{w}', R) \qquad \Gamma \vdash o : w\ \texttt{owns}\ R}{\Gamma \vdash \texttt{owner\_permit\_wt}(p, o) : \texttt{maywt}(\mathbf{w}', R)}$$

We can now enrich the logic with affirmations as presented by Garg et al. [GBB+06]. Affirmations are used to express intent. We use the judgment form $\mathbf{w}\ \texttt{says}\ P$ to express that the principal thinks that the proposition P is true. It is important to note that P may not be considered to be true simply because a principal believes it to be. The above judgment only expresses the point of view of the principal $\mathbf{w}$. Further, we assume that all principals are rational and thereby affirm any true proposition. We thus have the judgment form $\mathbf{w}\ \texttt{says}\ P$ where P ranges over the set of propositions in the logic and $\mathbf{w}$ ranges over principals. We characterize the judgment using hypothetical judgments – the context Γ is a set of assumptions of the form $P\ \texttt{true}$.

$$\frac{\Gamma \vdash P\ \texttt{true}}{\Gamma \vdash w\ \texttt{says}\ P}$$

We internalize this judgment as a modality:

$$\frac{\Gamma \vdash w\ \texttt{says}\ P}{\Gamma \vdash \langle \mathbf{w} \rangle P\ \texttt{true}}$$

Next, we see how to use the affirmation judgment. Since an affirmation only expresses the point of view of a principal, it can be used to derive conclusions that are affirmations by the same principal.

$$\frac{\Gamma \vdash p : \langle \mathbf{w} \rangle P\ \texttt{true} \qquad \Gamma, x : P\ \texttt{true} \vdash q : \langle \mathbf{w} \rangle Q\ \texttt{true}}{\Gamma \vdash \texttt{let}\ x = p\ \texttt{in}\ q : \langle \mathbf{w} \rangle Q\ \texttt{true}}$$

This finishes our example logic. The proof terms of the logic appear as static constructors in the language and the propositions appear as classifiers of proof terms. Consider formulating a typical policy statement found in access-control lists. Suppose Alice is the owner of a file foo and wants to give read permission for the file to Bob. The ownership relation would be made manifest by a primitive certificate $\mathcal{C}_1$ of the type $\texttt{owns}(\text{Alice}, \text{foo})$. Alice could then issue a certificate $\mathcal{C}_2$ of the type $\langle\text{Alice}\rangle\texttt{mayrd}(\text{Bob}, \text{foo},)$. The proof that Bob can read the file foo would have to be assembled as owner_permit_rd$(\mathcal{C}_1, \mathcal{C}_2)$.

Kinds	K	$::=$	$\mathtt{TYPE} \mid \mathtt{RES} \mid \mathbf{w}_1 \leq \mathbf{w}_2$	
(Embedded propositions)				
		$\mid$	$\mathtt{mayrd}(\mathbf{w}, I) \mid \mathtt{maywt}(\mathbf{w}, I) \mid K_1 \to K_2$	
		$\mid$	$K_1 \times K_2 \mid \top$	
Constructors	A, P, I, w	$::=$	$\mathbf{c}$	Type constants
		$\mid$	$\mathbf{w}$	world constants
		$\mid$	$\mathtt{string}$	A type constant
		$\mid$	$A_1 \to A_2$	Function types
		$\mid$	$\mathtt{AC}[\mathbf{w}]A$	Monadic type constructor
		$\mid$	$\mathtt{Res}[I]$	Resource type
		$\mid$	$\mathbf{1}$	Unit
		$\mid$	α	Type variable
(Embedded proofs)				
		$\mid$	$\star$	
		$\mid$	$\langle P_1, P_2 \rangle$	Pair
		$\mid$	$\mathtt{fst}\ P$	Projection
		$\mid$	$\mathtt{snd}\ P$	
		$\mid$	$\lambda\alpha :: K.A$	Abstraction
		$\mid$	$A_1 A_2$	Application
Pure terms	M	$::=$	$x \mid \lambda x{:}A.M \mid M_1\ M_2 \mid \mathtt{ac}[\mathbf{w}]C \mid \langle\rangle$	
		$\mid$	$\iota[I]$	Constant ref-cells
Instructions	Ins	$::=$	$\mathtt{sudo}[\mathbf{w}][P](C)$	Movement to an accessible world
		$\mid$	$\mathtt{read}\ [I][P](M)$	Reading a ref cell
		$\mid$	$\mathtt{write}\ [I][P](M_1)(M_2)$	writing to a ref cell
Computations	C	$::=$	$\mathtt{return}\ M$	Monadic unit
		$\mid$	$\mathtt{letac}\ x = M\ \mathtt{in}\ C$	Monadic bind
		$\mid$	$\mathtt{su}[\mathbf{w}](M)\{\alpha.C_1 \mid C_2\}$	Gatekeeper for world-accessibility proofs
		$\mid$	$\mathtt{proverd}[I][\mathbf{w}]\{\alpha.C_1 \mid C_2\}$	Gatekeeper
		$\mid$	$\mathtt{provewt}[I][\mathbf{w}]\{\alpha.C_1 \mid C_2\}$	Gatekeeper
		$\mid$	Ins; $x.C$	Instruction
Values	v	$::=$	$\lambda x{:}A.M \mid \mathtt{ac}[\mathbf{w}]C \mid \iota[I] \mid \langle\rangle$	
Static context	Δ	$::=$	$\alpha :: K$	
Dynamic context	Γ	$::=$	$x{:}A$	
Signature	Σ	$::=$	$\cdot \mid \Sigma, c{::}K$	

Figure 1: Syntax

3 Syntax

The language distinguishes between effectful terms, called computations (C), and pure terms (M) [1]. We use a single computational monad [Mog89] to characterize access-control effects. Since effectful computations are always executed on behalf of a principal, the monad is indexed with the principal executing the computation. In this language, we consider only a fixed set of resources. Since the resources are fixed, they have been statically indexed. The indices are introduced at the level of types and the family of such indices is called RES. Resources are simply modeled as injections $\iota[I]$ of types I of the kind RES into the level of terms. The resource indices are introduced as constants using the signature Σ. The language can thus be thought of as being parametrized by the set of fixed resources.

The central primitives for accessing a resource are $\mathtt{read}\ [I][P](M)$ and $\mathtt{write}\ [I][P](M_1)(M_2)$. These primitives are parametrized by the resource index I being accessed. The read primitive reads the resource M (which should be of type $\mathtt{Res}[I]$). The write primitive modifies the value of the resource M_1 to M_2. Each of these primitives requires a proof P that permits the principal executing the primitive to access the resource.

[1] Computations are further divided into instructions (Ins) and plain computations (C). Instructions are the most primitive computations that cannot be further divided. More on why this division is required appears in Section 5.1.

The instruction $\texttt{sudo}[\mathbf{w}][P](C)$ allows a principal to switch to another principal. That is to say, the computation C is executed on behalf of the principal $\mathbf{w}$. After C is finished, computation of following code proceeds on behalf of the former principal. This operation requires a proof of movement P between the two principals which is generated using the computation $\texttt{su}[\mathbf{w}](M)\{\alpha.C_1 \mid C_2\}$. The instruction su is used to abstract away the details of runtime generation of proofs of movement. In an implementation, su might correspond to prompting the user for a password, and generating a proof if the password is correct. The computation C_1 is typed in a hypothetical setting assuming existence of a proof of accessibility. The computation C_2, on the other hand does not need such a proof. Thus the purpose of $\texttt{su}[\mathbf{w}](M)\{\alpha.C_1 \mid C_2\}$ is two-fold: first, it generates a proof of movement between worlds; second, it discharges the assumption of movement by substituting the proof for a free variable in C_1. In case no such proof exists, the second branch C_2 is executed.

Similar to the su are the commands $\texttt{proverd}[\mathbf{w}][I]\{\alpha.C_1 \mid C_2\}$ and $\texttt{provewt}[\mathbf{w}][I]\{\alpha.C_1 \mid C_2\}$ which interface the language with the access-control database. They are used to search for primitive proofs of accessibility. The computation $\texttt{proverd}[\mathbf{w}][I]\{\alpha.C_1 \mid C_2\}$ searches for a primitive proof of the kind $\texttt{mayrd}(\mathbf{w}, I)$. If such a proof is found in the access-control database, the assumption about existence of this proof in C_1 is discharged by substituting the actual proof for the free variable α representing the hypothesis. If no such proof is found, execution continues with C_2.

As the syntax suggests, the generation of proofs and the use of proofs have been separately dealt with. This allows for hoisting runtime checks for proofs away from the place where they are used thereby permitting reuse of proofs. The task of the type system is to ensure that the right kind of proofs are used in access-control sensitive operations.

Constructors in the language can be categorized into three groups, as is evident at the level of kinds: (i) those that classify terms in the language, which are classified by the kind TYPE; (ii) those that represent resource indices, classified by the kind RES; and those corresponding to proof terms in the access-control logic; classified by kinds that correspond to propositions in the logic. We refer to the latter sort of constructors as embedded proofs, and their kinds as embedded propositions, since they are defined by an embedding from proofs and propositions in the access-control logic.

3.1 Static typing

The typing judgments for computations are written as $\Delta; \Gamma \vdash_\Sigma C \,@\, \mathbf{w} \sim A$ meaning that the computation C is well-typed and can be executed on behalf of the principal $\mathbf{w}$. Pure terms are typed using the judgment $\Delta; \Gamma \vdash_\Sigma M : A$. The judgment $\Delta \vdash_\Sigma A :: K$ classifies constructors using kinds. Well-formed kinds are given by the judgment $\Delta \vdash_\Sigma K\ \texttt{kind}$. All the typing judgments are parametrized by a signature Σ which is used to introduce constant types. Note that no knowledge of the access-control database π is required for typechecking.

We begin by defining well-formed signatures. The primary purpose of a signature is to introduce constant resource indices. The type definitions in the signature form an ordered sequence, where latter definitions may depend on constants defined in former ones.

$\boxed{\Sigma\ \texttt{sig}}$

$$\frac{}{\cdot\ \texttt{sig}} \qquad \frac{\cdot \vdash_\Sigma K\ \texttt{kind} \qquad \Sigma\ \texttt{sig}}{\Sigma, c{::}K\ \texttt{sig}} \qquad \frac{\Sigma\ \texttt{sig} \qquad \cdot \vdash_\Sigma I{::}\texttt{RES}}{\Sigma, l{:}\texttt{Res}[I]\ \texttt{sig}}$$

$\boxed{\Delta \vdash_\Sigma K\ \texttt{kind}}$

$$\frac{}{\Delta \vdash_\Sigma \texttt{TYPE}\ \texttt{kind}} \qquad \frac{}{\Delta \vdash_\Sigma \texttt{RES}\ \texttt{kind}} \qquad \frac{}{\Delta \vdash_\Sigma \mathbf{w_1} \le \mathbf{w_2}\ \texttt{kind}} \qquad \frac{\Delta \vdash_\Sigma I :: \texttt{RES}}{\Delta \vdash_\Sigma \texttt{mayrd}(\mathbf{w}, I)\ \texttt{kind}}$$

$$\frac{\Delta \vdash_\Sigma I :: \texttt{RES}}{\Delta \vdash_\Sigma \texttt{maywt}(\mathbf{w}, I)\ \texttt{kind}} \qquad \frac{\Delta \vdash_\Sigma K_1\ \texttt{kind} \qquad \Delta \vdash_\Sigma K_2\ \texttt{kind}}{\Delta \vdash_\Sigma K_1 \to K_2\ \texttt{kind}} \qquad \frac{\Delta \vdash_\Sigma K_1\ \texttt{kind} \qquad \Delta \vdash_\Sigma K_2\ \texttt{kind}}{\Delta \vdash_\Sigma K_1 \times K_2\ \texttt{kind}}$$

$$\Delta \vdash_\Sigma \top\ \texttt{kind}$$

$\boxed{\Delta \vdash_\Sigma A :: K}$

$$\frac{\mathbf{c} :: K \in \Sigma}{\Delta \vdash_\Sigma \mathbf{c} :: K} \qquad \frac{\alpha :: K \in \Delta}{\Delta \vdash_\Sigma \alpha :: K} \qquad \frac{}{\Delta \vdash_\Sigma \mathtt{string}::\mathtt{TYPE}} \qquad \frac{}{\Delta \vdash_\Sigma \mathbf{1}::\mathtt{TYPE}}$$

$$\frac{\Delta \vdash_\Sigma A_1 :: \mathtt{TYPE} \qquad \Delta \vdash_\Sigma A_2 :: \mathtt{TYPE}}{\Delta \vdash_\Sigma A_1 \to A_2 :: \mathtt{TYPE}} \qquad \frac{\Delta \vdash_\Sigma A :: \mathtt{TYPE}}{\Delta \vdash_\Sigma \mathtt{AC}[\mathbf{w}]A :: \mathtt{TYPE}} \qquad \frac{\Delta \vdash_\Sigma I :: \mathtt{RES}}{\Delta \vdash_\Sigma \mathtt{Res}[I] :: \mathtt{TYPE}}$$

$$\frac{\Delta \vdash_\Sigma P_1 :: K_1 \qquad \Delta \vdash_\Sigma P_2 :: K_2}{\Delta \vdash_\Sigma \langle P_1, P_2 \rangle :: K_1 \times K_2} \qquad \frac{\Delta \vdash_\Sigma P :: K_1 \times K_2}{\Delta \vdash_\Sigma \mathtt{fst}\, P :: K_1} \qquad \frac{\Delta \vdash_\Sigma P :: K_1 \times K_2}{\Delta \vdash_\Sigma \mathtt{snd}\, P :: K_2}$$

$$\frac{\Delta, \alpha :: K_1 \vdash_\Sigma P :: K_2}{\Delta \vdash_\Sigma \lambda\alpha{::}K_1.P :: K_1 \to K_2} \qquad \frac{\Delta \vdash_\Sigma P_1 :: K_1 \to K_2 \qquad \Delta \vdash_\Sigma P_2 :: K_1}{\Delta \vdash_\Sigma\ P_1 P_2 :: K_2} \qquad \frac{}{\Delta \vdash_\Sigma \star :: \top}$$

$\boxed{\Delta; \Gamma \vdash_\Sigma \mathrm{Ins} \mathbin{@} \mathbf{w} \sim A}$

$$\frac{\Delta \vdash_\Sigma P :: \mathbf{w}' \leq \mathbf{w} \qquad \Delta; \Gamma \vdash_\Sigma C \mathbin{@} \mathbf{w}' \sim A}{\Delta; \Gamma \vdash_\Sigma \mathtt{sudo}[\mathbf{w}'][P](C) \mathbin{@} \mathbf{w} \sim A}$$

$$\frac{\Delta \vdash_\Sigma I :: \mathtt{RES} \qquad \Delta; \Gamma \vdash_\Sigma M : \mathtt{Res}[I] \qquad \Delta \vdash_\Sigma P :: \mathtt{mayrd}(\mathbf{w}, I)}{\Delta; \Gamma \vdash_\Sigma \mathtt{read}\,[I][P](M) \mathbin{@} \mathbf{w} \sim \mathtt{string}}$$

$$\frac{\Delta \vdash_\Sigma I :: \mathtt{RES} \qquad \Delta; \Gamma \vdash_\Sigma M_1 : \mathtt{Res}[I] \qquad \Delta; \Gamma \vdash_\Sigma M_2 : \mathtt{string} \qquad \Delta \vdash_\Sigma P :: \mathtt{maywt}(\mathbf{w}, I)}{\Delta; \Gamma \vdash_\Sigma \mathtt{write}\,[I][P](M_1)(M_2) \mathbin{@} \mathbf{w} \sim \mathbf{1}}$$

$\boxed{\Delta; \Gamma \vdash_\Sigma C \mathbin{@} \mathbf{w} \sim A}$

$$\frac{\Delta; \Gamma \vdash_\Sigma M : A}{\Delta; \Gamma \vdash_\Sigma \mathtt{return}\, M \mathbin{@} \mathbf{w} \sim A} \qquad \frac{\Delta; \Gamma \vdash_\Sigma M : \mathtt{AC}[\mathbf{w}]A_1 \qquad \Delta; \Gamma, x{:}A_1 \vdash_\Sigma C \mathbin{@} \mathbf{w} \sim A}{\Delta; \Gamma \vdash_\Sigma \mathtt{letac}\, x = M\, \mathtt{in}\, C \mathbin{@} \mathbf{w} \sim A}$$

$$\frac{\Delta; \Gamma \vdash_\Sigma M : \mathtt{string} \qquad \Delta, \alpha :: \mathbf{w}' \leq \mathbf{w}; \Gamma \vdash_\Sigma C_1 \mathbin{@} \mathbf{w} \sim A \qquad \Delta; \Gamma \vdash_\Sigma C_2 \mathbin{@} \mathbf{w} \sim A}{\Delta; \Gamma \vdash_\Sigma \mathtt{su}[\mathbf{w}'](M)\{\alpha.C_1 \mid C_2\} \mathbin{@} \mathbf{w} \sim A}$$

$$\frac{\Delta \vdash_\Sigma I :: \mathtt{RES} \qquad \Delta, \alpha :: \mathtt{mayrd}(\mathbf{w}', I); \Gamma \vdash_\Sigma C_1 \mathbin{@} \mathbf{w} \sim A \qquad \Delta; \Gamma \vdash_\Sigma C_2 \mathbin{@} \mathbf{w} \sim A}{\Delta; \Gamma \vdash_\Sigma \mathtt{proverd}[I][\mathbf{w}']\{\alpha.C_1 \mid C_2\} \mathbin{@} \mathbf{w} \sim A}$$

$$\frac{\Delta \vdash_\Sigma I :: \mathtt{RES} \qquad \Delta, \alpha :: \mathtt{maywt}(\mathbf{w}', I); \Gamma \vdash_\Sigma C_1 \mathbin{@} \mathbf{w} \sim A \qquad \Delta; \Gamma \vdash_\Sigma C_2 \mathbin{@} \mathbf{w} \sim A}{\Delta; \Gamma \vdash_\Sigma \mathtt{provewt}[I][\mathbf{w}']\{\alpha.C_1 \mid C_2\} \mathbin{@} \mathbf{w} \sim A}$$

$$\frac{\Delta; \Gamma \vdash_\Sigma \mathrm{Ins} \mathbin{@} \mathbf{w} \sim A' \qquad \Delta; \Gamma, x : A' \vdash_\Sigma C \mathbin{@} \mathbf{w} \sim A}{\Delta; \Gamma \vdash_\Sigma \mathrm{Ins}; x.C \mathbin{@} \mathbf{w} \sim A}$$

$\boxed{\Delta; \Gamma \vdash_\Sigma M : A}$

$$\frac{x{:}A \in \Gamma}{\Delta; \Gamma \vdash_\Sigma x{:}A} \qquad \frac{\Delta \vdash_\Sigma I :: \mathtt{RES}}{\Delta; \Gamma \vdash_\Sigma \iota[I] : \mathtt{Res}[I]} \qquad \frac{\Delta; \Gamma, x{:}A_1 \vdash_\Sigma M : A_2}{\Delta; \Gamma \vdash_\Sigma \lambda x{:}A.M : A_1 \to A_2}$$

$$\frac{\Delta; \Gamma \vdash_\Sigma M_1 : A_1 \to A_2 \qquad \Delta; \Gamma \vdash_\Sigma M_2 : A_1}{\Delta; \Gamma \vdash_\Sigma\ M_1 M_2 : A_2} \qquad \frac{\Delta; \Gamma \vdash_\Sigma C \mathbin{@} \mathbf{w} \sim A}{\Delta; \Gamma \vdash_\Sigma \mathtt{ac}[\mathbf{w}]C : \mathtt{AC}[\mathbf{w}]A} \qquad \frac{}{\Delta; \Gamma \vdash_\Sigma \langle\rangle : \mathbf{1}}$$

4 A judgmental formulation

4.1 The basic judgments

We begin with the question: what does access-control try to achieve? A first guess would be "access-control deals with controlling executability of terms by principals in accordance with some policy". But this definition seems far too general; for instance, why would one like to restrict a principal from evaluating the function application $\lambda x.x+1\;2$ (assuming that we are not concerned with information flow)? We would like any principal to be able to execute this term without having to produce a certificate permitting him to do so. Upon some thought, it is easy to see that access-control deals with controlling particular kinds of effects produced by principals, e.g. printing a file on a printer, reading the contents of a file, writing to a file etc. which are all effectful operations. Precisely what effects are being controlled depends on the particular setting. Following this observation, we divide the term level of our language into two syntactic categories: effectful computations and pure terms.

The two most basic judgments that give rise to terms and computations in the language are $A\;\texttt{true}$ and $A @ \mathbf{w}\;\texttt{comp}$ resp.. The first judgment is not surprising. The second is the computability judgment, similar to the computability judgment in [PH04], stating that A holds after the principal $\mathbf{w}$ produces some effect. The computability judgment has been formulated so as to include the mention of the principal on whose behalf computation would be executed. The idea is that a computation of the type $A @ \mathbf{w}\;\texttt{comp}$ should be executable on behalf of the principal $\mathbf{w}$.

Let us now look at proof term assignment for the above judgments. An evidence for the judgment $A\;\texttt{true}$ is a pure term (ranged over by M). The corresponding typing judgment is written as $M : A$. Since this typing judgment does not depend on any principal, it is necessary to restrict the proof terms of this judgment to pure terms that can be executed by all principals.

An evidence for the judgment $A @ \mathbf{w}\;\texttt{comp}$ is an effectful expression (C) which produces a value of type A in addition to producing an effect when run on behalf of the principal $\mathbf{w}$. We use the judgment $C @ \mathbf{w} \sim A$ to show the evidence of the judgment $A @ \mathbf{w}\;\texttt{comp}$. A may be considered to be the "type" of the computation C. Note that the type of a computation carries information about the security level of the expression viz. the principal that is allowed to run the computation.

We now characterize computability judgment using the lax modality as described in [PH04]. The extra principal parameter can be seen as indexing the monad. In order to define the computability judgment, we need to resort to hypothetical judgments. It turns out, for the purpose of this definition, that the hypotheses we need to assume are of the form $A\;\texttt{true}$. We do not need to assume anything of the form $A @ \mathbf{w}\;\texttt{comp}$. In what follows, let the context Γ denote hypotheses in the form of truth judgments.

Definition of computability judgment

1. If $\Gamma \vdash A\;\texttt{true}$, then $\Gamma \vdash A @ \mathbf{w}\;\texttt{comp}$
2. If $\Gamma \vdash A @ \mathbf{w}\;\texttt{comp}$ and $\Gamma, A\;\texttt{true} \vdash B @ \mathbf{w}\;\texttt{comp}$, then $\Gamma \vdash B @ \mathbf{w}\;\texttt{comp}$.

The first axiom says that if A is true, then it is also the case that A is true after a principal $\mathbf{w}$ executes something effectful. The proof of $A\;\texttt{true}$ is a pure term. This rule suggests that any principal must be able to execute a pure term as a computation, thereby producing the empty effect. Thus we obtain the computation form $\texttt{return}\,M$ which simply executes M and returns the resulting value. It is no surprise that the typing rule for this computation must be:

$$\frac{\Gamma \vdash M : A}{\Gamma \vdash \texttt{return}\,M @ \mathbf{w} \sim A}$$

To understand the second axiom, assume that A holds after $\mathbf{w}$ executes something effectful. Further, assume that in a hypothetical setting where A is true, $\mathbf{w}$ is able to execute something effectful to make B true. Then $\mathbf{w}$ can make B true in a world where A is not necessarily true in the first place. This axiom fits our interpretation of proof terms of computability judgments as effectful expressions. The trick is to string together the proof terms corresponding to A and B one after another. The resulting computation is executable by $\mathbf{w}$ (since its pieces are) and is effectful; the final effect being the two effects applied one after another. This leads to the composition expression Ins; $x.C$. We do not need the computation $C_1; x.C_2$ as it can be expressed inductively using the composition of instruction with a computation as the base case. This has been defined in section 5.1.

Finally, the computability judgment is internalized as the proposition $\mathtt{AC}[\mathbf{w}]A$:

$$\frac{\Delta; \Gamma \vdash C \mathbin{@} \mathbf{w} \sim A}{\Delta; \Gamma \vdash \mathtt{ac}[\mathbf{w}]C : \mathtt{AC}[\mathbf{w}]A}$$

This forms the introduction rule for the monadic type constructor. The proof term for the monadic type can be seen as a suspended computation. It is interesting to note that the proof term contains enough information to "open" up the suspended computation and start executing it. In particular, it specifies which principal is authorized to run it. The elimination rule for the monad makes sure that this restriction is obeyed. The elimination form is precisely the operation that opens up the computation and runs it. Since this operation is potentially effectful, the elimination form is a kind of computation. Note that a suspended computation of type $\mathtt{AC}[\mathbf{w}]A$ is opened only while running on behalf of the principal $\mathbf{w}$.

$$\frac{\Delta; \Gamma \vdash M : \mathtt{AC}[\mathbf{w}]A_1 \qquad \Delta; \Gamma, x : A_1 \vdash C \mathbin{@} \mathbf{w} \sim A_2}{\Delta; \Gamma \vdash \mathtt{letac}\ x = M\ \mathtt{in}\ C \mathbin{@} \mathbf{w} \sim A_2}$$

Note that the suspended computation is opened only on behalf of the principal that is authorized to open it. The elimination form can be viewed as a substitution of computations into computations.

It is interesting to see that there is no substitution principle for substituting computations inside terms. This is because there is no way to go from a term to a computational monad.

4.2 Movement between principals

The computability judgment in itself is not very interesting. Once computation starts on behalf of a principal, there is no way to get out of the indexed monad. In terms of execution, computation proceeds on behalf of a single principal and there is no way to switch between principals. We therefore add a new judgment which specifies *movement* between principals. A principal may be allowed to switch roles and effectively start executing on behalf of another principal. A direct application of such a facility would be in the implementation an `ssh`-like utility. A user Alice may `ssh` as the user Bob and start executing programs on behalf of Bob. Considering it from the point of computation, an `ssh` does a context-switch. The expression that was being run on behalf of Alice starts executing on behalf of Bob after the `ssh`. Another typical example is concerned with downgrading privileges to the minimum needed to do an operation. This is often referred to as the principle of least privilege. Consider, for instance, the `getty` process in Unix. This process is used to start a shell when a user logs in. The process runs on behalf of the principal `root`. However, just before spawning a shell for a user, it downgrades to the privileges of the user.

The movement judgment is written as $\mathbf{w} \leq \mathbf{w}'$. This says that the principal $\mathbf{w}'$ is allowed to switch to the principal $\mathbf{w}$ (in some sense, $\mathbf{w}'$ is *stronger* than $\mathbf{w}$ as it can execute all the expressions that $\mathbf{w}$ can). We use this intuition to define the judgment using the following rule:

If $\Gamma \vdash w \leq \mathbf{w}'$ and $\Gamma \vdash A \mathbin{@} \mathbf{w}\ \mathtt{comp}$, then $\Gamma \vdash A \mathbin{@} \mathbf{w}'\ \mathtt{comp}$

This rule gives us an elimination form for the judgment $\mathbf{w} \leq \mathbf{w}'$. We have not specified what stands as an evidence of the movement judgment. In our current language, the only way to establish an accessibility between principals ($\mathbf{w} \leq \mathbf{w}'$ may be seen as $\mathbf{w}$ being *accessible* from $\mathbf{w}'$) is by typing in the password for the principal $\mathbf{w}$ while executing on behalf of $\mathbf{w}'$. In an implementation, an evidence for this judgment would be generated by the runtime system whenever $\mathbf{w}'$ keys in $\mathbf{w}$'s password. The instruction $\mathtt{su}[\mathbf{w}](M)\{\alpha.C_1 \mid C_2\}$ is used to generate the above evidence by entering the password M. To keep things simple, we assume that M is a pure term of type `string` instead of an effectful computation reading off the password from some input stream. The `su` expression is one of the language interfaces between static type-checking and runtime generation of proofs. In case the password is correct, a proof certifying the accessibility to principal $\mathbf{w}$ is generated. Execution proceeds with C_1 after substituting the proof for α in C_1. In the other case, if the password turns out to be wrong, execution proceeds with C_2. C_2 has been typed in a context which does not assume that such a proof exists.

$$\frac{\Gamma \vdash M : \mathtt{string} \qquad \Gamma, \alpha : \mathbf{w}' \leq \mathbf{w} \vdash C_1 \mathbin{@} \mathbf{w} \sim A \qquad \Gamma \vdash C_2 \mathbin{@} \mathbf{w} \sim A}{\Gamma \vdash \mathtt{su}[\mathbf{w}'](M)\{\alpha.C_1 \mid C_2\} \mathbin{@} \mathbf{w} \sim A}$$

Note that C_1 has been typed in a hypothetical setting assuming that such a proof exists. Thus existence of such a proof is crucial to evaluate the expression C_1. C_1 may use this proof to establish further proofs of accessibility and to perform the actual context switch to a different principal. For instance, one may think of a scenario where the principals form a join-semilattice. Further suppose the following rule exists:

$$\frac{\Gamma \vdash p_1 : \mathbf{w}_1 \leq \mathbf{w}_0 \qquad \Gamma \vdash p_2 : \mathbf{w}_2 \leq \mathbf{w}_0}{\Gamma \vdash \mathtt{join}(p_1, p_2) : (\mathbf{w}_1 \sqcup \mathbf{w}_2) \leq \mathbf{w}_0}$$

In such a scenario, $\mathbf{w}_0$ would generate proofs of accessibility to $\mathbf{w}_1$ and $\mathbf{w}_2$ using M_1 and M_2 as passwords to $\mathbf{w}_1$ and $\mathbf{w}_2$ resp. and use those proofs to produce a proof of accessibility to $\mathbf{w}_1 \sqcup \mathbf{w}_2$:

$$\mathtt{su}[\mathbf{w}_1](M_1)\{\alpha.(\mathtt{su}[\mathbf{w}_2](M_2)\{\beta.\mathtt{sudo}[\mathbf{w}_1 \sqcup \mathbf{w}_2][\mathtt{join}(\alpha, \beta)](C_1) \mid \mathtt{fail}\}) \mid \mathtt{fail}\}$$

The movement from one principal to another is facilitated using the sudo instruction. The sudo instruction running on behalf of $\mathbf{w}_1$ requires a proof of the type $\mathbf{w}_2 \leq \mathbf{w}_1$. This proof establishes that the principal $\mathbf{w}_1$ can execute any computation that $\mathbf{w}_2$ is allowed to. Upon doing a sudo, the execution starts on behalf of the principal $\mathbf{w}_2$.

We specifically assume that the policy does not change over time. In such a case, a proof of movement to another world or a proof of being allowed to access a resource is persistent. The same proof may therefore be used again and again. Computations are typed in a hypothetical setting which assume certain proofs to be available. The primitive proofs are generated at runtime upon entry of a password/searching in the access-control list to see if a proof of access exists. More complex proofs may be constructed by using the proof constructors. The reference monitors for access operations explicitly require proofs of accessibility

5 Runtime Semantics

We present the runtime semantics of the language in terms of transition relation between states of an abstract machine. The states are organized as stacks of partially finished computation. In addition, the state consists of the reference-cell store ξ, and the store σ, of all the proofs of movement between principals that have been generated during the program execution. The state $\sigma; \xi; \cdot \rhd C \mathbin{@} \mathbf{w}$ with a closed computation C being evaluated on behalf of the principal $\mathbf{w}$ in an empty context forms the initial state of the machine. Note that computations are evaluated *on behalf of* a principal whereas the evaluation of terms is the same irrespective of the principal executing it. This is because the effects produced by computations may depend on the principal executing it.

While computations are said to be executed on behalf of some principal, it is important to note that the values are not situated. A computation can be termed as *situated* at a principal $\mathbf{w}$ if it can be run on behalf of $\mathbf{w}$. Values, on the other hand are not evaluated any further, and hence are not situated.

Before discussing the operational semantics, let us look at the additional syntax needed for describing the semantics. A continuation of the form $\kappa \parallel F$ represents a stack of partially finished executions. The frame F is a computation, a part of which is being evaluated. The part being evaluated may be a computation or a term (we shall view instructions as computations for the time being). The remaining computation in F is resumed after the part has been evaluated to a value. Depending on whether the part being evaluated is a computation or a term, the abstract machine state may be depicted as $\kappa \parallel F \rhd C \mathbin{@} \mathbf{w}$ or $\kappa \parallel F \rhd M$ resp. The $\rhd$ symbol is used to denote the focus of our attention while giving the rules of evaluation. In the first case, we are analyzing the computation C on behalf of $\mathbf{w}$, in order to evaluate it down to a value. Similarly, in the second case, a term is being analyzed. Another form of the abstract machine state comes into play when we have finished evaluating a computation/term to a value. It is then when the topmost frame is evaluated. The rules then specify how the recently calculated result of the subpart of F should be used in the evaluation of F. This state is depicted as $\kappa \parallel F \lhd v$, the symbol $\lhd$ denoting that our point of attention is now the topmost frame F where the value v would be used.

Continuation	κ	$::=$	$\cdot \mid \kappa \parallel F$
Continuation frame	F	$::=$	$\texttt{return}\ \bullet\ @\ \mathbf{w}$
		$\mid$	$\bullet; x.C\ @\ \mathbf{w}$
		$\mid$	$\texttt{letac}\ x = \bullet\ \texttt{in}\ C\ @\ \mathbf{w}$
		$\mid$	$\texttt{su}[\mathbf{w}](\bullet)\{\alpha.C_1 \mid C_2\}\ @\ \mathbf{w}$
		$\mid$	$\texttt{read}\ [I][P](\bullet)\ @\ \mathbf{w}$
		$\mid$	$\texttt{write}\ [I][P](\bullet)(M)\ @\ \mathbf{w}$
		$\mid$	$\texttt{write}\ [I][P](v)(\bullet)\ @\ \mathbf{w}$
		$\mid$	$(\bullet)\ M$
		$\mid$	$\lambda x{:}A.M\ (\bullet)$
Access control lists	π	$::=$	$\epsilon \mid \pi, \mathbf{c}{:}\texttt{mayrd}(\mathbf{w}, I)$
		$\mid$	$\pi, \mathbf{c}{:}\texttt{maywt}(\mathbf{w}, I)$
Ref-cell store	ξ	$::=$	$\cdot \mid \xi[\iota[I] \mapsto v]$
Su-permissions	σ	$::=$	$\cdot \mid \sigma, \alpha{::}w \leq \mathbf{w}'$
Abstract machine states	Abs	$::=$	$\sigma; \xi; \kappa \rhd C\ @\ \mathbf{w}$
		$\mid$	$\sigma; \xi; \kappa \rhd \text{Ins}\ @\ \mathbf{w}$
		$\mid$	$\sigma; \xi; \kappa \rhd M$
		$\mid$	$\sigma; \xi; \kappa \lhd v$

Figure 2: Syntax for describing operational semantics

Execution of terms is influenced by the facts about access-control pertinent at the time of execution. These decisions are captured in the access-control database π which also forms the context for inference using the access-control theory (see Section 6). π is used to model an access-control list which simply lists the files that a principal is allowed to access. It contains primitive proofs of the type $\texttt{mayrd}(I, \mathbf{w})$ and $\texttt{maywt}(I, \mathbf{w})$. We assume that all the proofs in π have unique names. Since no new resources are added at runtime and the access-control theory does not change with program execution, π remains fixed. σ represents the store of all the proofs of movement between principals generated during program execution. We assume a countably infinite set of names α which is used to generate a unique proof. It is possible to do away with any name for a primitive proof because the actual proofs are irrelevant. Two proofs with different shapes would both qualify as a proof for accessibility as long as they have the correct type.

The transition relation for operational semantics is parameterized by the access control database π and the language signature Σ. The relation is defined in Figure 3.

We have chosen to specify a non-deterministic semantics. In an implementation, all the non-deterministic choices can be resolved. The choices occur in the transition rules for $\texttt{su}, \texttt{proverd}$ and $\texttt{provewt}$. In each of three cases different branches are taken depending on whether a required proof is available. For $\texttt{su}[\mathbf{w}'](M)\{\alpha.C_1 \mid C_2\}$, in an actual implementation, the choice can be made deterministically by always generating a proof of movement if M is the correct password of $\mathbf{w}'$. Similarly, nondeterminism can be resolved in case of $\texttt{proverd}[I][\mathbf{w}']\{\alpha.C_1 \mid C_2\}$ (and similarly for $\texttt{provewt}$) by always choosing the first branch if a proof of type $\texttt{mayrd}(I, \mathbf{w}')$ ($\texttt{maywt}(I, \mathbf{w}')$) exists in the ACL database π, and substituting that proof for α in C_1.

The rest of operational semantics is fairly standard except for the rules concerning the access-control monad. The instruction $\texttt{sudo}$ is used to change the principal on whose behalf the computation is being run. A computation $\sigma; \xi; \kappa \rhd \texttt{sudo}[\mathbf{w}'][P](C)\ @\ \mathbf{w}$ steps to executing C on behalf of the new principal $\mathbf{w}'$. Note that this excursion to another principal $\mathbf{w}'$ is temporary. When the computation C finishes, execution resumes on behalf of the principal $\mathbf{w}$ that was executing the computation before $\texttt{sudo}$.

Another interesting transition rule is that of the computation $\texttt{letac}\ x = M\ \texttt{in}\ C$. M is a suspended computation. The type system ensures that the principal currently executing the $\texttt{letac}$ command is allowed to open the suspended computation. First, M is evaluated. The canonical forms lemma for values (Lemma B.8) ensures that M evaluates to a value of the form $\texttt{ac}[\mathbf{w}]C'$. The next step is to evaluate C' followed by execution of C after plugging in the result of C' in it. This sequential composition of two computations is written as $\langle C'/x \rangle C$ and is defined inductively in Section 5.1. We shall refer to this substitution as a *leftist* substitution since it is defined by induction on the structure of

$$
\begin{array}{rcll}
\sigma;\xi;\kappa \rhd \texttt{return}\ M\ @\ \mathbf{w} & \mapsto_{\pi;\Sigma} & \sigma;\xi;\kappa \parallel \texttt{return}\ \bullet\ @\ \mathbf{w} \rhd M & \\
\sigma;\xi;\kappa \parallel \texttt{return}\ \bullet\ @\ \mathbf{w} \lhd v & \mapsto_{\pi;\Sigma} & \sigma;\xi;\kappa \lhd v & \\
\\
\sigma;\xi;\kappa \rhd \texttt{letac}\ x = M\ \texttt{in}\ C\ @\ \mathbf{w} & \mapsto_{\pi;\Sigma} & \sigma;\xi;\kappa \parallel \texttt{letac}\ x = \bullet\ \texttt{in}\ C\ @\ \mathbf{w} \rhd M & \\
\sigma;\xi;\kappa \parallel \texttt{letac}\ x = \bullet\ \texttt{in}\ C\ @\ \mathbf{w} \lhd \texttt{ac}[\mathbf{w}]C' & \mapsto_{\pi;\Sigma} & \sigma;\xi;\kappa \rhd \langle C'/x\rangle C\ @\ \mathbf{w} & \\
\\
\sigma;\xi;\kappa \rhd \texttt{su}[\mathbf{w}'](M)\{\alpha.C_1 \mid C_2\}\ @\ \mathbf{w} & \mapsto_{\pi;\Sigma} & \sigma;\xi;\kappa \parallel \texttt{su}[\mathbf{w}'](\bullet)\{\alpha.C_1 \mid C_2\}\ @\ \mathbf{w} \rhd M & \\
\sigma;\xi;\kappa \parallel \texttt{su}[\mathbf{w}'](\bullet)\{\alpha.C_1 \mid C_2\}\ @\ \mathbf{w} \lhd v & \mapsto_{\pi;\Sigma} & \sigma[\beta : \mathbf{w}' \leq \mathbf{w}];\xi;\kappa \rhd [\beta/\alpha]C_1\ @\ \mathbf{w} & (\beta \text{ fresh}) \\
\sigma;\xi;\kappa \parallel \texttt{su}[\mathbf{w}'](\bullet)\{\alpha.C_1 \mid C_2\}\ @\ \mathbf{w} \lhd v & \mapsto_{\pi;\Sigma} & \sigma;\xi;\kappa \rhd C_2\ @\ \mathbf{w} & \\
\\
\sigma;\xi;\kappa \rhd \texttt{proverd}[I][\mathbf{w}']\{\alpha.C_1 \mid C_2\}\ @\ \mathbf{w} & \mapsto_{\pi;\Sigma} & \sigma;\xi;\kappa \rhd [\ulcorner m \urcorner/\alpha]C_1\ @\ \mathbf{w} & \text{if } \pi \vdash_{\mathcal{L}} m : \texttt{mayrd}(\mathbf{w}', I) \\
\sigma;\xi;\kappa \rhd \texttt{proverd}[I][\mathbf{w}']\{\alpha.C_1 \mid C_2\}\ @\ \mathbf{w} & \mapsto_{\pi;\Sigma} & \sigma;\xi;\kappa \rhd C_2\ @\ \mathbf{w} & \\
\sigma;\xi;\kappa \rhd \texttt{provewt}[I][\mathbf{w}']\{\alpha.C_1 \mid C_2\}\ @\ \mathbf{w} & \mapsto_{\pi;\Sigma} & \sigma;\xi;\kappa \rhd [\ulcorner m \urcorner/\alpha]C_1\ @\ \mathbf{w} & \text{if } \pi \vdash_{\mathcal{L}} m : \texttt{maywt}(\mathbf{w}', I) \\
\sigma;\xi;\kappa \rhd \texttt{provewt}[I][\mathbf{w}']\{\alpha.C_1 \mid C_2\}\ @\ \mathbf{w} & \mapsto_{\pi;\Sigma} & \sigma;\xi;\kappa \rhd C_2\ @\ \mathbf{w} & \\
\\
\sigma;\xi;\kappa \rhd \text{Ins}; x.C\ @\ \mathbf{w} & \mapsto_{\pi;\Sigma} & \sigma;\xi;\kappa \parallel \bullet; x.C\ @\ \mathbf{w} \rhd \text{Ins}\ @\ \mathbf{w} & \\
\sigma;\xi;\kappa \parallel \bullet; x.C\ @\ \mathbf{w} \lhd v & \mapsto_{\pi;\Sigma} & \sigma;\xi;\kappa \rhd [v/x]C & \\
\\
\sigma;\xi;\kappa \rhd \texttt{sudo}[\mathbf{w}'][P](C)\ @\ \mathbf{w} & \mapsto_{\pi;\Sigma} & \sigma;\xi;\kappa \rhd C\ @\ \mathbf{w}' & \\
\\
\sigma;\xi;\kappa \rhd \texttt{read}\ [I][P](M)\ @\ \mathbf{w} & \mapsto_{\pi;\Sigma} & \sigma;\xi;\kappa \parallel \texttt{read}\ [I][P](\bullet)\ @\ \mathbf{w} \rhd M & \\
\xi[\iota[I] \mapsto v]\ ; \kappa \parallel \texttt{read}\ [I][P](\bullet)\ @\ \mathbf{w} \lhd \iota[I] & \mapsto_{\pi;\Sigma} & \sigma;\xi[l \mapsto v]\ ; \kappa \rhd v & \\
\sigma;\xi;\kappa \rhd \texttt{write}\ [I][P](M_1)(M_2)\ @\ \mathbf{w} & \mapsto_{\pi;\Sigma} & \sigma;\xi;\kappa \parallel \texttt{write}\ [I][P](\bullet)(M_2)\ @\ \mathbf{w} \rhd M_1 & \\
\sigma;\xi;\kappa \parallel \texttt{write}\ [I][P](\bullet)(M_2)\ @\ \mathbf{w} \lhd \iota[I] & \mapsto_{\pi;\Sigma} & \sigma;\xi;\kappa \parallel \texttt{write}\ [I][P](l)(\bullet) \rhd M_2 & \\
\sigma;\xi[\iota[I] \mapsto v]\ ; \kappa \parallel \texttt{write}\ [I][P](l)(\bullet) \lhd v' & \mapsto_{\pi;\Sigma} & \xi[\iota[I] \mapsto v']; \kappa \rhd \langle\rangle & \\
\\
\sigma;\xi;\kappa \rhd\ M_1 M_2 & \mapsto_{\pi;\Sigma} & \sigma;\xi;\kappa \parallel\ (\bullet)M_2 \rhd M_1 & \\
\sigma;\xi;\kappa \parallel\ (\bullet)M' \lhd \lambda x{:}A.M & \mapsto_{\pi;\Sigma} & \sigma;\xi;\kappa \parallel\ \lambda x{:}A.M(\bullet) \rhd M' & \\
\sigma;\xi;\kappa \parallel\ \lambda x{:}A.M(\bullet) \lhd v & \mapsto_{\pi;\Sigma} & \sigma;\xi;\kappa \rhd [v/x]M & \\
\sigma;\xi;\kappa \rhd v & \mapsto_{\pi;\Sigma} & \sigma;\xi;\kappa \lhd v &
\end{array}
$$

Figure 3: Operational semantics

the computation being substituted. What makes this substitution unique is the form being substituted. The computation C' is not substituted directly in C. It is first evaluated to a value which is then substituted for x in C.

5.1 Leftist substitution

The substitution $\langle C_1/x\rangle C_2$ executes the computations C_1 and C_2 sequentially, substituting the value obtained by evaluating C_1 into C_2 for x. Since C_1 is evaluated first, this is defined by induction on the structure of C_1 instead of C_2.

$$
\begin{array}{lcl}
\langle \texttt{return}\ M/x\rangle C & = & [M/x]C \\
\langle \texttt{su}[\mathbf{w}'](M)\{\alpha.C_1 \mid C_2\}/x\rangle C & = & \texttt{su}[\mathbf{w}'](M)\{\alpha.\langle C_1/x\rangle C \mid \langle C_2/x\rangle C\} \\
\langle \text{Ins}; x'.C'/x\rangle C & = & \text{Ins}; x'.\langle C'/x\rangle C \\
\langle \texttt{letac}\ y = M\ \texttt{in}\ C_1/x\rangle C_2 & = & \texttt{letac}\ y = M\ \texttt{in}\ \langle C_1/x\rangle C_2 \\
\langle \texttt{proverd}[\mathbf{w}'][I]\{\alpha.C_1 \mid C_2\}/x\rangle C & = & \texttt{proverd}[\mathbf{w}'][I]\{\alpha.\langle C_1/x\rangle C \mid \langle C_2/x\rangle C\} \\
\langle \texttt{provewt}[\mathbf{w}'][I]\{\alpha.C_1 \mid C_2\}/x\rangle C & = & \texttt{provewt}[\mathbf{w}'][I]\{\alpha.\langle C_1/x\rangle C \mid \langle C_2/x\rangle C\}
\end{array}
$$

Note that the base cases for induction over C_1 are exactly what we have termed as instructions. Since instructions can be no further analyzed, sequencing one with a computation simply executes the instruction first. This is given by the computation form $\text{Ins}; x.C$. The cases of proof-generating computations are all similar. Let us consider $\langle \texttt{su}[\mathbf{w}'](M)\{\alpha.C_1 \mid C_2\}/x\rangle C$ first. The computation branches off to either C_1 or to C_2 before executing C. C can therefore be pushed inside both the branches after alpha-varying the binding variable α of the first branch to ensure that it does not falsely capture a free variable in C.

Next, consider the composition $\langle \texttt{letac}\, y = M \texttt{ in } C_1/x\rangle C_2$. This rule characterizes associativity of sequential composition. The first computation can be viewed as sequentially composing three computations: the suspension that M evaluates to, the computation C_1 and the computation C_2. These can be written as M composed with the sequentially composed computation $\langle C_1/x\rangle C_2$. Again, care must be taken to α-vary y to avoid capture in C_2.

6 The access-control logic

We use a very simple notion of access-control based on access-control lists (ACLs). We present a constructive logic extended with propositions that reflect access permissions.

Proposition	$p ::=$	$\texttt{mayrd}(\mathbf{w}, I) \mid \texttt{maywt}(\mathbf{w}, I) \mid p_1 \wedge p_2 \mid p_1 \supset p_2 \mid \top$
Proof terms	$m ::=$	$x \mid c \mid \langle m_1, m_2\rangle_{\mathcal{L}} \mid \texttt{fst}_{\mathcal{L}} m \mid \texttt{snd}_{\mathcal{L}} m \mid \lambda_{\mathcal{L}} x{:}p.m \mid \texttt{app}_{\mathcal{L}} m_1\, m_2 \mid \langle\rangle_{\mathcal{L}}$
Hypothesis	$\pi ::=$	$x{:}p, \pi \mid c{:}p, \pi$

where c represent constants.

The rules of deduction in the logic are given in Figure 4 using hypothetical judgments.

$$\frac{}{\pi, u{:}p \vdash_{\mathcal{L}} u{:}p}(\text{hyp}) \qquad \frac{\pi \vdash_{\mathcal{L}} m_1 : p_1 \qquad \pi \vdash_{\mathcal{L}} m_2 : p_2}{\pi \vdash_{\mathcal{L}} \langle m_1, m_2\rangle_{\mathcal{L}} : p_1 \wedge p_2}(\wedge\text{I}) \qquad \frac{\pi \vdash_{\mathcal{L}} m : p_1 \wedge p_2}{\pi \vdash_{\mathcal{L}} \texttt{fst}_{\mathcal{L}} m : p_1}(\wedge\text{E1})$$

$$\frac{\pi \vdash_{\mathcal{L}} m : p_1 \wedge p_2}{\pi \vdash_{\mathcal{L}} \texttt{snd}_{\mathcal{L}} m : p_2}(\wedge\text{E2}) \qquad \frac{\pi, u : p_1 \vdash_{\mathcal{L}} m : p_2}{\pi \vdash_{\mathcal{L}} \lambda_{\mathcal{L}} u{:}p.m : p_1 \supset p_2}(\supset \text{I}) \qquad \frac{\pi \vdash_{\mathcal{L}} m_1 : p_1 \supset p_2 \qquad \pi \vdash_{\mathcal{L}} m_2 : p_1}{\pi \vdash_{\mathcal{L}} \texttt{app}_{\mathcal{L}} m_1\, m_2 : p_2}(\supset \text{E})$$

$$\pi \vdash_{\mathcal{L}} \langle\rangle_{\mathcal{L}} : \top$$

Figure 4: The access-control logic

Note that we do not have introduction and elimination forms of the propositions $\texttt{mayrd}(\mathbf{w}, I)$ and $\texttt{maywt}(\mathbf{w}, I)$. An introduction form is missing because the only way to come up with a proof of the proposition, say, $\texttt{mayrd}(\mathbf{w}, I)$, in the current logic based on ACLs is to look up the access-control list π for such a proof. Since no information is put in to construct a proof of the proposition $\texttt{mayrd}(\mathbf{w}, I)$, there is no corresponding elimination construct for it.

6.1 Embedding the access-control logic into the language

We embed the propositions of the logic as kinds in the programming language. Proofs of propositions are embedded as constructors, with the kinding relation in the language reflecting the typing relation in the logic. This embedding is described in Figure 5 by induction on the structure of propositions and proofs thereof:

It is easy to check that this embedding preserves typing. This is proved by the following theorem:

Theorem 6.1. (Soundness of embedding)

1. *If p is a proposition in the logic, $\cdot \vdash_{\Sigma} \ulcorner p \urcorner\ \texttt{kind}$.*
2. *If $\pi \vdash_{\mathcal{L}} m : p$*
   then $\mathcal{V}ar(\ulcorner\pi\urcorner) \vdash_{\Sigma, \mathcal{C}on(\ulcorner\pi\urcorner)} \ulcorner m \urcorner :: \ulcorner p \urcorner$,
   where $\mathcal{V}ar(\cdot)$ and $\mathcal{C}on(\cdot)$ represent the variable and constant parts of the context.

Proof. 1. By induction on the structure of p.

2. By induction on the derivation of $\pi \vdash_{\mathcal{L}} m : p$.

□

$$
\begin{array}{lcl}
\ulcorner \mathtt{mayrd}(\mathbf{w}, I) \urcorner & = & \mathtt{mayrd}(\mathbf{w}, I) \\
\ulcorner \mathtt{maywt}(\mathbf{w}, I) \urcorner & = & \mathtt{maywt}(\mathbf{w}, I) \\
\ulcorner p_1 \wedge p_2 \urcorner & = & \ulcorner p_1 \urcorner \times \ulcorner p_2 \urcorner \\
\ulcorner p_1 \supset p_2 \urcorner & = & \ulcorner p_1 \urcorner \to \ulcorner p_2 \urcorner \\
\ulcorner \top \urcorner & = & \top \\
 & & \\
\ulcorner u \urcorner & = & u \\
\ulcorner c \urcorner & = & c \\
\ulcorner \langle m_1, m_2 \rangle_{\mathcal{L}} \urcorner & = & \langle m_1, m_2 \rangle \\
\ulcorner \mathtt{fst}_{\mathcal{L}} m \urcorner & = & \mathtt{fst}\ \ulcorner m \urcorner \\
\ulcorner \mathtt{snd}_{\mathcal{L}} m \urcorner & = & \mathtt{snd}\ \ulcorner m \urcorner \\
\ulcorner \lambda_{\mathcal{L}} u : p.m \urcorner & = & \lambda u {::} \ulcorner p \urcorner . \ulcorner m \urcorner \\
\ulcorner \mathtt{app}_{\mathcal{L}} m_1\ m_2 \urcorner & = & \ulcorner m_1 \urcorner \ulcorner m_2 \urcorner \\
\ulcorner \langle \rangle_{\mathcal{L}} \urcorner & = & \star
\end{array}
$$

Figure 5: Embedding access-control logic into language

6.2 An access-control database

Programs in the language execute under a static set of "facts" about access-control. These facts represent the access-control decisions relevant in that particular execution, and are represented as a set of proof-proposition pairs (m, p), where each term m is a *constant term* and represents a proof of the corresponding proposition p. We refer to such proof terms as *primitive* proofs. Note that this is the way we introduce proofs of propositions like $\mathtt{mayrd}(\mathbf{w}, I)$ and $\mathtt{maywt}(\mathbf{w}, I)$ which do not have any introduction rules. In addition to these propositions, the access-control database may have primitive proofs of other propositions as well. This access-control database along with the rules of deduction of the logic form the *access-control theory*.

7 Type safety and access-control safety

The language is type-safe with respect to the operational semantics. A formal proof of type-safety appears in Appendix A.

Apart from type-safety, we require that the language is safe in terms of how it allows access to resources. Formalizing the access-control safety theorem has two parts to it: first, proving that whenever a principal accesses a resource, the access control theory permits that access, i.e. there is a proof derivable from the access-control theory π under which the computation was evaluated. This is shown by the following theorem (and a corresponding theorem for `write`):

Theorem 7.1. *If $\cdot \vdash_\Sigma C \mathbin{@} \mathbf{w} \sim A$ and $\cdot; \xi; \cdot \rhd C \mathbin{@} \mathbf{w} \mapsto^{\star}_{\pi;\Sigma} \sigma; \xi'; \kappa \rhd \mathtt{read}\ [I][P](M) \mathbin{@} \mathbf{w}'$, then access to resource indexed I is allowed to the principal $\mathbf{w}'$ according to the access-control theory, i.e. there exists a proof p such that $\pi \vdash_{\mathcal{L}} p : \mathtt{mayrd}(\mathbf{w}', I)$.*

Proof. Refer to appendix C. □

Secondly, since principals can switch identities, we need stronger guarantees with respect to what all a principal can execute. The main idea is that a principal executing a computation C is responsible for all further computations to which C evaluates to, no matter on whose behalf they execute. The following theorem proves that starting execution on behalf of a principal $\mathbf{w}$, a computation can only lead to execution on behalf of those principals $\mathbf{w}'$, for which $\mathbf{w}$ has a proof of movement, i.e. a proof of type $\mathbf{w}' \leq \mathbf{w}$ exists. Proving this theorem requires us to generalize the statement to include all principals that appear on the computation stack of an execution. We refer to such principals as being *active* in the abstract machine state.

The *active* principals in an abstract machine state Abs are defined by induction on Abs. The judgment $\mathbf{w} \uparrow \text{Abs}$ is used to denote that $\mathbf{w}$ is active in the state Abs. The auxiliary judgment form $\mathbf{w} \uparrow \kappa$ denotes that $\mathbf{w}$ is active in the continuation κ.

$$\frac{\mathbf{w} \uparrow \sigma; \xi; \kappa}{\mathbf{w} \uparrow \sigma; \xi; \kappa \triangleright C \mathbin{@} \mathbf{w}'} \qquad \frac{\mathbf{w} \uparrow \sigma; \xi; \kappa}{\mathbf{w} \uparrow \sigma; \xi; \kappa \triangleright M} \qquad \frac{\mathbf{w} \uparrow \sigma; \xi; \kappa}{\mathbf{w} \uparrow \sigma; \xi; \kappa \triangleright \text{Ins} \mathbin{@} \mathbf{w}'} \qquad \frac{\mathbf{w} \uparrow \sigma; \xi; \kappa}{\mathbf{w} \uparrow \sigma; \xi; \kappa \triangleleft v}$$

$$\mathbf{w} \uparrow \sigma; \xi; \kappa \triangleright C \mathbin{@} \mathbf{w} \qquad \mathbf{w} \uparrow \sigma; \xi; \kappa \triangleright \text{Ins} \mathbin{@} \mathbf{w}$$

$$\frac{\mathbf{w} \uparrow \kappa}{\mathbf{w} \uparrow \kappa \parallel F} \qquad \frac{}{\mathbf{w} \uparrow \kappa \parallel \texttt{return} \bullet \mathbin{@} \mathbf{w}} \qquad \frac{}{\mathbf{w} \uparrow \kappa \parallel \bullet; x.C \mathbin{@} \mathbf{w}} \qquad \frac{}{\mathbf{w} \uparrow \kappa \parallel \texttt{letac}\ x = \bullet\ \texttt{in}\ C \mathbin{@} \mathbf{w}}$$

$$\frac{}{\mathbf{w} \uparrow \kappa \parallel \texttt{su}[\mathbf{w}'](\bullet)\{\alpha.C_1 \mid C_2\} \mathbin{@} \mathbf{w}} \qquad \frac{}{\mathbf{w} \uparrow \kappa \parallel \texttt{read}\ [I][P](\bullet) \mathbin{@} \mathbf{w}} \qquad \frac{}{\mathbf{w} \uparrow \kappa \parallel \texttt{write}\ [I][P](\bullet)(M) \mathbin{@} \mathbf{w}}$$

$$\frac{}{\mathbf{w} \uparrow \kappa \parallel \texttt{write}\ [I][P](v)(\bullet) \mathbin{@} \mathbf{w}}$$

Theorem 7.2. (Access-control safety): *If* $\pi, \Delta; \cdot \vdash_\Sigma C \mathbin{@} \mathbf{w} \sim A$
and $\cdot; \xi; \cdot \triangleright C \mathbin{@} \mathbf{w} \mapsto^{\star}_{\pi;\Sigma}$ *Abs,*
then for each $\mathbf{w}_i$ *active in Abs, either* $\mathbf{w}_i = \mathbf{w}$*, or there exist sequences* $\langle \mathbf{w}_1, \ldots, \mathbf{w}_{i_n} \rangle, \langle P_1, \ldots, P_{i_n - 1} \rangle$*,*
such that $\mathbf{w}_1 = \mathbf{w}, \mathbf{w}_{i_n} = \mathbf{w}_i$ *and* $\sigma, \Delta \vdash_{\Sigma, \ulcorner\pi\urcorner} P_j :: \mathbf{w}_{j+1} \leq \mathbf{w}_j$ *for* $1 \leq j < i_n$*, where* σ *is the Su-permissions component of Abs.*

Proof. We proceed by induction on the number of steps of the operational semantics it takes to reach Abs from $\cdot; \xi; \cdot \triangleright C \mathbin{@} \mathbf{w}$. The full proof appears in Appendix C. □

8 An example

To illustrate the concepts in the language, let us consider programming an editor for writing reviews to papers submitted in a conference. A paper is assigned to one or more referees and a referee may read or write reviews only for a paper that has been assigned to him. In addition to referees, there is a program chair who can read/write reviews for any paper he wishes. Usually a program chair has additional responsibilities such as assigning papers to referees, but we do not consider them here.

The system consists of an access policy determining the referees assigned to each paper. For a referee $\mathtt{r}$ assigned to a paper $\mathtt{p}$, the policy database would have the following axioms: $\mathtt{mayrd}(\mathtt{r}, \mathtt{p_r})$ and $\mathtt{maywt}(\mathtt{r}, \mathtt{p_r})$, where $\mathtt{p_r}$ is the static index of the review file of the referee $\mathtt{r}$ for the paper $\mathtt{p}$. In addition to similar rules for each (referee, paper) pair, axioms of the form $\mathtt{mayrd}(\mathtt{pc}, \mathtt{p_r})$ and $\mathtt{mayrd}(\mathtt{pc}, \mathtt{p_r})$ for each pair of $\mathtt{p}$ and $\mathtt{r}$, are needed in order for the program chair ($\mathtt{pc}$) to access all the reviews. How these axioms are entered into the policy database and how this policy is manipulated by the program chair is out of scope of this paper. The access-control logic is the same as that described in this paper. Note that this kind of a policy prohibits a reviewer from accessing reviews written by other reviewers.

In order to read/write the review for a paper, a referee logs into the system and edits the review using a customized editor. The task of the editor is to provide an interface to securely edit reviews. Figure 6 describes the skeleton of the editor illustrating how it can be based on the primitives provided in our security language. The example shows the code of an editor compiled for a specific referee $\mathtt{r}$. Making the editor polymorphic for the users requires extending the security language for runtime principals and is a topic for future work.

The editor compiled for the referee $\mathtt{r}$ first prompts the user to enter the correct password in order to authenticate him as the referee $\mathtt{r}$. This is done using the `su` primitive. Thus the editor starts running on behalf of the `public` user but immediately switches to executing on behalf of a specific referee. Once authenticated, the user may enter the name of the review file to edit. This may involve the user typing the name of the file on the terminal or selecting it from a list of files presented to it by the editor. The details of exactly how this happens has been abstracted away in the computation $\mathtt{whichreview?}\{p : \mathtt{Res}[{}'p].C\}$. After interaction with the user, the name of the review file to be edited is bound to the variable $\mathtt{p}$ and the static index representing the file is bound to $'\mathtt{p}$ in the incipient computation C. At

```
su[r](passwd){'a.sudo[r]['a](
                    whichreview?{p : Res['p].
                          proverd[r]['p]{'b.
                               provewt[r]['p]{'c.
                                     read['p]['b](p);
                                     cnts.display(cnts);
                                     _. // The read-save loop
                                        // Loop until the user quits
                                     while(true){
                                        read_terminal;
                                        cnts.write['p]['c](p)(cnts)
                                     }
                                   | print "Read-only file"
                               }
                             | print "Could not open the file for reading"
                          }
                     }
                )
              | print "Login failed."
}
```

Figure 6: An editor for writing paper reviews securely

this stage, the editor checks if the referee r has the required permissions to read the file p. This is done by doing a proverd. If successful, this search yields a proof of the type $\mathtt{mayrd}(\mathtt{r}, \mathtt{p_r})$. This proof is then used by the editor to read the file from the review repository binding the contents to cnts and then display it to the user. At this stage the editor enters a read-save loop.

In the read-save loop, the editor cycles periodically between reading the input from the user and saving the current version back to the file in the repository. Reading the input from the user for a fixed amount of time has been abstracted in the computation read_terminal. Each call to write the file back to the repository requires the editor to use a proof of the type $\mathtt{maywt}(\mathtt{r}, \mathtt{p})$. This proof is generated only once per editing session and is reused during the read-save loop.

9 Future work

The current work provides a basic framework for building more complex languages for access-control. We would like to extend the work along the following dimensions:

Runtime principals Assuming a constant set of principals is too restrictive. In the real world, a number of different principals may interact with the system. The identity of all such principals that would interact with the system may not even be known statically. A simple example is that of a web browser. The web pages that the browser opens may be considered to be principals (e.g. the web page might be an applet trying to execute on the local host). In such a scenario, an applet may try to read a local file after presenting a certificate that the local user has delegated the authority to it. The browser should allow all such applets to function properly.

Runtime principals in a security setting similar to ours have been investigated by Tse et al. [TZ04]. They use polymorphism over principals to accommodate reasoning with principals that are unknown statically. In our setup, the typing judgments for computations depend on the identity of the principal executing the computation. Principals thus interact in a non-trivial manner with type checking. Here we would like to investigate the use of static indices to model runtime principals, much the same way as we handle resources in the current framework.

Runtime generation of resources The system would be pretty much unusable if there is no more than a fixed number of resources to work with. Here again, we would like to facilitate generation of new resources at runtime. In the current framework, resources are identified by their resource index. A resource of type $\mathtt{Res}[I]$ has index $I :: \mathtt{RES}$.

In order to extend the current framework to support dynamic resources, we need a way to create unique indices at runtime. Note that the actual index of a resource is of no runtime concern to the principal. Indices are static entities meant only to identify resources statically. This suggests that the actual index of the newly created resource should be abstracted away. One way to achieve this is to use existential types. Suppose we have a computation `new` that creates resources at runtime. For the time being, let us not worry about the access control policies associated with this resource. Since we wish to abstract away the index of the resource, we could have the following typing rule for `new`:

$$\frac{}{\Delta; \Gamma \vdash \texttt{new} @ \mathbf{w} \sim \exists i{::}\texttt{RES}.\texttt{Res}[i]}$$

`new` can then be thought of as packaging a resource along with the index that identifies it with the index forming the hidden part of the package. The elimination form for existential quantifier would open the package as:

$$\frac{\Delta; \Gamma \vdash M_1 : \exists \alpha{::}K.A_1 \qquad \Delta, \alpha :: K; \Gamma, x : A_1 \vdash M_2 : A_2 \qquad \Delta; \Gamma \vdash A_2 :: \texttt{TYPE}}{\Delta; \Gamma \vdash \texttt{unpack}\ M_1\ \texttt{as}\ \{\alpha, x\}\ \texttt{in}\ M_2 : A_2}(\exists\text{-E})$$

There is however a problem with this approach. This arises due to the *generativity* of existentially quantified packages. Each time an existential package is opened, we get new identities for the hidden (and therefore for the dependent) part of the package. This is precisely the opposite of what we want. A resource package generated using `new` should yield the same resource each time when opened. To this end, we could use translucent sums instead of existential packages.

Translucent sums are more flexible than existential packages. As the name might suggest, the difference between translucent sums and existential packages is that of the amount of information available about the hidden component in the scope it is being used. Existential packages are opaque in the sense that absolutely no information can be attached to the hidden part. This is the reason why opening the same package twice generates two different entities because there is no information whatsoever available about the hidden part which could be used to check an equivalence. Translucent sums, on the other hand, allow a selfification rule that enables one to note the identity of the opened instance and use it to type the package. We do not go into further details of translucent sums here. A detailed account can be found in [HL94].

Using translucent sums, it is easy to type `new` as:

$$\frac{}{\Delta; \Gamma \vdash \texttt{new} @ \mathbf{w} \sim \{\mathbf{I} \rhd i :: \texttt{RES}, \mathbf{R} \rhd r : \texttt{Res}[i]\}}$$

`new` would return a dependent labeled sum with the $\mathbf{I}$ component being the resource index and the $\mathbf{R}$ component being the actual resource. The following selfification rules, taken from [HL94], would allow propagation of the identity of opened package to the type of the package (V denotes a value):

$$\frac{\Delta; \Gamma \vdash V : \{\mathbf{L_1} \rhd \alpha{::}K, \ldots, \mathbf{L_n} \rhd x_n{:}A_n\}}{\Delta; \Gamma \vdash V : \{\mathbf{L_1} \rhd \alpha = V.\mathbf{L_1}{::}K, \ldots, \mathbf{L_n} \rhd x_n{:}A_n\}}(\text{Value-o})$$

$$\frac{\Delta \vdash V.\mathbf{L_1}{::}K' \qquad \Delta; \Gamma \vdash V : \{\mathbf{L_1} \rhd \alpha{::}K, \ldots, \mathbf{L_n} \rhd x_n{:}A_n\}}{\Delta; \Gamma \vdash V : \{\mathbf{L_1} \rhd \alpha{::}K', \ldots, \mathbf{L_n} \rhd x_n{:}A_n\}}(\text{Value-v})$$

Dynamic policy updates Almost any interesting real-world situation requires updating the access-control policy at runtime. For example, when new resources are generated, the access control theory must be augmented with new policies regarding the new resources. With the architecture for `new` as sketched above in place, adding new access control axioms to the theory can be done relatively easily. We shall briefly sketch this.

Another kind of updates to access control theory involve changing the theory in a manner that previously established proofs might no longer hold. This is the case with revocation. e.g. A proof that principal B speaks for principal A holds only until A has revoked the authority that it had granted to B. If we wish to provide revocation as a primitive computation form in the language, we need to tackle the following questions first:

1. **Modeling revocation** We view revocation of rights by a principal as a positive assertion of his intent. That is to say rather than regarding revocation as a process of subtracting a policy decision from the access control

theory, we would like to model it as a separate judgment which would be added to the theory. It is immediately clear that the present logic becomes unsuitable in such a scenario because the present logic is monotonic. In the new setting, addition of new axioms to the theory may invalidate proofs written in the previous theory. This suggests the use of non-monotonic logics to formulate revocation. In a naive implementation, the deduction of a judgment might depend on scanning the entire policy database. We do not yet know of a logic in which such a scenario can be elegantly expressed and proof search implemented efficiently.

2. **Status of proofs** Thus far, the proofs in the language had been static entities. This was largely due to the fact that proof-checking was required during type-checking. Revocation, which interacts with the proofs non-trivially is a runtime computation. This violates the principle of phase-distinction. Certain static entities (proofs in this case) start to depend on runtime terms. In order to maintain a phase-distinction, we need to come up with alternative formulations of the proofs that work with the revocation language while still preserving decidability of static proof-checking.

Information flow Access control in systems is intimately connected with flow of information. This connection between access control and information flow is not trivial. In the SLam calculus [HR98], access-control is ensured by controlling the set of direct readers of an object. This restricts legitimate eliminations to those in which the introduction form being eliminated is owned by a principal that *trusts* the eliminator principal. Information flow is achieved by specifying the set of readers for the result of the elimination. This approach views information flow control as a mechanism to ensure that no illegal access to information takes place.

An orthogonal view (as suggested by Frank Pfenning) is that access control deals with controlling who can read an item while information flow control is concerned with controlling writes. This view is suitable particularly in our setting of reference cells, where information flows from one principal to another when the data written to a ref-cell by a principal is read by another. If one considers this view of information flow control, the following question becomes pertinent: Can access control alone suffice to prevent illegal flows of information?

Another interesting body of work is related to quantifying the information leaked by a program using concepts from information theory. Malacaria et al. [Mal07, CHM05] present an information-theoretic account of leakage of information in a functional programming language. Such an approach is interesting because some leakage of information turns out to be essential for the functioning of many applications. For instance, checking if the password input by a user is corrected itself reveals some information (howsoever little) about the secret password.

The challenge before us is to identify a view of information flow conducive to our setting of effectful computations and come up with the non-interference theorem that we would like of our system to hold.

10 Related Work

There is a large body of research work related to access control. In this section, we shall only touch upon some of the work that has influenced our work or is related to it.

10.1 Logics for access control

A large body of work regarding access control is related to designing logics which provide constructs to efficiently reason about various notions found in access control. One of the first accounts of a study of access control from a logical perspective can be found in [ABLP93]. In this paper, Abadi et al. formulate notions of principals, assertions by principals and delegation in a logical manner and discuss a spectrum of design choices that can be made depending on the setting at hand. We believe that our language can be extended to include the calculus of principals as introduced by Abadi et al. in [ABLP93]. Some similarities can be already seen. For instance, the judgment $\mathbf{w}_1 \leq \mathbf{w}_2$ in our language can be viewed as a form of delegation: principal $\mathbf{w}_1$ delegating all his capabilities to $\mathbf{w}_2$. This view of delegation is an indirect one. One usually thinks of delegation as being initiated by the granter. In our scenario, it is the grantee which actively generates the delegation by typing a password. Currently, the syntax of principals in our language lacks a way to specify roles since the set of principals is fixed. One of the future tasks is to extend the calculus to include runtime generation of principals. Dynamic specification of new roles provides a motivating example to consider this line of future work.

10.1.1 The `says` modality

A number of other studies on using logic to formulate notions regarding access control have been undertaken. A brief survey can be found in [Aba03]. One of the key notions dealt with in such logics is the concept of assertion by a principal. This is of central importance because access control is about determining whether a principal should be trusted on what it says. Garg and Pfenning [GP06] provide a constructive logic which defines the affirmation modality. Their logic is particularly useful in our setting because of two reasons: first, being constructive, it is very well suited for a setting like ours of proof carrying authorization; and secondly, because it enjoys a useful non-interference-like proof-theoretic property.

10.1.2 Dependency due to access control

In [Aba06], Abadi provides another characterization of the `says` modality using the framework of Dependency Core Calculus [ABHR99] by interpreting the indexed dependency monad as a proposition expressing assertion by a principal. Not surprisingly, the calculus is mainly concerned with controlling the dependencies between computations at various levels of security. A computation at a high level can not depend on a value at a lower level. In other words, a computation at a particular security level l is able to influence only the computations at levels where l is trusted, i.e levels below l in the trust lattice. This basic idea of dependence can also be seen in our approach where computations run on behalf of a principal $\mathbf{w}_1$ can effectively be run (by doing a `sudo`) by another principal $\mathbf{w}_2$ as long as $\mathbf{w}_2$ is allowed to act as $\mathbf{w}_1$. In contrast however, instead of assuming a fixed lattice of principals, our language allows the relationship between principals to evolve dynamically as principals try to identify themselves as other principals by typing in their passwords.

10.2 Languages for authorization policy

Authorization policies can be specified in various ways each differing in expressive power and mechanisms for reasoning about the policy. Languages for specifying authorization policies range from the simple access-control lists, to the logically founded languages such as Binder. Examples of commercial security languages, usually designed for a specific scenarios abound, e.g. X.509, SDSI, SPKI [RB] and KeyNote [BFK98] etc. Binder [DeT02] is based on the Datalog fragment of Prolog. Authorization policies are expressed as logic programs. The language is polynomial-time decidable.

10.3 Type systems for access-control

10.3.1 Ownership types

A closely related type system is the one with ownership types proposed by Krishnaswami et al. [KA05]. Their system is primarily concerned with information hiding in a module system by preventing leakage of information through aliasing. They formulate the solution by restricting access to module functions from other modules. This notion can be related to the idea of restricting resource access to principals. Just as with ownership types, principals can be thought of as being owners of certain resources. Access to resources is permitted only if the accessor principal has permissions to access resources owned by another principal. While adequate in the setting of a module system, ownership types do not provide a flexible setting for writing programs dealing with access control. Their type system, adapted to access control, can be considered as an instance of our framework obtained by fixing the access control theory to consist of notions of ownership. Our framework will, however, be able to work with richer logics of access control since the only judgments regarding access control that the language depends on are those regarding accessing a resource by a principal. The kind of judgments that access control judgments are derived from depends completely on the access control logic which can be developed orthogonally.

10.3.2 SLam calculus

Heintze et al. present the SLam calculus [HR98] for access control. They consider a setting in which pieces of data are owned by principals and programs are executed on behalf of principals. Further, a partial order on principals is assumed. This ordering models the trust lattice of the principals. A principal executing a program receives all information about the result of the program when the execution finishes. The calculus determines if executing a program on behalf of a principal is safe, i.e. data owned by a high-security principal are not leaked to a low security principal upon execution of a program by the low-security principal.

Since the programs are represented by elimination forms, the SLam calculus requires that all the elimination forms be annotated with a principal. This represents executing the elimination form on behalf of the tagged principal. Introduction forms, on the other hand, carry information about which principal is allowed to *use* the value by eliminating the introduction. The type system ensures that the tags on the introduction form and the elimination construct match, or at least that the elimination tag has all the *read* permissions that the introduction tag enjoys.

10.3.3 Runtime stack inspection

Another widely studied approach to access control is the runtime stack inspection method [FG03] used by the Java Virtual Machine. The underlying idea is that the privileges available to a piece of code at runtime depend on the history of execution. A system function call that has been invoked by an untrusted caller would have less privileges as compared to the same function invoked by a trusted code. Skalka and Smith [SS00], and Pottier et al. [PSS05] have provided a type system to statically ensure that no violations of the security policy would happen during runtime call stack inspection. Higuchi et al. [HO03] extend the ideas to provide a type system for Java bytecode language. One of the weaknesses of their system is the loss of modularity. All classes that are subclasses of a particular class need to be known for type-checking the bytecode.

10.3.4 File-system secrecy

Chaudhuri et al. [CA06] study a type system for statically ensuring secrecy in a file system. They consider a setting in which filenames and/or their contents may be secret. The principals are modeled as processes in a pi calculus. One of the limitations of their system is the static nature of the trust characteristics of the principals. In particular, they assume that it is statically known as to which principals can be trusted. In contrast, our approach makes no such assumptions because trust is established explicitly by means of a proof.

11 Conclusions

This paper presents a prototype language for programming in environments where access to resources is governed by the explicit proofs. The main motivation for this language has been to capture results of runtime checks as types. The type system ensures that all the required checks would be passed before accessing any resource, while obviating redundant runtime checks.

The language can be thought of as being parametric in the access-control logic being employed. We have motivated this separation by treating the logic as being a distinct structure from the language. An embedding of proofs and propositions in the logic to constructors and kinds in the language serves as the point of connection between the logic and the language. This clear separation allows for independent development of the two systems.

The idea of separation, however, brings up a question. What kinds of logics can be made to work with the language in the manner employed here? For instance, is it possible to use a classical logic for access-control along with this language? Another interesting question worth asking is: how good is a constructive logic for expressing access-control decisions? Are there other logics better suited? We do not yet know the answers to these.

References

[Aba03] Martin Abadi. Logic in access control. In *LICS '03: Proceedings of the 18th Annual IEEE Symposium on Logic in Computer Science*, page 228, Washington, DC, USA, 2003. IEEE Computer Society.

[Aba06] Martin Abadi. Access control in a core calculus of dependency. *SIGPLAN Not.*, 41(9):263–273, 2006.

[ABHR99] Martin Abadi, Anindya Banerjee, Nevin Heintze, and Jon G. Riecke. A core calculus of dependency. In *POPL '99: Proceedings of the 26th ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, pages 147–160, New York, NY, USA, 1999. ACM Press.

[ABLP93] Martin Abadi, Michael Burrows, Butler Lampson, and Gordon Plotkin. A calculus for access control in distributed systems. *ACM Trans. Program. Lang. Syst.*, 15(4):706–734, 1993.

[BFK98] Matt Blaze, Joan Feigenbaum, and Angelos D. Keromytis. Keynote: Trust management for public-key infrastructures. In *Security Protocols International Workshop*, pages 59–63, Cambridge, England, April 1998. Springer LNCS vol. 1550.

[CA06] Avik Chaudhuri and Martin Abadi. Secrecy by typing and file-access control. In *CSFW '06: Proceedings of the 19th IEEE Workshop on Computer Security Foundations*, pages 112–123, Washington, DC, USA, 2006. IEEE Computer Society.

[CHM05] D. Clark, S. Hunt, and P. Malacaria. Quantitative information flow, relations and polymorphic types. *Journal of Logic and Computation, Special Issue on Lambda-calculus, type theory and natural language*, 18(2):181–199, 2005.

[CHP+07] Karl Crary, Robert Harper, Frank Pfenning, Benjamin C. Pierce, Stephanie Weirich, and Stephan Zdancewic. Manifest security. January 2007.

[DeT02] John DeTreville. Binder, a logic-based security language. In *SP '02: Proceedings of the 2002 IEEE Symposium on Security and Privacy*, page 105, Washington, DC, USA, 2002. IEEE Computer Society.

[FG03] Cedric Fournet and Andrew D. Gordon. Stack inspection: Theory and variants. *ACM Trans. Program. Lang. Syst.*, 25(3):360–399, 2003.

[GBB+06] Deepak Garg, Lujo Bauer, Kevin Bowers, Frank Pfenning, and Michael Reiter. A linear logic of authorization and knowledge. *11th European Symposium on Research in Computer Security*, 2006.

[GP06] Deepak Garg and Frank Pfenning. Non-interference in constructive authorization logic. In *CSFW '06: Proceedings of the 19th IEEE Workshop on Computer Security Foundations*, pages 283–296, Washington, DC, USA, 2006. IEEE Computer Society.

[HL94] Robert Harper and Mark Lillibridge. A type-theoretic approach to higher-order modules with sharing. In *POPL '94: Proceedings of the 21st ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, pages 123–137, New York, NY, USA, 1994. ACM Press.

[HO03] Tomoyuki Higuchi and Atsushi Ohori. A static type system for jvm access control. In *ICFP '03: Proceedings of the eighth ACM SIGPLAN international conference on Functional programming*, pages 227–237, New York, NY, USA, 2003. ACM Press.

[HR98] Nevin Heintze and Jon G. Riecke. The SLam calculus: programming with secrecy and integrity. In *POPL '98: Proceedings of the 25th ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, pages 365–377, New York, NY, USA, 1998. ACM Press.

[KA05] Neel Krishnaswami and Jonathan Aldrich. Permission-based ownership: encapsulating state in higher-order typed languages. In *PLDI '05: Proceedings of the 2005 ACM SIGPLAN conference on Programming language design and implementation*, pages 96–106, New York, NY, USA, 2005. ACM Press.

[LH05] Daniel R. Licata and Robert Harper. A formulation of Dependent ML with explicit equality proofs. Technical Report CMU-CS-05-178, School of Computer Science, Carnegie Mellon University, December 2005.

[Lof96] Per Martin Lof. On the meanings of the logical constants and the justifications of the logical laws. *Nordic Journal of Philosophical Logic*, 1(1):11 – 60, 1996.

[Mal07] Pasquale Malacaria. Assessing security threats of looping constructs. *SIGPLAN Not.*, 42(1):225–235, 2007.

[Mog89] E. Moggi. Computational lambda-calculus and monads. In *Proceedings of the Fourth Annual Symposium on Logic in computer science*, pages 14–23, Piscataway, NJ, USA, 1989. IEEE Press.

[PH04] Sungwoo Park and Robert Harper. A logical view of effects. *Unpublished manuscript*, 2004.

[PSS05] Francois Pottier, Christian Skalka, and Scott Smith. A systematic approach to static access control. *ACM Trans. Program. Lang. Syst.*, 27(2):344–382, 2005.

[RB] R.L.Rivest and B.Lampson. SDSI: A simple distributed security infrastructure.

[SS00] Christian Skalka and Scott Smith. Static enforcement of security with types. In *ICFP '00: Proceedings of the fifth ACM SIGPLAN international conference on Functional programming*, pages 34–45, New York, NY, USA, 2000. ACM Press.

[TZ04] Stephen Tse and Steve Zdancewic. Run-time principals in information-flow type systems. *IEEE Symposium on Security and Privacy*, 00:179, 2004.

A Metatheory

We first prove that the language is type-safe with respect to the operational semantics. In order to state and prove the type safety theorem, we need a new set of judgments stating that an abstract machine state is well-formed. Since the access-control lists provide proof terms, the judgments below consider the abstract machine state augmented with an ACL π.

$$\sigma; \xi; \kappa \rhd C \mathbin{@} \mathbf{w}\ \mathtt{ok}[\pi]$$
$$\sigma; \xi; \kappa \rhd M\ \mathtt{ok}[\pi]$$
$$\sigma; \xi; \kappa \lhd v\ \mathtt{ok}[\pi]$$

The proper way to read the above judgments is: *"the abstract machine state $\sigma; \xi; \kappa \rhd C \mathbin{@} \mathbf{w}$ (and similarly for other states) under the ACL π represents a valid state"*. Next we need to specify that a state is an initial state (i.e. the computation starts from that state) and that a state is final (it terminates the execution sequence).

$\boxed{\text{Abs initial}}$

$$\frac{\Delta \vdash_\Sigma \cdot; \xi; \cdot \rhd C \mathbin{@} \mathbf{w}\ \mathtt{ok}[\pi]}{\Delta \vdash_\Sigma \cdot; \xi; \cdot \rhd C \mathbin{@} \mathbf{w}\ \text{initial}}(C\text{-init}) \qquad \frac{\Delta \vdash_\Sigma \cdot; \xi; \cdot \rhd M\ \mathtt{ok}[\pi]}{\Delta \vdash_\Sigma \cdot; \xi; \cdot \rhd M\ \text{initial}}(M\text{-init})$$

$\boxed{\text{Abs final}}$

$$\frac{\Delta \vdash_\Sigma \sigma; \xi; \cdot \lhd v\ \mathtt{ok}[\pi]}{\Delta \vdash_\Sigma \sigma; \xi; \cdot \lhd v\ \text{final}}(v\text{-final})$$

The judgment for well-formedness of a state $\sigma; \xi; \kappa \rhd C$ checks that the stack κ is well-formed and that C evaluates to a value of the same type as is expected by the top-most frame of κ. We therefore need a new judgment form to state that a stack is well-formed and expects a value of a particular type. This is written as $\kappa : A$.

$\boxed{\Delta \vdash_\Sigma \sigma; \xi; \kappa \rhd C \mathbin{@} \mathbf{w}\ \mathtt{ok}[\pi]}$

$$\frac{\sigma, \Delta \vdash_{\Sigma, \ulcorner\pi\urcorner} \kappa : A \qquad \sigma, \Delta; \cdot \vdash_{\Sigma, \ulcorner\pi\urcorner} C \mathbin{@} \mathbf{w} \sim A \qquad \Delta \vdash_{\Sigma, \ulcorner\pi\urcorner} \sigma\ \mathtt{ok}[\pi] \qquad \sigma, \Delta \vdash_{\Sigma, \ulcorner\pi\urcorner} \xi\ \mathtt{ok}[\pi]}{\Delta \vdash_\Sigma \sigma; \xi; \kappa \rhd C \mathbin{@} \mathbf{w}\ \mathtt{ok}[\pi]}(\text{Abs-}C\text{-ok})$$

In addition, we also need auxiliary judgments stating the well-formedness of the ref-cell and proof stores resp. These are fairly straightforward.

$\boxed{\Delta \vdash_\Sigma \sigma\ \mathtt{ok}[\pi]}$

$$\frac{}{\Delta \vdash_\Sigma \cdot\ \mathtt{ok}[\pi]}(\sigma\text{-ok1}) \qquad \frac{\Delta \vdash_\Sigma \sigma\ \mathtt{ok}[\pi] \qquad \alpha \# \sigma}{\Delta \vdash_\Sigma \sigma, \alpha{::}w \leq \mathbf{w}'\ \mathtt{ok}[\pi]}(\sigma\text{-ok2})$$

$\boxed{\Delta \vdash_\Sigma \xi\ \mathtt{ok}[\pi]}$

$$\frac{}{\Delta \vdash_\Sigma \cdot\ \mathtt{ok}[\pi]}(\xi\text{-ok1}) \qquad \frac{\Delta \vdash_\Sigma \xi\ \mathtt{ok}[\pi] \qquad \Delta; \cdot \vdash_\Sigma v\ \mathtt{value} \qquad \Delta \vdash_\Sigma I :: \mathtt{RES}}{\Delta \vdash_\Sigma \xi[\iota[I] \mapsto v]\ \mathtt{ok}[\pi]}(\xi\text{-ok2})$$

$\boxed{\Delta \vdash_\Sigma \kappa : A}$

$$\frac{}{\Delta \vdash_\Sigma \cdot : A}(\kappa\text{-ok}) \qquad \frac{\Delta \vdash \kappa : A}{\Delta \vdash_\Sigma \kappa \parallel \mathtt{return}\ \bullet\ @\ \mathbf{w} : A}(\kappa\text{-}\mathtt{return})$$

$$\frac{\Delta; x : A \vdash_\Sigma C\ @\ \mathbf{w} \sim A' \qquad \Delta \vdash_\Sigma \kappa : A'}{\Delta \vdash_\Sigma \kappa \parallel \bullet; x.C\ @\ \mathbf{w} : A}(\kappa\text{-}\mathtt{seq})$$

$$\frac{\Delta \vdash_\Sigma \kappa : A' \qquad \Delta, \alpha :: \mathbf{w}' \leq \mathbf{w}; \cdot \vdash_\Sigma C_1\ @\ \mathbf{w} \sim A' \qquad \Delta; \cdot \vdash_\Sigma C_2\ @\ \mathbf{w} \sim A'}{\Delta \vdash_\Sigma \kappa \parallel \mathtt{su}[\mathtt{w}'](\bullet)\{\alpha.C_1 \mid C_2\}\ @\ \mathbf{w} : \mathtt{string}}(\kappa\text{-}\mathtt{su})$$

$$\frac{\Delta \vdash_\Sigma \kappa : A \qquad \Delta; x{:}A' \vdash_\Sigma C\ @\ \mathbf{w} : A}{\Delta \vdash \kappa \parallel \mathtt{letac}\ x = \bullet\ \mathtt{in}\ C\ @\ \mathbf{w} : \mathtt{AC}[\mathtt{w}]A'}(\kappa\text{-}\mathtt{letac})$$

$$\frac{\Delta \vdash_\Sigma \kappa : \mathtt{string} \qquad \Delta \vdash_\Sigma I :: \mathtt{RES} \qquad \Delta \vdash_\Sigma P :: \mathtt{mayrd}(\mathbf{w}, I)}{\Delta \vdash_\Sigma \kappa \parallel \mathtt{read}\ [I][P](\bullet)\ @\ \mathbf{w} : \mathtt{Res}[I]}(\kappa\text{-}\mathtt{read})$$

$$\frac{\Delta \vdash_\Sigma \kappa : \mathbf{1} \qquad \Delta \vdash_\Sigma I :: \mathtt{RES} \qquad \Delta \vdash_\Sigma P :: \mathtt{maywt}(\mathbf{w}, I) \qquad \Delta; \cdot \vdash_\Sigma M : \mathtt{string}}{\Delta \vdash_\Sigma \kappa \parallel \mathtt{write}\ [I][P](\bullet)(M)\ @\ \mathbf{w} : \mathtt{Res}[I]}(\kappa\text{-}\mathtt{write1})$$

$$\frac{\Delta \vdash_\Sigma \kappa : \mathbf{1} \qquad \Delta \vdash_\Sigma I :: \mathtt{RES} \qquad \Delta \vdash_\Sigma P :: \mathtt{maywt}(\mathbf{w}, I) \qquad \Delta; \cdot \vdash_\Sigma v : \mathtt{Res}[I]}{\Delta \vdash_\Sigma \kappa \parallel \mathtt{write}\ [I][P](v)(\bullet)\ @\ \mathbf{w} : \mathtt{string}}(\kappa\text{-}\mathtt{write2})$$

$$\frac{\Delta \vdash_\Sigma \kappa : A_2 \qquad \Delta; \cdot \vdash_\Sigma M : A_1}{\Delta \vdash_\Sigma \kappa \parallel (\bullet)\ M : A_1 \to A_2}(\kappa\text{-app1}) \qquad \frac{\Delta \vdash_\Sigma \kappa : A' \qquad \Delta; x{:}A \vdash_\Sigma M : A'}{\Delta \vdash_\Sigma \kappa \parallel \lambda x{:}A.M\ (\bullet) : A}(\kappa\text{-app2})$$

$\boxed{\Delta \vdash_\Sigma \sigma; \xi; \kappa \rhd M\ \mathtt{ok}[\pi]}$

$$\frac{\Delta \vdash_{\Sigma, \ulcorner\pi\urcorner} \sigma\ \mathtt{ok}[\pi] \qquad \sigma, \Delta; \cdot \vdash_{\Sigma, \ulcorner\pi\urcorner} M : A \qquad \sigma, \Delta \vdash_{\Sigma, \ulcorner\pi\urcorner} \kappa : A \qquad \Delta \vdash_{\Sigma, \ulcorner\pi\urcorner} \xi\ \mathtt{ok}[\pi]}{\Delta \vdash_\Sigma \sigma; \xi; \kappa \rhd M\ \mathtt{ok}[\pi]}(\text{Abs-}M\text{-ok})$$

$\boxed{\Delta \vdash_\Sigma \sigma; \xi; \kappa \lhd v\ \mathtt{ok}[\pi]}$

$$\frac{\Delta \vdash_{\Sigma, \ulcorner\pi\urcorner} \sigma\ \mathtt{ok}[\pi] \qquad \sigma, \Delta; \cdot \vdash_{\Sigma, \ulcorner\pi\urcorner} v\ \mathtt{value} \qquad \sigma, \Delta; \cdot \vdash_{\Sigma, \ulcorner\pi\urcorner} v : A \qquad \sigma, \Delta \vdash_{\Sigma, \ulcorner\pi\urcorner} \kappa : A}{\Delta \vdash_\Sigma \sigma; \xi; \kappa \lhd v\ \mathtt{ok}[\pi]}(\text{Abs-}v\text{-ok})$$

The safety theorem can now be stated as follows:

Progress: (a) If $\Delta \vdash_\Sigma \sigma; \xi; \kappa \rhd C\ @\ \mathbf{w}\ \mathtt{ok}[\pi]$, then there exists a state Abs s.t. $\sigma; \xi; \kappa \rhd C\ @\ \mathbf{w} \mapsto_{\pi;\Sigma}$ Abs.

(b) If $\Delta \vdash_\Sigma \sigma; \xi; \kappa \triangleright \text{Ins} \mathbin{@} \mathbf{w}\ \texttt{ok}[\pi]$, then there exists a state Abs s.t. $\sigma; \xi; \kappa \triangleright \text{Ins} \mathbin{@} \mathbf{w} \mapsto_{\pi;\Sigma}$ Abs.

(c) If $\Delta \vdash_\Sigma \sigma; \xi; \kappa \triangleright M\ \texttt{ok}[\pi]$, then there exists a state Abs s.t. $\sigma; \xi; \kappa \triangleright M \mapsto_{\pi;\Sigma}$ Abs.

(d) If $\Delta \vdash_\Sigma \sigma; \xi; \kappa \triangleleft v\ \texttt{ok}[\pi]$, then there exists a state Abs s.t. $\xi; \kappa \triangleleft v \mapsto_{\pi;\Sigma}$ Abs or $\Delta \vdash_\Sigma \sigma; \xi; \kappa \triangleleft v$ final.

Preservation: (a) If $\Delta \vdash_\Sigma \sigma; \xi; \kappa \triangleright C \mathbin{@} \mathbf{w}\ \texttt{ok}[\pi]$ and $\sigma; \xi; \kappa \triangleright C \mathbin{@} \mathbf{w} \mapsto_{\pi;\Sigma}$ Abs, then $\Delta \vdash_\Sigma$ Abs $\texttt{ok}[\pi]$.

(b) If $\Delta \vdash_\Sigma \sigma; \xi; \kappa \triangleright \text{Ins} \mathbin{@} \mathbf{w}\ \texttt{ok}[\pi]$ and $\sigma; \xi; \kappa \triangleright \text{Ins} \mathbin{@} \mathbf{w} \mapsto_{\pi;\Sigma}$ Abs, then $\Delta \vdash_\Sigma$ Abs $\texttt{ok}[\pi]$.

(c) If $\Delta \vdash_\Sigma \sigma; \xi; \kappa \triangleright M\ \texttt{ok}[\pi]$ and $\sigma; \xi; \kappa \triangleright M \mapsto_{\pi;\Sigma}$ Abs, then $\Delta \vdash_\Sigma$ Abs $\texttt{ok}[\pi]$.

(d) If $\Delta \vdash_\Sigma \sigma; \xi; \kappa \triangleleft v\ \texttt{ok}[\pi]$ and $\sigma; \xi; \kappa \triangleleft v \mapsto_{\pi;\Sigma}$ Abs, then $\Delta \vdash_\Sigma$ Abs $\texttt{ok}[\pi]$.

A.1 Proof of the progress theorem

The progress theorem is easier to prove than the preservation theorem. For part (a), we proceed by analyzing all different cases of C in the abstract machine state $\sigma; \xi; \kappa \triangleright C \mathbin{@} \mathbf{w}$. Part (b) and (c) follow similarly by analyzing different forms of instruction Ins, and of pure terms M resp.. Below we show cases for all the three parts together.

Case: $C = \texttt{return}\ M$

$\sigma; \xi; \kappa \triangleright \texttt{return}\ M \mathbin{@} \mathbf{w} \mapsto_{\pi;\Sigma} \sigma; \xi; \kappa \parallel \texttt{return}\ \bullet \mathbin{@} \mathbf{w} \triangleright M.$

Case: $C = \texttt{letac}\ x = M\ \texttt{in}\ C$

$\xi; \kappa \triangleright \texttt{letac}\ x = M\ \texttt{in}\ C \mathbin{@} \mathbf{w} \mapsto_{\pi;\Sigma} \xi; \kappa \parallel \texttt{letac}\ x = \bullet\ \texttt{in}\ C \mathbin{@} \mathbf{w} \triangleright C \mathbin{@} \mathbf{w}$

Case: $C = \text{Ins}; x.C$

$\sigma; \xi; \kappa \triangleright \text{Ins}; x.C \mathbin{@} \mathbf{w} \mapsto_{\pi;\Sigma} \sigma; \xi; \kappa \parallel \bullet; x.C \mathbin{@} \mathbf{w} \triangleright \text{Ins} \mathbin{@} \mathbf{w}$

Case: $C = \texttt{su}[\mathbf{w}'](M)\{\alpha.C_1 \mid C_2\}$

$\sigma; \xi; \kappa \triangleright \texttt{su}[\mathbf{w}'](M)\{\alpha.C_1 \mid C_2\} \mathbin{@} \mathbf{w} \mapsto_{\pi;\Sigma} \sigma; \xi; \kappa \parallel \texttt{su}[\mathbf{w}'](\bullet)\{\alpha.C_1 \mid C_2\} \mathbin{@} \mathbf{w} \triangleright M$

Case: $C = \texttt{proverd}[I][\mathbf{w}']\{\alpha.C_1 \mid C_2\}$

Again, following the pattern similar to the one in su, we have the following two cases:

Case 1: $\pi \vdash_{\mathcal{L}} m : \texttt{mayrd}(\mathbf{w}', I)$

$\xi; \kappa \triangleright \texttt{proverd}[I][\mathbf{w}']\{\alpha.C_1 \mid C_2\} \mathbin{@} \mathbf{w} \mapsto_{\pi;\Sigma} \xi; \kappa \triangleright [\ulcorner m \urcorner/\alpha]C_1 \mathbin{@} \mathbf{w}$

Case 2: $\xi; \kappa \triangleright \texttt{proverd}[I][\mathbf{w}']\{\alpha.C_1 \mid C_2\} \mathbin{@} \mathbf{w} \mapsto_{\pi;\Sigma} \xi; \kappa \triangleright C_2 \mathbin{@} \mathbf{w}$

Case: $C = \texttt{provewt}[I][\mathbf{w}']\{\alpha.C_1 \mid C_2\}$

Case 1: $\pi \vdash_{\mathcal{L}} m : \texttt{maywt}(\mathbf{w}', I)$

$\xi; \kappa \triangleright \texttt{provewt}[I][\mathbf{w}']\{\alpha.C_1 \mid C_2\} \mathbin{@} \mathbf{w} \mapsto_{\pi;\Sigma} \xi; \kappa \triangleright [\ulcorner m \urcorner/\alpha]C_1 \mathbin{@} \mathbf{w}$

Case 2: $\xi; \kappa \triangleright \texttt{provewt}[I][\mathbf{w}']\{\alpha.C_1 \mid C_2\} \mathbin{@} \mathbf{w} \mapsto_{\pi;\Sigma} \xi; \kappa \triangleright C_2 \mathbin{@} \mathbf{w}$

Case: $\text{Ins} = \texttt{sudo}[\mathbf{w}'][P](C)$

$\xi; \kappa \triangleright \texttt{sudo}[\mathbf{w}'][P](C) \mathbin{@} \mathbf{w} \mapsto_{\pi;\Sigma} \xi; \kappa \triangleright C \mathbin{@} \mathbf{w}'$

Case: $\text{Ins} = \texttt{read}\ [I][P](M)$

$\xi; \kappa \triangleright \texttt{read}\ [I][P](M) \mathbin{@} \mathbf{w} \mapsto_{\pi;\Sigma} \xi; \kappa \parallel \texttt{read}\ [I][P](\bullet) \mathbin{@} \mathbf{w} \triangleright M$

Case: $\text{Ins} = \texttt{write}\ [I][P](M_1)(M_2)$

$\xi; \kappa \triangleright \texttt{write}\ [I][P](M_1)(M_2) \mathbin{@} \mathbf{w} \mapsto_{\pi;\Sigma} \xi; \kappa \parallel \texttt{write}\ [I][P](\bullet)(M_2) \mathbin{@} \mathbf{w} \triangleright M_1$

Case: $M = x$

$\Delta \vdash_{\Sigma} \sigma; \xi; \kappa \rhd x \ \mathtt{ok}[\pi]$	Assumption
$\sigma, \Delta; \cdot \vdash_{\Sigma, \ulcorner \pi \urcorner} x : A$	Inverting Abs-M-ok

Since there is no typing derivation of x in the empty dynamic context, M can not be x. Thus this case cannot occur.

Case: $M = \lambda x : A.M$.

$\sigma; \xi; \kappa \rhd \lambda x : A.M \mapsto_{\pi;\Sigma} \sigma; \xi; \kappa \lhd \lambda x : A.M$

Case: $M = M_1\ M_2$

$\sigma; \xi; \kappa \rhd\ M_1 M_2 \mapsto_{\pi;\Sigma} \sigma; \xi; \kappa \parallel\ (\bullet) M_2 \rhd M_1$

Case: $M = \mathtt{ac}[\mathbf{w}]C$

$\sigma; \xi; \kappa \rhd \mathtt{ac}[\mathbf{w}]C \mapsto_{\pi;\Sigma} \sigma; \xi; \kappa \lhd \mathtt{ac}[\mathbf{w}]C$

Case: $M = \langle\rangle$

$\sigma; \xi; \kappa \rhd \langle\rangle \mapsto_{\pi;\Sigma} \sigma; \xi; \kappa \lhd \langle\rangle$

In order to prove part (d), we first assume that the domain of ref-cell store includes all locations l defined in the signature Σ. We proceed by analyzing all possible values for κ.

Case: $\kappa = \cdot$.

$$\Delta \vdash_{\Sigma} \sigma; \xi; \cdot \lhd v \ \mathtt{ok}[\pi]$$
$$\Delta \vdash_{\Sigma} \sigma; \xi; \cdot \lhd v \text{ final}$$

Case: $\kappa = \kappa' \parallel F$.

We now proceed by induction over the structure of F.

Case: $F = \mathtt{return}\ \bullet\ @\ \mathbf{w}$

$$\sigma; \xi; \kappa' \parallel \mathtt{return}\ \bullet\ @\ \mathbf{w} \lhd v \mapsto_{\pi;\Sigma} \sigma; \xi; \kappa' \lhd v$$

Case: $F = \bullet; x.C\ @\ \mathbf{w}$

$$\sigma; \xi; \kappa' \parallel \bullet; x.C\ @\ \mathbf{w} \lhd v \mapsto_{\pi} \sigma; \xi; \kappa' \rhd [v/x]C\ @\ \mathbf{w}$$

Case: $F = \mathtt{letac}\ x = \bullet\ \mathtt{in}\ C\ @\ \mathbf{w}$

$\sigma, \Delta \vdash_{\Sigma, \ulcorner \pi \urcorner} \kappa' \parallel \mathtt{letac}\ x = \bullet\ \mathtt{in}\ C\ @\ \mathbf{w} : A$	Premise
$A = \mathtt{AC}[\mathbf{w}]A'$	Inversion
$v = \mathtt{ac}[\mathbf{w}]C'$	Canonical forms lemma B.8
$\sigma; \xi; \kappa' \parallel \mathtt{letac}\ x = \bullet\ \mathtt{in}\ C\ @\ \mathbf{w} \lhd \mathtt{ac}[\mathbf{w}]C' \mapsto_{\pi;\Sigma} \sigma; \xi; \kappa' \rhd \langle C'/x \rangle C\ @\ \mathbf{w}$	

Case: $F = \mathtt{su}[\mathbf{w}'](\bullet)\{\alpha.C_1 \mid C_2\}\ @\ \mathbf{w}$

We have the following two cases, corresponding in an actual implementation to checking if the password is correct:

Case: $\sigma; \xi; \kappa' \parallel \mathtt{su}[\mathbf{w}'](\bullet)\{\alpha.C_1 \mid C_2\} \lhd v \mapsto_{\pi} \sigma[\beta : \mathbf{w}' \leq \mathbf{w}]; \xi; \kappa' \rhd [\beta/\alpha]C_1\ @\ \mathbf{w}$ (β fresh).

Case: $\sigma; \xi; \kappa' \parallel \mathtt{su}[\mathbf{w}'](\bullet)\{\alpha.C_1 \mid C_2\} \lhd v \mapsto_{\pi} \sigma; \xi; \kappa' \rhd C_2\ @\ \mathbf{w}$

Case: $F = \texttt{read}\ [I][P](\bullet)\ @\ \mathbf{w}$

$A = \texttt{Res}[I]$ Inversion
$\sigma, \Delta; \cdot \vdash_{\Sigma,\ulcorner\pi\urcorner} v : \texttt{Res}[I]$ Premise
$v = \iota[I]$ Canonical form lemma B.8
$\iota[I] \in dom(\xi)$ Assumption about ref-cell store
$\xi = \xi'[\iota[I] \mapsto v']$
$\sigma; \xi'[\iota[I] \mapsto v']; \kappa' \parallel \texttt{read}\ [I][P](\bullet)\ @\ \mathbf{w} \lhd v \mapsto_\pi \sigma; \xi'[\iota[I] \mapsto v']; \kappa' \rhd v'$

Case: $F = \texttt{write}\ [I][P](\bullet)(M)\ @\ \mathbf{w}$
Proof proceeds exactly the same way as for $\texttt{read}\ [I][P](\bullet)$.

Case: $F = \texttt{write}\ [I][P](v)(\bullet)\ @\ \mathbf{w}$

$A = \texttt{string}$ Inversion
$\sigma, \Delta; \cdot \vdash_{\Sigma,\ulcorner\pi\urcorner} v : \texttt{Res}[I]$ Inversion
$v = \iota[I]$ Canonical form lemma B.8
$\xi = \xi'[\iota[I] \mapsto v']$ Assumption about ref-cell store
$\sigma; \xi'[\iota[I] \mapsto v']; \kappa' \parallel \texttt{write}\ [I][P](v)(\bullet)\ @\ \mathbf{w} \lhd v \mapsto_\pi \sigma; \xi'[\iota[I] \mapsto v]; \kappa' \rhd \langle\rangle$

Case: $F = (\bullet)\ M$

$A = A_1 \to A_2$ Inversion
$v = \lambda x{:}A_1.M'$ Canonical form lemma B.8
$\sigma; \xi; \kappa' \parallel (\bullet)\ M \lhd \lambda x{:}A_1.M' \mapsto_\pi \sigma; \xi; \kappa' \parallel \lambda x{:}A_1.M' \rhd M$

Case: $F = \lambda x{:}A.M\ (\bullet)$
$\sigma; \xi; \kappa' \parallel \lambda x{:}A.M\ (\bullet) \lhd v \mapsto_\pi \sigma; \xi; \kappa' \rhd [v/x]M$

A.2 Proof of preservation theorem

To prove the part (a) of preservation theorem, assume we have the following derivation:

$$\frac{\sigma, \Delta \vdash_{\Sigma,\ulcorner\pi\urcorner} \kappa : A \qquad \sigma, \Delta; \cdot \vdash_{\Sigma,\ulcorner\pi\urcorner} C\ @\ \mathbf{w} \sim A \qquad \Delta \vdash_{\Sigma,\ulcorner\pi\urcorner} \sigma\ \texttt{ok}[\pi] \qquad \Delta \vdash_{\Sigma,\ulcorner\pi\urcorner} \xi\ \texttt{ok}[\pi]}{\Delta \vdash_\Sigma \sigma; \xi; \kappa \rhd C\ @\ \mathbf{w}\ \texttt{ok}[\pi]}$$

We proceed by analyzing the transition relation $\sigma; \xi; \kappa \rhd C\ @\ \mathbf{w} \mapsto_{\pi;\Sigma} Abs$ for various cases of C. To prove part (b) we analyze the relation for various cases Ins.

Case: $\sigma; \xi; \kappa \rhd \texttt{return}\ M\ @\ \mathbf{w} \mapsto_{\pi;\Sigma} \sigma; \xi; \kappa \parallel \texttt{return}\ \bullet\ @\ \mathbf{w} \rhd M$

$\Delta \vdash_\Sigma \sigma; \xi; \kappa \rhd \texttt{return}\ M\ @\ \mathbf{w}\ \texttt{ok}[\pi]$ Assumption
$\Delta \vdash_{\Sigma,\ulcorner\pi\urcorner} \sigma\ \texttt{ok}[\pi]$ and $\Delta \vdash_{\Sigma,\ulcorner\pi\urcorner} \xi\ \texttt{ok}[\pi]$ Inversion
$\sigma, \Delta; \cdot \vdash_{\Sigma,\ulcorner\pi\urcorner} \texttt{return}\ M\ @\ \mathbf{w} \sim A$ Inversion
$\sigma, \Delta \vdash_{\Sigma,\ulcorner\pi\urcorner} \kappa : A$ Inversion
$\sigma, \Delta; \cdot \vdash_{\Sigma,\ulcorner\pi\urcorner} M : A$ Inversion of typing derivation
$\sigma, \Delta \vdash_{\Sigma,\ulcorner\pi\urcorner} \kappa \parallel \texttt{return}\ \bullet\ @\ \mathbf{w} : A$
$\Delta \vdash_\Sigma \sigma; \xi; \kappa \parallel \texttt{return}\ \bullet\ @\ \mathbf{w} \rhd M\ \texttt{ok}[\pi]$

Case: $\sigma;\xi;\kappa \rhd \mathtt{letac}\ x = M\ \mathtt{in}\ C\ @\ \mathbf{w} \mapsto_{\pi;\Sigma} \sigma;\xi;\kappa \parallel \mathtt{letac}\ x = \bullet\ \mathtt{in}\ C\ @\ \mathbf{w} \rhd M$

$\sigma,\Delta \vdash_{\Sigma} \sigma;\xi;\kappa \rhd \mathtt{letac}\ x = M\ \mathtt{in}\ C\ @\ \mathbf{w}\ \mathtt{ok}[\pi]$	Assumption
$\Delta \vdash_{\Sigma,\ulcorner\pi\urcorner} \sigma\ \mathtt{ok}[\pi]$ and $\Delta \vdash_{\Sigma,\ulcorner\pi\urcorner} \xi\ \mathtt{ok}[\pi]$	Inversion
$\sigma,\Delta;\cdot \vdash_{\Sigma,\ulcorner\pi\urcorner} \mathtt{letac}\ x = M\ \mathtt{in}\ C\ @\ \mathbf{w} \sim A$	Inversion
$\sigma,\Delta \vdash_{\Sigma,\ulcorner\pi\urcorner} \kappa : A$	Inversion
$\sigma,\Delta;\cdot \vdash_{\Sigma,\ulcorner\pi\urcorner} M : \mathtt{AC}[\mathbf{w}]A'$	Inversion of typing derivation
$\sigma,\Delta;x{:}A' \vdash_{\Sigma,\ulcorner\pi\urcorner} C\ @\ \mathbf{w} \sim A$	Inversion of typing derivation
$\sigma,\Delta \vdash_{\Sigma,\ulcorner\pi\urcorner} \kappa \parallel \mathtt{letac}\ x = \bullet\ \mathtt{in}\ C\ @\ \mathbf{w} : \mathtt{AC}[\mathbf{w}]A'$	
$\Delta \vdash_{\Sigma} \sigma;\xi;\kappa \parallel \mathtt{letac}\ x = \bullet\ \mathtt{in}\ C\ @\ \mathbf{w} \rhd M\ \mathtt{ok}[\pi]$	

Case: $\sigma;\xi;\kappa \rhd \mathtt{sudo}[\mathbf{w}'][P](C)\ @\ \mathbf{w} \mapsto_{\pi;\Sigma} \sigma;\xi;\kappa \rhd C\ @\ \mathbf{w}'$

$\Delta \vdash_{\Sigma} \sigma;\xi;\kappa \rhd \mathtt{sudo}[\mathbf{w}'][P](C)\ @\ \mathbf{w}\ \mathtt{ok}[\pi]$	Assumption
$\Delta \vdash_{\Sigma,\ulcorner\pi\urcorner} \sigma\ \mathtt{ok}[\pi]$ and $\Delta \vdash_{\Sigma,\ulcorner\pi\urcorner} \xi\ \mathtt{ok}[\pi]$	Inversion
$\sigma,\Delta \vdash_{\Sigma,\ulcorner\pi\urcorner} \kappa : A$	
$\sigma,\Delta;\cdot \vdash_{\Sigma,\ulcorner\pi\urcorner} \mathtt{sudo}[\mathbf{w}'][P](C)\ @\ \mathbf{w} \sim A$	Inversion
$\sigma,\Delta;\cdot \vdash_{\Sigma,\ulcorner\pi\urcorner} C\ @\ \mathbf{w}' \sim A$	Inversion
$\Delta \vdash_{\Sigma} \sigma;\xi;\kappa \rhd C\ @\ \mathbf{w}'\ \mathtt{ok}[\pi]$	

Case: $\sigma;\xi;\kappa \rhd \mathtt{su}[\mathbf{w}'](M)\{\alpha.C_1 \mid C_2\}\ @\ \mathbf{w} \mapsto_{\pi;\Sigma} \sigma;\xi;\kappa \parallel \mathtt{su}[\mathbf{w}'](\bullet)\{\alpha.C_1 \mid C_2\}\ @\ \mathbf{w} \rhd M$

$\Delta \vdash_{\Sigma} \sigma;\xi;\kappa \rhd \mathtt{su}[\mathbf{w}'](M)\{\alpha.C_1 \mid C_2\}\ @\ \mathbf{w}\ \mathtt{ok}[\pi]$	Assumption
$\Delta \vdash_{\Sigma,\ulcorner\pi\urcorner} \sigma\ \mathtt{ok}[\pi]$ and $\Delta \vdash_{\Sigma,\ulcorner\pi\urcorner} \xi\ \mathtt{ok}[\pi]$	Inversion
$\sigma,\Delta \vdash_{\Sigma,\ulcorner\pi\urcorner} \kappa : A$	Inversion
$\sigma,\Delta;\cdot \vdash_{\Sigma,\ulcorner\pi\urcorner} \mathtt{su}[\mathbf{w}'](M)\{\alpha.C_1 \mid C_2\}\ @\ \mathbf{w} \sim A$	Inversion
$\sigma,\Delta,\alpha :: \mathbf{w}' \leq \mathbf{w};\cdot \vdash_{\Sigma,\ulcorner\pi\urcorner} C_1\ @\ \mathbf{w} \sim A$	
$\sigma,\Delta;\cdot \vdash_{\Sigma,\ulcorner\pi\urcorner} C_2\ @\ \mathbf{w} \sim A$	
$\sigma,\Delta;\cdot \vdash_{\Sigma,\ulcorner\pi\urcorner} M : \mathtt{string}$	Inversion
$\sigma,\Delta \vdash_{\Sigma,\ulcorner\pi\urcorner} \kappa \parallel \mathtt{su}[\mathbf{w}'](\bullet)\{\alpha.C_1 \mid C_2\}\ @\ \mathbf{w} : \mathtt{string}$	
$\Delta \vdash_{\Sigma} \sigma;\xi;\kappa \parallel \mathtt{su}[\mathbf{w}'](\bullet)\{\alpha.C_1 \mid C_2\}\ @\ \mathbf{w} \rhd M\ \mathtt{ok}[\pi]$	

Case: $\sigma;\xi;\kappa \rhd \mathtt{proverd}[I][\mathbf{w}']\{\alpha.C_1 \mid C_2\}\ @\ \mathbf{w} \mapsto_{\pi;\Sigma} \sigma;\xi;\kappa \rhd [\ulcorner c\urcorner/\alpha]C_1\ @\ \mathbf{w}$

$\pi \vdash_{\mathcal{L}} m : \mathtt{mayrd}(\mathbf{w}', I)$	Premise
$\Delta \vdash_{\Sigma} \sigma;\xi;\kappa \rhd \mathtt{proverd}[I][\mathbf{w}']\{\alpha.C_1 \mid C_2\}\ @\ \mathbf{w}\ \mathtt{ok}[\pi]$	Inversion
$\sigma,\Delta \vdash_{\Sigma,\ulcorner\pi\urcorner} \kappa : A$	
$\sigma,\Delta;\cdot \vdash_{\Sigma,\ulcorner\pi\urcorner} \mathtt{proverd}[I][\mathbf{w}']\{\alpha.C_1 \mid C_2\}\ @\ \mathbf{w} \sim A$	
$\sigma,\Delta,\alpha :: \mathtt{mayrd}(\mathbf{w}', I);\cdot \vdash_{\Sigma,\ulcorner\pi\urcorner} C_1\ @\ \mathbf{w} \sim A$	
$\sigma,\Delta \vdash_{\Sigma,\ulcorner\pi\urcorner} \ulcorner m\urcorner :: \mathtt{mayrd}(\mathbf{w}', I)$	By Theorem 6.1
$\sigma,\Delta;\cdot \vdash_{\Sigma,\ulcorner\pi\urcorner} [\ulcorner m\urcorner/\alpha]C_1\ @\ \mathbf{w} \sim A$	Substitution Lemma
$\Delta \vdash_{\Sigma} \sigma;\xi;\kappa \rhd [\ulcorner m\urcorner/\alpha]C_1\ @\ \mathbf{w}\ \mathtt{ok}[\pi]$	

Case: $\sigma;\xi;\kappa \rhd \mathtt{proverd}[I][\mathbf{w}']\{\alpha.C_1 \mid C_2\}\ @\ \mathbf{w} \mapsto_{\pi;\Sigma} \sigma;\xi;\kappa \rhd C_2\ @\ \mathbf{w}$

$\Delta \vdash_{\Sigma} \sigma;\xi;\kappa \rhd \mathtt{proverd}[I][\mathbf{w}']\{\alpha.C_1 \mid C_2\}\ @\ \mathbf{w}\ \mathtt{ok}[\pi]$	Inversion
$\sigma,\Delta \vdash_{\Sigma,\ulcorner\pi\urcorner} \kappa : A$	Inversion
$\sigma,\Delta;\cdot \vdash_{\Sigma,\ulcorner\pi\urcorner} \mathtt{proverd}[I][\mathbf{w}']\{\alpha.C_1 \mid C_2\}\ @\ \mathbf{w} \sim A$	
$\sigma,\Delta;\cdot \vdash_{\Sigma,\ulcorner\pi\urcorner} C_2\ @\ \mathbf{w} \sim A$	
$\Delta \vdash_{\Sigma} \sigma;\xi;\kappa \rhd C_2\ @\ \mathbf{w}\ \mathtt{ok}[\pi]$	

Case: $\sigma; \xi; \kappa \rhd \texttt{provewt}[I][\mathbf{w}']\{\alpha.C_1 \mid C_2\} \mathbin{@} \mathbf{w} \mapsto_{\pi;\Sigma} \sigma; \xi; \kappa \rhd [P/\alpha]C_1 \mathbin{@} \mathbf{w}$

$\pi \vdash_{\mathcal{L}} m : \texttt{maywt}(\mathbf{w}', I)$	Premise
$\Delta \vdash_\Sigma \sigma; \xi; \kappa \rhd \texttt{provewt}[I][\mathbf{w}']\{\alpha.C_1 \mid C_2\} \mathbin{@} \mathbf{w}\ \texttt{ok}[\pi]$	Inversion
$\sigma, \Delta \vdash_{\Sigma,\ulcorner\pi\urcorner} \kappa : A$	
$\sigma, \Delta; \cdot \vdash_{\Sigma,\ulcorner\pi\urcorner} \texttt{provewt}[I][\mathbf{w}']\{\alpha.C_1 \mid C_2\} \mathbin{@} \mathbf{w} \sim A$	
$\sigma, \Delta, \alpha :: \texttt{maywt}(\mathbf{w}', I); \cdot \vdash_{\Sigma,\ulcorner\pi\urcorner} C_1 \mathbin{@} \mathbf{w} \sim A$	
$\sigma, \Delta \vdash_{\Sigma,\ulcorner\pi\urcorner} \ulcorner m \urcorner :: \texttt{maywt}(\mathbf{w}', I)$	By Theorem 6.1
$\sigma, \Delta; \cdot \vdash_{\Sigma,\ulcorner\pi\urcorner} [\ulcorner m \urcorner/\alpha]C_1 \mathbin{@} \mathbf{w} \sim A$	Substitution Lemma
$\Delta \vdash_\Sigma \sigma; \xi; \kappa \rhd [\ulcorner m \urcorner/\alpha]C_1 \mathbin{@} \mathbf{w}\ \texttt{ok}[\pi]$	

Case: $\sigma; \xi; \kappa \rhd \texttt{provewt}[I][\mathbf{w}']\{\alpha.C_1 \mid C_2\} \mathbin{@} \mathbf{w} \mapsto_{\pi;\Sigma} \sigma; \xi; \kappa \rhd C_2 \mathbin{@} \mathbf{w}$

$\Delta \vdash_\Sigma \sigma; \xi; \kappa \rhd \texttt{provewt}[I][\mathbf{w}']\{\alpha.C_1 \mid C_2\} \mathbin{@} \mathbf{w}\ \texttt{ok}[\pi]$	Assumption
$\sigma, \Delta \vdash_{\Sigma,\ulcorner\pi\urcorner} \kappa : A$	Inversion
$\sigma, \Delta; \cdot \vdash_{\Sigma,\ulcorner\pi\urcorner} \texttt{provewt}[I][\mathbf{w}']\{\alpha.C_1 \mid C_2\} \mathbin{@} \mathbf{w} \sim A$	Inversion
$\sigma, \Delta; \cdot \vdash_{\Sigma,\ulcorner\pi\urcorner} C_2 \mathbin{@} \mathbf{w} \sim A$	Inversion of typing derivation
$\Delta \vdash_\Sigma \sigma; \xi; \kappa \rhd C_2 \mathbin{@} \mathbf{w}\ \texttt{ok}[\pi]$	

Case: $\sigma; \xi; \kappa \rhd \texttt{read}\ [I][P](M) \mathbin{@} \mathbf{w} \mapsto_{\pi;\Sigma} \sigma; \xi; \kappa \parallel \texttt{read}\ [I][P](\bullet) \mathbin{@} \mathbf{w} \rhd M$

$\Delta \vdash_\Sigma \sigma; \xi; \kappa \rhd \texttt{read}\ [I][P](M) \mathbin{@} \mathbf{w}\ \texttt{ok}[\pi]$	
$\sigma, \Delta; \cdot \vdash_{\Sigma,\ulcorner\pi\urcorner} \texttt{read}\ [I][P](M) \mathbin{@} \mathbf{w} \sim \texttt{string}$	Inversion, twice
$\sigma, \Delta \vdash_{\Sigma,\ulcorner\pi\urcorner} \kappa : \texttt{string}$	
$\sigma, \Delta; \cdot \vdash_{\Sigma,\ulcorner\pi\urcorner} M : \texttt{Res}[I]$	
$\sigma, \Delta \vdash_{\Sigma,\ulcorner\pi\urcorner} I :: \texttt{RES}$	
$\sigma, \Delta \vdash_{\Sigma,\ulcorner\pi\urcorner} P :: \texttt{mayrd}(\mathbf{w}, I)$	
$\sigma, \Delta \vdash_{\Sigma,\ulcorner\pi\urcorner} \kappa \parallel \texttt{read}\ [I][P](\bullet) \mathbin{@} \mathbf{w} : \texttt{Res}[I]$	
$\Delta \vdash_\Sigma \sigma; \xi; \kappa \parallel \texttt{read}\ [I][P](\bullet) \mathbin{@} \mathbf{w} \rhd M\ \texttt{ok}[\pi]$	

Case: $\sigma; \xi; \kappa \rhd \texttt{write}\ [I][P](M_1)(M_2) \mathbin{@} \mathbf{w} \mapsto_{\pi;\Sigma} \sigma; \xi; \kappa \parallel \texttt{write}\ [I][P](\bullet)(M_2) \mathbin{@} \mathbf{w} \rhd M_1$

$\Delta \vdash_\Sigma \sigma; \xi; \kappa \rhd \texttt{write}\ [I][P](M_1)(M_2) \mathbin{@} \mathbf{w}\ \texttt{ok}[\pi]$	
$\sigma, \Delta; \cdot \vdash_{\Sigma,\ulcorner\pi\urcorner} \texttt{write}\ [I][P](M_1)(M_2) \mathbin{@} \mathbf{w} \sim \mathbf{1}$	Inversion, twice
$\sigma, \Delta \vdash_{\Sigma,\ulcorner\pi\urcorner} \kappa : \mathbf{1}$	
$\sigma, \Delta; \cdot \vdash_{\Sigma,\ulcorner\pi\urcorner} M_1 : \texttt{Res}[I]$	
$\sigma, \Delta; \cdot \vdash_{\Sigma,\ulcorner\pi\urcorner} M_2 : \texttt{string}$	
$\sigma, \Delta \vdash_{\Sigma,\ulcorner\pi\urcorner} I :: \texttt{RES}$	
$\sigma, \Delta \vdash_{\Sigma,\ulcorner\pi\urcorner} P :: \texttt{maywt}(\mathbf{w}, I)$	
$\sigma, \Delta \vdash_{\Sigma,\ulcorner\pi\urcorner} \kappa \parallel \texttt{write}\ [I][P](\bullet)(M_2) \mathbin{@} \mathbf{w} : \texttt{Res}[I]$	
$\Delta \vdash_\Sigma \sigma; \xi; \kappa \parallel \texttt{write}\ [I][P](\bullet)(M_2) \mathbin{@} \mathbf{w} \rhd M_1\ \texttt{ok}[\pi]$	

Case: $\sigma; \xi; \kappa \rhd \text{Ins}; x.C \mathbin{@} \mathbf{w} \mapsto_{\pi;\Sigma} \sigma; \xi; \kappa \parallel \bullet; x.C \mathbin{@} \mathbf{w} \rhd \text{Ins} \mathbin{@} \mathbf{w}$

$\Delta \vdash_\Sigma \sigma; \xi; \kappa \rhd \text{Ins}; x.C \mathbin{@} \mathbf{w}\ \texttt{ok}[\pi]$	Assumption
$\sigma, \Delta; \cdot \vdash_{\Sigma,\ulcorner\pi\urcorner} \text{Ins}; x.C \mathbin{@} \mathbf{w} \sim A$	Inversion
$\sigma, \Delta \vdash_{\Sigma,\ulcorner\pi\urcorner} \kappa : A$	Inversion
$\sigma, \Delta; \cdot \vdash_{\Sigma,\ulcorner\pi\urcorner} \text{Ins} \mathbin{@} \mathbf{w} \sim A'$	Inversion
$\sigma, \Delta; x : A' \vdash_{\Sigma,\ulcorner\pi\urcorner} C \mathbin{@} \mathbf{w} \sim A$	Inversion
$\sigma, \Delta \vdash_{\Sigma,\ulcorner\pi\urcorner} \kappa \parallel \bullet; x.C \mathbin{@} \mathbf{w} : A'$	
$\Delta \vdash_\Sigma \sigma; \xi; \kappa \parallel \bullet; x.C \mathbin{@} \mathbf{w} \rhd \text{Ins} \mathbin{@} \mathbf{w}\ \texttt{ok}[\pi]$	

$\boxed{\text{If } \sigma;\xi;\kappa \rhd M \mapsto_{\pi;\Sigma} \text{Abs and } \sigma;\xi;\kappa \rhd M\ \texttt{ok}[\pi],\ \text{then Abs}\ \ \texttt{ok}[\pi]}$

We proceed by analyzing various cases of the transition relation $\sigma;\xi;\kappa \rhd M \mapsto_{\pi;\Sigma}$ Abs:

Case: $\sigma;\xi;\kappa \rhd M_1\ M_2 \mapsto_{\pi;\Sigma} \sigma;\xi;\kappa \parallel (\bullet)\ M_2 \rhd M_1$

$\Delta \vdash_\Sigma \sigma;\xi;\kappa \rhd M_1\ M_2\ \texttt{ok}[\pi]$	Assumption
$\sigma,\Delta \vdash_{\Sigma,\ulcorner\pi\urcorner} \kappa : A$	
$\sigma,\Delta;\cdot \vdash_{\Sigma,\ulcorner\pi\urcorner} M_1\ M_2 : A$	
$\sigma,\Delta;\cdot \vdash_{\Sigma,\ulcorner\pi\urcorner} M_1 : A_2 \to A$	
$\sigma,\Delta;\cdot \vdash_{\Sigma,\ulcorner\pi\urcorner} M_2 : A_2$	
$\sigma,\Delta \vdash_{\Sigma,\ulcorner\pi\urcorner} \kappa \parallel (\bullet)\ M_2 : A_2 \to A$	
$\Delta \vdash_\Sigma \sigma;\xi;\kappa \parallel (\bullet)\ M_2 \rhd M_1\ \texttt{ok}[\pi]$	

Case: $\sigma;\xi;\kappa \rhd v \mapsto_{\pi;\Sigma} \sigma;\xi;\kappa \lhd v$

$\Delta \vdash_\Sigma \sigma;\xi;\kappa \rhd v\ \texttt{ok}[\pi]$	Assumption
$\sigma,\Delta \vdash_{\Sigma,\ulcorner\pi\urcorner} \kappa : A$	
$\sigma,\Delta;\cdot \vdash_{\Sigma,\ulcorner\pi\urcorner} v : A$	
$\sigma,\Delta;\cdot \vdash_{\Sigma,\ulcorner\pi\urcorner} v\ \texttt{value}$	Assumption
$\Delta \vdash_\Sigma \sigma;\xi;\kappa \lhd v\ \texttt{ok}[\pi]$	

$\boxed{\text{If } \sigma;\xi;\kappa \lhd v \mapsto_{\pi;\Sigma}\ \text{Abs, and } \sigma;\xi;\kappa \lhd v\ \texttt{ok}[\pi],\ \text{then Abs}\ \ \texttt{ok}[\pi]}$

As usual, we proceed by analyzing various cases of the transition relation $\sigma;\xi;\kappa \lhd v \mapsto_{\pi;\Sigma}$ Abs:

Case: $\sigma;\xi;\kappa \parallel \texttt{return}\ \bullet\ @\ \mathbf{w} \lhd v \mapsto_{\pi;\Sigma} \sigma;\xi;\kappa \lhd v$

$\Delta \vdash_\Sigma \sigma;\xi;\kappa \parallel \texttt{return}\ \bullet\ @\ \mathbf{w} \lhd v\ \texttt{ok}[\pi]$	Assumption
$\sigma,\Delta \vdash_{\Sigma,\ulcorner\pi\urcorner} \kappa \parallel \texttt{return}\ \bullet\ @\ \mathbf{w} : A$	Inversion
$\Delta \vdash_{\Sigma,\ulcorner\pi\urcorner} \kappa : A$	
$\sigma,\Delta;\cdot \vdash_{\Sigma,\ulcorner\pi\urcorner} v : A$, and $\sigma,\Delta;\cdot \vdash v\ \texttt{value}$	
$\Delta \vdash_\Sigma \sigma;\xi;\kappa \lhd v\ \texttt{ok}[\pi]$	

Case: $\sigma;\xi;\kappa \parallel \texttt{letac}\ x = \bullet\ \texttt{in}\ C\ @\ \mathbf{w} \lhd \texttt{ac}[\mathbf{w}]C' \mapsto_{\pi;\Sigma} \sigma;\xi;\kappa \rhd \langle C'/x\rangle C\ @\ \mathbf{w}$

$\Delta \vdash_\Sigma \sigma;\xi;\kappa \parallel \texttt{letac}\ x = \bullet\ \texttt{in}\ C\ @\ \mathbf{w} \lhd \texttt{ac}[\mathbf{w}]C'\ \texttt{ok}[\pi]$	Assumption
$\sigma,\Delta \vdash_{\Sigma,\ulcorner\pi\urcorner} \kappa \parallel \texttt{letac}\ x = \bullet\ \texttt{in}\ C\ @\ \mathbf{w} : \texttt{AC}[\mathbf{w}]A'$	Inversion, twice
$\sigma,\Delta \vdash_{\Sigma,\ulcorner\pi\urcorner} \kappa : A$	
$\sigma,\Delta;\cdot \vdash_{\Sigma,\ulcorner\pi\urcorner} \texttt{ac}[\mathbf{w}]C' : \texttt{AC}[\mathbf{w}]A'$	
$\sigma,\Delta;\cdot \vdash_{\Sigma,\ulcorner\pi\urcorner} C'\ @\ \mathbf{w} \sim A'$	Inversion of typing
$\sigma,\Delta;x{:}A' \vdash_{\Sigma,\ulcorner\pi\urcorner} C\ @\ \mathbf{w} \sim A$	
$\sigma,\Delta;\cdot \vdash_{\Sigma,\ulcorner\pi\urcorner} \kappa \parallel \langle C'/x\rangle C\ @\ \mathbf{w} : A$	By Lemma B.6
$\Delta \vdash_\Sigma \sigma;\xi;\kappa \rhd \langle C'/x\rangle C\ @\ \mathbf{w}\ \texttt{ok}[\pi]$	

Case: $\sigma;\xi[\iota[I] \mapsto v];\kappa \parallel \texttt{read}\ [I][P](\bullet)\ @\ \mathbf{w} \lhd \iota[I] \mapsto_{\pi;\Sigma} \sigma;\xi[\iota[I] \mapsto v];\kappa \rhd v$

$\Delta \vdash_\Sigma \sigma;\xi[\iota[I] \mapsto v];\kappa \parallel \texttt{read}\ [I][P](\bullet)\ @\ \mathbf{w} \lhd \iota[I]\ \texttt{ok}[\pi]$	Assumption
$\Delta \vdash_{\Sigma,\ulcorner\pi\urcorner} \xi[\iota[I] \mapsto v]\ \texttt{ok}[\pi]$	
$\Delta;\cdot \vdash_{\Sigma,\ulcorner\pi\urcorner} v{:}\texttt{string}$	
$\sigma,\Delta \vdash_{\Sigma,\ulcorner\pi\urcorner} v{:}\texttt{string}$	weakening
$\sigma,\Delta \vdash_{\Sigma,\ulcorner\pi\urcorner} \xi[\iota[I] \mapsto v];\kappa \parallel \texttt{read}\ [I][P](\bullet)\ @\ \mathbf{w} : \texttt{Res}[I]$	Inversion, twice
$\sigma,\Delta \vdash_{\Sigma,\ulcorner\pi\urcorner} \xi[\iota[I] \mapsto v];\kappa : \texttt{string}$	
$\sigma,\Delta;\cdot \vdash_{\Sigma,\ulcorner\pi\urcorner} \iota[I] : \texttt{Res}[I]$	
$\Delta \vdash_\Sigma \sigma;\xi[\iota[I] \mapsto v];\kappa \rhd v\ \texttt{ok}[\pi]$	

Case: $\sigma; \xi; \kappa \parallel \texttt{write}\ [I][P](\bullet)(M_2) \ @\ \mathbf{w} \lhd \iota[I] \mapsto_{\pi;\Sigma} \sigma; \xi; \kappa \parallel \texttt{write}\ [I][P](\iota[I])(\bullet) \ @\ w \rhd M_2$

$$\begin{array}{l}
\sigma, \Delta \vdash_{\Sigma, \ulcorner\pi\urcorner} \kappa \parallel \texttt{write}\ [I][P](\bullet)(M_2) \ @\ \mathbf{w} : \texttt{Res}[I] \\
\sigma, \Delta; \cdot \vdash_{\Sigma, \ulcorner\pi\urcorner} \iota[I] : \texttt{Res}[I] \\
\sigma, \Delta \vdash_{\Sigma, \ulcorner\pi\urcorner} \kappa : \mathbf{1} \\
\sigma, \Delta \vdash_{\Sigma, \ulcorner\pi\urcorner} I :: \texttt{RES} \\
\sigma, \Delta \vdash_{\Sigma, \ulcorner\pi\urcorner} P :: \texttt{maywt}(\mathbf{w}, I) \\
\sigma, \Delta; \cdot \vdash_{\Sigma, \ulcorner\pi\urcorner} M_2 : \texttt{string} \\
\sigma, \Delta \vdash_{\Sigma, \ulcorner\pi\urcorner} \kappa \parallel \texttt{write}\ [I][P](l)(\bullet) \ @\ \mathbf{w} : \texttt{string} \\
\Delta \vdash_{\Sigma, \ulcorner\pi\urcorner} \sigma; \xi; \kappa \parallel \texttt{write}\ [I][P](l)(\bullet) \ @\ \mathbf{w} \rhd M_2\ \texttt{ok}[\pi]
\end{array}$$

Case: $\sigma; \xi[\iota[I] \mapsto v]; \kappa \parallel \texttt{write}\ [I][P](\iota[I])(\bullet) \ @\ \mathbf{w} \lhd v' \mapsto_{\pi;\Sigma} \sigma; \xi[\iota[I] \mapsto v']; \kappa \rhd \langle\rangle$

$$\begin{array}{ll}
\Delta \vdash_{\Sigma} \xi[\iota[I] \mapsto v]; \kappa \parallel \texttt{write}\ [I][P](\iota[I])(\bullet) \ @\ \mathbf{w} \lhd v'\ \texttt{ok}[\pi] & \text{Assumption} \\
\sigma, \Delta \vdash_{\Sigma, \ulcorner\pi\urcorner} \kappa \parallel \texttt{write}\ [I][P](\iota[I])(\bullet) \ @\ \mathbf{w} : \texttt{string} & \text{Inversion, twice} \\
\Delta \vdash_{\Sigma, \ulcorner\pi\urcorner} \xi[\iota[I] \mapsto v]\ \texttt{ok}[\pi] & \text{Inversion} \\
\Delta \vdash_{\Sigma, \ulcorner\pi\urcorner} \xi\ \texttt{ok}[\pi] & \\
\Delta \vdash_{\Sigma, \ulcorner\pi\urcorner} I :: \text{RES and } \Delta; \cdot \vdash_{\Sigma, \ulcorner\pi\urcorner} \iota[I]{:}\texttt{Res}[I] & \\
\sigma, \Delta; \cdot \vdash_{\Sigma, \ulcorner\pi\urcorner} v{:}\texttt{string} & \\
\sigma, \Delta; \cdot \vdash v'{:}\texttt{string} & \text{Premise} \\
\sigma, \Delta \vdash_{\Sigma, \ulcorner\pi\urcorner} \xi[\iota[I] \mapsto v']\ \texttt{ok}[\pi] & \\
\sigma, \Delta \vdash_{\Sigma, \ulcorner\pi\urcorner} \kappa : \mathbf{1} & \\
\sigma, \Delta; \cdot \vdash_{\Sigma, \ulcorner\pi\urcorner} \langle\rangle : \mathbf{1} & \\
\Delta \vdash_{\Sigma} \sigma; \xi[\iota[I] \mapsto v']; \kappa \rhd \langle\rangle\ \texttt{ok}[\pi] &
\end{array}$$

Case: $\sigma; \xi; \kappa \parallel \bullet; x.C \ @\ \mathbf{w} \lhd v \mapsto_{\pi;\Sigma} \sigma; \xi; \kappa \rhd [v/x]C \ @\ \mathbf{w}$

$$\begin{array}{ll}
\Delta \vdash_{\Sigma} \sigma; \xi; \kappa \parallel \bullet; x.C \ @\ \mathbf{w} \lhd v\ \texttt{ok}[\pi] & \text{Assumption} \\
\sigma, \Delta \vdash_{\Sigma, \ulcorner\pi\urcorner} v{:}A & \text{Inversion} \\
\sigma, \Delta \vdash_{\Sigma, \ulcorner\pi\urcorner} v\ \texttt{value} & \text{Inversion} \\
\sigma, \Delta \vdash_{\Sigma, \ulcorner\pi\urcorner} \kappa \parallel \bullet; x.C \ @\ \mathbf{w} : A & \\
\sigma, \Delta \vdash_{\Sigma, \ulcorner\pi\urcorner} \kappa : A' & \\
\sigma, \Delta; x{:}A \vdash_{\Sigma, \ulcorner\pi\urcorner} C \ @\ \mathbf{w} \sim A' & \\
\sigma, \Delta; \cdot \vdash_{\Sigma, \ulcorner\pi\urcorner} [v/x]C \ @\ \mathbf{w} \sim A' & \text{By Lemma B.4} \\
\Delta \vdash_{\Sigma} \sigma; \xi; \kappa \rhd [v/x]C \ @\ \mathbf{w}\ \texttt{ok}[\pi] &
\end{array}$$

Case: $\dfrac{(\beta \text{ fresh})}{\sigma; \xi; \kappa \parallel \text{su}[\mathbf{w}'](\bullet)\{\alpha.C_1 \mid C_2\} \ @\ \mathbf{w} \lhd v \mapsto_{\pi;\Sigma} \sigma[\beta :: \mathbf{w}' \leq \mathbf{w}]; \xi; \kappa \rhd [\beta/\alpha]C_1 \ @\ \mathbf{w}}$

$$\begin{array}{ll}
\Delta \vdash_{\Sigma} \sigma; \xi; \kappa \parallel \text{su}[\mathbf{w}'](\bullet)\{\alpha.C_1 \mid C_2\} \ @\ \mathbf{w} \lhd v\ \texttt{ok}[\pi] & \text{Assumption} \\
\sigma, \Delta \vdash_{\Sigma, \ulcorner\pi\urcorner} v{:}\texttt{string} & \\
\sigma, \Delta \vdash_{\Sigma, \ulcorner\pi\urcorner} \kappa \parallel \text{su}[\mathbf{w}'](\bullet)\{\alpha.C_1 \mid C_2\} \ @\ \mathbf{w} : \texttt{string} & \text{Inversion twice} \\
\sigma, \Delta \vdash_{\Sigma, \ulcorner\pi\urcorner} \kappa : A & \\
\sigma, \Delta, \alpha :: \mathbf{w}' \leq \mathbf{w}; \cdot \vdash_{\Sigma, \ulcorner\pi\urcorner} C_1 \ @\ \mathbf{w} \sim A & \\
\sigma, \Delta, \alpha :: \mathbf{w}' \leq \mathbf{w}, \beta :: \mathbf{w}' \leq \mathbf{w}; \cdot \vdash_{\Sigma, \ulcorner\pi\urcorner} C_1 \ @\ \mathbf{w} \sim A & \text{weakening} \\
\sigma, \Delta, \beta :: \mathbf{w}' \leq \mathbf{w} \vdash_{\Sigma, \ulcorner\pi\urcorner} \beta :: \mathbf{w}' \leq \mathbf{w} & \\
\sigma, \Delta, \beta :: \mathbf{w}' \leq \mathbf{w}; \cdot \vdash_{\Sigma, \ulcorner\pi\urcorner} [\beta/\alpha]C_1 \ @\ \mathbf{w} \sim A & \text{Substitution Lemma} \\
\sigma, \Delta, \beta :: \mathbf{w}' \leq \mathbf{w} \vdash_{\Sigma, \ulcorner\pi\urcorner} \kappa : A & \text{weakening} \\
\Delta \vdash_{\Sigma} \sigma[\beta :: \mathbf{w}' \leq \mathbf{w}]; \xi; \kappa \rhd [\beta/\alpha]C_1 \ @\ \mathbf{w}\ \texttt{ok}[\pi] &
\end{array}$$

Case: $\sigma;\xi;\kappa \parallel \texttt{su}[\mathbf{w}'](\bullet)\{\alpha.C_1 \mid C_2\} @ \mathbf{w} \lhd v \mapsto_{\pi;\Sigma} \sigma;\xi;\kappa \rhd C_2 @ \mathbf{w}$

$$
\begin{array}{ll}
\Delta \vdash_{\Sigma} \sigma;\xi;\kappa \parallel \texttt{su}[\mathbf{w}'](\bullet)\{\alpha.C_1 \mid C_2\} @ \mathbf{w} \lhd v \ \texttt{ok}[\pi] & \text{Assumption} \\
\sigma,\Delta \vdash_{\Sigma,\ulcorner\pi\urcorner} v{:}\texttt{string} & \\
\sigma,\Delta \vdash_{\Sigma,\ulcorner\pi\urcorner} \kappa \parallel \texttt{su}[\mathbf{w}'](\bullet)\{\alpha.C_1 \mid C_2\} @ \mathbf{w} : \texttt{string} & \text{Inversion twice} \\
\sigma,\Delta \vdash_{\Sigma,\ulcorner\pi\urcorner} \kappa : A & \\
\sigma,\Delta \vdash_{\Sigma,\ulcorner\pi\urcorner} C_2 @ \mathbf{w} \sim A & \\
\Delta \vdash_{\Sigma} \sigma;\xi;\kappa \rhd C_2 @ \mathbf{w} \ \texttt{ok}[\pi] &
\end{array}
$$

Case: $\sigma;\xi;\kappa \parallel (\bullet)\ M_2 \lhd \lambda x{:}A.M \mapsto_{\pi;\Sigma} \sigma;\xi;\kappa \parallel (\lambda x{:}A.M)\ (\bullet) \rhd M_2$

$$
\begin{array}{ll}
\Delta \vdash_{\Sigma} \sigma;\xi;\kappa \parallel (\bullet)\ M_2 \lhd \lambda x{:}A.M \ \texttt{ok}[\pi] & \text{Assumption} \\
\sigma,\Delta;\cdot \vdash_{\Sigma,\ulcorner\pi\urcorner} \lambda x{:}A.M : A \to A' & \text{Premise, Inversion} \\
\sigma,\Delta \vdash_{\Sigma,\ulcorner\pi\urcorner} \kappa \parallel (\bullet)\ M_2 : A \to A' & \\
\sigma,\Delta;\cdot \vdash_{\Sigma,\ulcorner\pi\urcorner} M_2 : A & \\
\sigma,\Delta \vdash_{\Sigma,\ulcorner\pi\urcorner} \xi;\kappa \parallel (\lambda x{:}A.M)\ (\bullet) : A & \\
\Delta \vdash_{\Sigma} \sigma;\xi;\kappa \parallel (\lambda x{:}A.M)\ (\bullet) \rhd M_2 \ \texttt{ok}[\pi] &
\end{array}
$$

Case: $\sigma;\xi;\kappa \parallel \lambda x{:}A.M\ (\bullet) \lhd v \mapsto_{\pi;\Sigma} \sigma;\xi;\kappa \rhd [v/x]M$

$$
\begin{array}{ll}
\Delta \vdash_{\Sigma} \sigma;\xi;\kappa \parallel \lambda x{:}A.M\ (\bullet) \lhd v \ \texttt{ok}[\pi] & \text{Assumption} \\
\sigma,\Delta \vdash_{\Sigma,\ulcorner\pi\urcorner} \kappa : A' & \\
\sigma,\Delta; x{:}A \vdash_{\Sigma,\ulcorner\pi\urcorner} M : A' & \\
\sigma,\Delta;\cdot \vdash_{\Sigma,\ulcorner\pi\urcorner} v{:}A & \\
\sigma,\Delta;\cdot \vdash_{\Sigma,\ulcorner\pi\urcorner} [v/x]M : A' & \text{Substitution Lemma} \\
\Delta \vdash_{\Sigma} \sigma;\xi;\kappa \rhd [v/x]M \ \texttt{ok}[\pi] &
\end{array}
$$

B Useful lemmas

First, let us define substitution of types and terms in terms, instructions and computations inductively. The metavariable P stands for a type constructor and α stands for a type variable.

$$
\begin{array}{rcl}
[P/\alpha]\texttt{return}\ M & = & \texttt{return}\ [P/\alpha]M \\
[P/\alpha]\texttt{letac}\ x = M\ \texttt{in}\ C & = & \texttt{letac}\ x = [P/\alpha]M\ \texttt{in}\ [P/\alpha]C \\
[P/\alpha](\text{Ins}; x.C) & = & ([P/\alpha]\text{Ins}); x.[P/\alpha]C \\
[P/\alpha]\texttt{sudo}[\mathbf{w}][P'](C) & = & \texttt{sudo}[\mathbf{w}][[P/\alpha]P']([P/\alpha]C) \\
[P/\alpha]\texttt{su}[\mathbf{w}](M)\{\beta.C_1 \mid C_2\} & = & \texttt{su}[\mathbf{w}]([P/\alpha]M)\{\beta.[P/\alpha]C_1 \mid [P/\alpha]C_2\} \\
[P/\alpha]\texttt{read}\ [I][P'](M) & = & \texttt{read}\ [[P/\alpha]I][[P/\alpha]P']([P/\alpha]M) \\
[P/\alpha]\texttt{write}\ [I][P'](M_1)(M_2) & = & \texttt{write}\ [[P/\alpha]I][[P/\alpha]P']([P/\alpha]M_1)([P/\alpha]M_2) \\
[P/\alpha]\texttt{proverd}[I][\mathbf{w}]\{\beta.C_1 \mid C_2\} & = & \texttt{proverd}[[P/\alpha]I][\mathbf{w}]\{\beta.[P/\alpha]C_1 \mid [P/\alpha]C_2\} \\
[P/\alpha]\texttt{provewt}[I][\mathbf{w}]\{\beta.C_1 \mid C_2\} & = & \texttt{provewt}[[P/\alpha]I][\mathbf{w}]\{\beta.[P/\alpha]C_1 \mid [P/\alpha]C_2\}
\end{array}
$$

Substitution into pure terms is defined as follows:

$$
\begin{array}{rcl}
[P/\alpha]x & = & x \\
[P/\alpha]\lambda x{:}A.M & = & \lambda x{:}[P/\alpha]A.[P/\alpha]M \\
[P/\alpha]M_1\ M_2 & = & [P/\alpha]M_1\ [P/\alpha]M_2 \\
[P/\alpha]\texttt{ac}[\mathbf{w}]C & = & \texttt{ac}[\mathbf{w}][P/\alpha]C \\
[P/\alpha]\iota[I] & = & \iota[[P/\alpha]I] \\
[P/\alpha]\langle\rangle & = & \langle\rangle
\end{array}
$$

Since constructors may depend on other constructors, we define substitution of constructors into constructors:

$$
\begin{array}{rcl}
[P/\alpha]\mathbf{w} & = & \mathbf{w} \\
[P/\alpha]\mathbf{c} & = & \mathbf{c} \\
[P/\alpha]\mathtt{string} & = & \mathtt{string} \\
[P/\alpha](A_1 \to A_2) & = & [P/\alpha]A_1 \to [P/\alpha]A_2 \\
[P/\alpha]\mathtt{AC}[\mathbf{w}]A & = & \mathtt{AC}[\mathbf{w}][P/\alpha]A \\
[P/\alpha]\mathtt{Res}[I] & = & \mathtt{Res}[[P/\alpha]I] \\
[P/\alpha]\alpha & = & P \\
[P/\alpha]\beta & = & \beta \\
[P/\alpha]\mathbf{1} & = & \mathbf{1} \\
[P/\alpha]\star & = & \star \\
[P/\alpha]\langle A_1, A_2\rangle & = & \langle [P/\alpha]A_1, [P/\alpha]A_2\rangle \\
[P/\alpha]\mathtt{fst}\ A & = & \mathtt{fst}\ [P/\alpha]A \\
[P/\alpha]\mathtt{snd}\ A & = & \mathtt{snd}\ [P/\alpha]A \\
[P/\alpha]\lambda\beta{::}K.A & = & \lambda\beta{::}[P/\alpha]K.[P/\alpha]A \\
[P/\alpha]\ A_1 A_2 & = & [P/\alpha]A_1[P/\alpha]A_2
\end{array}
$$

The kinds too depend on constructors. Thus we have:

$$
\begin{array}{rcl}
[P/\alpha]\mathtt{TYPE} & = & \mathtt{TYPE} \\
[P/\alpha]\mathtt{RES} & = & \mathtt{RES} \\
[P/\alpha]\mathbf{w}_1 \le \mathbf{w}_2 & = & \mathbf{w}_1 \le \mathbf{w}_2 \\
[P/\alpha]\mathtt{mayrd}(\mathbf{w}, I) & = & \mathtt{mayrd}(\mathbf{w}, [P/\alpha]I) \\
[P/\alpha]\mathtt{maywt}(\mathbf{w}, I) & = & \mathtt{maywt}(\mathbf{w}, [P/\alpha]I) \\
[P/\alpha](K_1 \to K_2) & = & [P/\alpha]K_1 \to [P/\alpha]K_2 \\
[P/\alpha](K_1 \times K_2) & = & [P/\alpha]K_1 \times [P/\alpha]K_2 \\
[P/\alpha]\top & = & \top
\end{array}
$$

Finally, we define the substitution operation on contexts straightforwardly as:

$$
\begin{array}{rcl}
[P/\alpha]\cdot & = & \cdot \\
[P/\alpha]\Delta, \beta :: K & = & [P/\alpha]\Delta, \beta :: [P/\alpha]K \\
[P/\alpha]\Delta, \alpha :: K & = & [P/\alpha]\Delta \\
\\
[P/\alpha]\cdot & = & \cdot \\
[P/\alpha]\Gamma, x{:}A & = & [P/\alpha]\Gamma, x{:}[P/\alpha]A
\end{array}
$$

Next, we define substitution of terms into instructions and computations.

$$
\begin{array}{rcl}
[M/x]\mathtt{return}\ M' & = & \mathtt{return}\ [M/x]M' \\
[M/x]\mathtt{letac}\ y = M'\ \mathtt{in}\ C & = & \mathtt{letac}\ y = [M/x]M'\ \mathtt{in}\ [M/x]C \\
[M/x]\mathtt{su}[\mathbf{w}](M')\{\alpha.C_1 \mid C_2\} & = & \mathtt{su}[\mathbf{w}]([M/x]M')\{\alpha.[M/x]C_1 \mid [M/x]C_2\} \\
[M/x]\mathtt{proverd}[I][\mathbf{w}]\{\alpha.C_1 \mid C_2\} & = & \mathtt{proverd}[I][\mathbf{w}]\{\alpha.[M/x]C_1 \mid [M/x]C_2\} \\
[M/x]\mathtt{provewt}[I][\mathbf{w}]\{\alpha.C_1 \mid C_2\} & = & \mathtt{provewt}[I][\mathbf{w}]\{\alpha.[M/x]C_1 \mid [M/x]C_2\} \\
[M/x]\mathrm{Ins}; y.C & = & [M/x]\mathrm{Ins}; y.[M/x]C \\
[M/x]\mathtt{sudo}[\mathbf{w}][P](C) & = & \mathtt{sudo}[\mathbf{w}][P]([M/x]C) \\
[M/x]\mathtt{read}\ [I][P](M') & = & \mathtt{read}\ [I][P]([M/x]M')
\end{array}
$$

Finally, the substitution of terms into terms is straightforward. The only interesting case is $\iota[I]$ which does not change upon substitution because there are no free term variables in it.

B.1 Type substitution lemmas

Constructors can be substituted for constructor variables in kinds, constructors, computations and terms. We therefore have the following set of lemmas (we club the context σ, Δ into Δ for the sake of brevity) :

Lemma B.1. *If* $\Delta, \alpha :: K \vdash_\Sigma A_1 :: K_1$*,*
and $\Delta \vdash_\Sigma P :: K$*,*
then $[P/\alpha]\Delta \vdash_\Sigma [P/\alpha]A_1 :: [P/\alpha]K_1$*.*

Proof. By induction over the derivation of $\Delta, \alpha :: K \vdash_\Sigma A_1 :: K_1$. □

Lemma B.2. *If* $\Delta, \alpha :: K; \Gamma \vdash_\Sigma C \mathbin{@} \mathbf{w} \sim A$*,*
and $\Delta \vdash_\Sigma P :: K$*,*
then $[P/\alpha]\Delta; [P/\alpha]\Gamma \vdash_\Sigma [P/\alpha]C \mathbin{@} \mathbf{w} \sim [P/\alpha]A$*.*

Proof. By induction over the derivation of $\Delta, \alpha :: K; \Gamma \vdash_\Sigma C \mathbin{@} \mathbf{w} \sim A$. □

Lemma B.3. *If* $\Delta, \alpha :: K; \Gamma \vdash_\Sigma M{:}A$*,*
and $\Delta \vdash_\Sigma P :: K$*,*
then $[P/\alpha]\Delta; [P/\alpha]\Gamma \vdash_\Sigma [P/\alpha]M{:}[P/\alpha]A$*.*

Proof. By induction over the derivation of $\Delta, \alpha :: K; \Gamma \vdash_\Sigma M{:}A$. □

B.2 Term substitution lemmas

Computations and pure terms depend on other terms. We have the following corresponding substitution lemmas:

Lemma B.4. *The rule*

$$\frac{\Delta; \Gamma, x{:}A \vdash_\Sigma C \mathbin{@} \mathbf{w} \sim A' \qquad \Delta; \Gamma \vdash_\Sigma M{:}A}{\Delta; \Gamma \vdash_\Sigma [M/x]C \mathbin{@} \mathbf{w} \sim A'}$$

is admissible.

Proof. We proceed by induction over the derivation of $\Delta; \Gamma, x{:}A \vdash_\Sigma C \mathbin{@} \mathbf{w} \sim A'$.
(Verified). □

Lemma B.5. *The rule*

$$\frac{\Delta; \Gamma, x{:}A \vdash_\Sigma M'{:}A' \qquad \Delta; \Gamma \vdash_\Sigma M{:}A}{\Delta; \Gamma \vdash_\Sigma [M/x]M'{:}A'}$$

is admissible.

B.3 Leftist substitution lemma

Lemma B.6.

$$\frac{\Delta; \Gamma \vdash_\Sigma C_1 \mathbin{@} \mathbf{w} \sim A_1 \qquad \Delta; \Gamma, x : A_1 \vdash_\Sigma C_2 \mathbin{@} \mathbf{w} \sim A_2}{\Delta; \Gamma \vdash_\Sigma \langle C_1/x \rangle C_2 \mathbin{@} \mathbf{w} \sim A_2}$$

is admissible.

Proof. We proceed by induction over the derivation of the premise $\Delta; \Gamma \vdash_\Sigma C_1 \mathbin{@} \mathbf{w} \sim A_1$.

Case:

$$\frac{\Delta; \Gamma \vdash_\Sigma M : A_1}{\Delta; \Gamma \vdash_\Sigma \mathtt{return}\, M \mathbin{@} \mathbf{w} \sim A_1}$$

$$\Delta; \Gamma \vdash_\Sigma [M/x]C_2 \mathbin{@} \mathbf{w} \sim A_2 \quad \text{By Lemma B.4}$$

Case:

$$\frac{\Delta;\Gamma\vdash_\Sigma M : \mathtt{AC}[\mathbf{w}]A \qquad \Delta;\Gamma,y{:}A\vdash_\Sigma C \mathbin{@} \mathbf{w}\sim A_1}{\Delta;\Gamma\vdash_\Sigma \mathtt{letac}\, y = M \,\mathtt{in}\, C \mathbin{@} \mathbf{w}\sim A_1}$$

($y\#C_2$, since it can always be α-varied.)

$\Delta;\Gamma\vdash_\Sigma M : \mathtt{AC}[\mathbf{w}]A$	Premise
$\Delta;\Gamma,y{:}A\vdash_\Sigma C \mathbin{@} \mathbf{w}\sim A_1$	Premise
$\Delta;\Gamma,x{:}A_1\vdash_\Sigma C_2 \mathbin{@} \mathbf{w}\sim A_2$	Premise
$\Delta;\Gamma,x{:}A_1,y{:}A\vdash_\Sigma C_2 \mathbin{@} \mathbf{w}\sim A_2$	weakening
$\Delta;\Gamma y{:}A\vdash_\Sigma \langle C_1/x\rangle C_2 \mathbin{@} \mathbf{w}\sim A_2$	Ind. hyp.
$\Delta;\Gamma\vdash_\Sigma \mathtt{letac}\, y = M \,\mathtt{in}\, \langle C_1/x\rangle C_2 \mathbin{@} \mathbf{w}\sim A_2$	by rule AC-E

Case:

$$\frac{\Delta;\Gamma\vdash_\Sigma M : \mathtt{string} \qquad \Delta,\alpha :: \mathbf{w}'\leq\mathbf{w};\Gamma\vdash_\Sigma C \mathbin{@} \mathbf{w}\sim A_1 \qquad \Delta;\Gamma\vdash_\Sigma C' \mathbin{@} \mathbf{w}\sim A_1}{\Delta;\Gamma\vdash_\Sigma \mathtt{su}[\mathbf{w}'](M)\{\alpha.C\mid C'\} \mathbin{@} \mathbf{w}\sim A_1}$$

$\alpha \# C_2$	otherwise, $\alpha-$ vary
$\Delta;\Gamma,x:A_1\vdash_\Sigma C_2 \mathbin{@} \mathbf{w}\sim A_2$	Premise
$\Delta,\alpha :: \mathbf{w}'\leq\mathbf{w};\Gamma,x:A_1\vdash_\Sigma C_2 \mathbin{@} \mathbf{w}\sim A_2$	weakening
$\Delta,\alpha :: \mathbf{w}'\leq\mathbf{w};\Gamma\vdash_\Sigma C \mathbin{@} \mathbf{w}\sim A_1$	Premise
$\Delta,\alpha :: \mathbf{w}'\leq\mathbf{w};\Gamma\vdash_\Sigma \langle C/x\rangle C_2 \mathbin{@} \mathbf{w}\sim A_2$	By ind. hyp.
$\Delta;\Gamma\vdash_\Sigma C' \mathbin{@} \mathbf{w}\sim A_1$	Premise
$\Delta;\Gamma\vdash_\Sigma \langle C'/x\rangle C_2 \mathbin{@} \mathbf{w}\sim A_2$	By ind. hyp.
$\Delta;\Gamma\vdash_\Sigma \mathtt{su}[\mathbf{w}'](M)\{\alpha.\langle C/x\rangle C_2\mid \langle C'/x\rangle C_2\} \mathbin{@} \mathbf{w}\sim A_2$	

Case:

$$\frac{\Delta\vdash_\Sigma P :: \mathbf{w}'\leq\mathbf{w} \qquad \Delta;\Gamma\vdash_\Sigma C_1 \mathbin{@} \mathbf{w}'\sim A_1}{\Delta;\Gamma\vdash_\Sigma \mathtt{sudo}[\mathbf{w}'][P](C_1) \mathbin{@} \mathbf{w}\sim A_1}$$

$\langle\mathtt{sudo}[\mathbf{w}'][P](C_1)/x\rangle C_2 = \mathtt{sudo}[\mathbf{w}'][P](C_1);x.C_2$	By definition
$\Delta;\Gamma\vdash_\Sigma \mathtt{sudo}[\mathbf{w}'][P](C_1) \mathbin{@} \mathbf{w}\sim A_1$	Assumption
$\Delta;\Gamma\vdash_\Sigma \mathtt{sudo}[\mathbf{w}'][P](C_1);x.C_2 \mathbin{@} \mathbf{w}\sim A_2$	

Case:

$$\frac{\Delta\vdash_\Sigma I :: \mathtt{RES} \qquad \Delta\vdash_\Sigma P :: \mathtt{mayrd}(\mathbf{w},I) \qquad \Delta;\Gamma\vdash_\Sigma M : \mathtt{Res}[I]}{\Delta;\Gamma\vdash_\Sigma \mathtt{read}\,[I][P](M) \mathbin{@} \mathbf{w}\sim A_1}$$

Similar to sudo case.

Case:

$$\frac{\Delta\vdash_\Sigma I :: \mathtt{RES} \qquad \Delta\vdash_\Sigma P :: \mathtt{maywt}(\mathbf{w},I) \qquad \Delta;\Gamma\vdash_\Sigma M_1 : \mathtt{Res}[I] \qquad \Delta;\Gamma\vdash_\Sigma M_2 : \mathtt{string}}{\Delta;\Gamma\vdash_\Sigma \mathtt{write}\,[I][P](M_1)(M_2) \mathbin{@} \mathbf{w}\sim A_1}$$

Similar to sudo case.

Case:

$$\frac{\Delta\vdash_\Sigma I :: \mathtt{RES} \qquad \Delta,\alpha :: \mathtt{mayrd}(\mathbf{w}',I);\Gamma\vdash_\Sigma C \mathbin{@} \mathbf{w}\sim A_1 \qquad \Delta;\Gamma\vdash_\Sigma C' \mathbin{@} \mathbf{w}\sim A_1}{\Delta;\Gamma\vdash_\Sigma \mathtt{proverd}[I][\mathbf{w}']\{\alpha.C\mid C'\} \mathbin{@} \mathbf{w}\sim A_1}$$

$\alpha \# C_2$	α-vary otherwise
$\Delta;\Gamma,x:A_1\vdash_\Sigma C_2 \mathbin{@} \mathbf{w}\sim A_2$	Assumption
$\Delta,\alpha :: \mathtt{mayrd}(\mathbf{w}',I);\Gamma,x:A_1\vdash_\Sigma C_2 \mathbin{@} \mathbf{w}\sim A_2$	weakening
$\Delta,\alpha :: \mathtt{mayrd}(\mathbf{w}',I);\Gamma\vdash_\Sigma C \mathbin{@} \mathbf{w}\sim A_1$	Premise
$\Delta,\alpha :: \mathtt{mayrd}(\mathbf{w}',I);\Gamma\vdash_\Sigma \langle C/x\rangle C_2 \mathbin{@} \mathbf{w}\sim A_2$	By ind. hyp.
$\Delta;\Gamma\vdash_\Sigma C' \mathbin{@} \mathbf{w}\sim A_1$	Premise
$\Delta;\Gamma\vdash_\Sigma \langle C'/x\rangle C_2 \mathbin{@} \mathbf{w}\sim A_2$	By ind. hyp.
$\Delta;\Gamma\vdash_\Sigma \mathtt{proverd}[I][\mathbf{w}']\{\alpha.\langle C/x\rangle C_2\mid \langle C'/x\rangle C_2\} \mathbin{@} \mathbf{w}\sim A_2$	

Case:

$$\frac{\Delta \vdash_\Sigma I :: \texttt{RES} \qquad \Delta, \alpha :: \texttt{maywt}(\mathbf{w}', I); \Gamma \vdash_\Sigma C \ @\ \mathbf{w} \sim A_1 \qquad \Delta; \Gamma \vdash_\Sigma C' \ @\ \mathbf{w} \sim A_1}{\Delta; \Gamma \vdash_\Sigma \texttt{provewt}[I][\mathbf{w}']\{\alpha.C \mid C'\} \ @\ \mathbf{w} \sim A_1}$$

Exactly similar to proverd case.

□

B.4 Proof term substitution

Apart from the above lemmas about terms in the language, we can prove substitution lemmas for proof terms in the logic. We first define the notion of *replacement* of a constant term by another term. Replacement of a constant c by a term m' in a proof term m (denoted as $[c \to m']m$) replaces all occurrences of c in m by m' avoiding capture of free variables in m'.

Lemma B.7. *(*Proof substitution*)*

1.

$$\frac{\pi, x{:}p' \vdash_{\mathcal{L}} m{:}p \qquad \pi \vdash_{\mathcal{L}} m'{:}p'}{\pi \vdash_{\mathcal{L}} [m'/x]m{:}p}$$

is admissible.

2.

$$\frac{\pi, c{:}p' \vdash_{\mathcal{L}} m{:}p \qquad \pi \vdash_{\mathcal{L}} m'{:}p'}{\pi \vdash_{\mathcal{L}} [c \to m']m{:}p}$$

is admissible.

Proof. By induction on the derivation of $\pi, x{:}p' \vdash_{\mathcal{L}} m{:}p$ and $\pi, c{:}p' \vdash_{\mathcal{L}} m{:}p$ resp. □

B.5 Canonical forms lemma for values

Lemma B.8. *(*Canonical forms*)*

1. *If v is a value of type $A_1 \to A_2$, then v is of the form $\lambda x{:}A_1.M$.*
2. *If v is a value of type $\texttt{AC}[\mathbf{w}]A$, then v is of the form $\texttt{ac}[\mathbf{w}]C$.*
3. *If v is a value of type $\texttt{Res}[I]$, then $v = \iota[I]$.*
4. *If v is a value of type $\mathbf{1}$, then $v = \langle\rangle$.*

Proof. The values v can either be of the $\lambda x A.M$, or $\texttt{ac}[\mathbf{w}]C$, or $\langle\rangle$. For part (1), $\texttt{ac}[\mathbf{w}]C$ and $\langle\rangle$ are ruled out because v is assumed to be of type $A_1 \to A_2$. By inversion lemma, the last two forms of values cannot have such a type. $\lambda x{:}A.M$ gives the desired answer. By inversion again, $A = A_1$.

Proofs for the remaining parts are similar. □

B.6 Inversion lemmas

Lemma B.9. *1. If $\Delta; \Gamma \vdash_\Sigma \texttt{sudo}[\mathbf{w}'][P](C) \ @\ \mathbf{w} \sim A$, then $\Delta \vdash_\Sigma P{::}\mathbf{w}' \leq \mathbf{w}$, and $\Delta; \Gamma \vdash_\Sigma C \ @\ \mathbf{w}' \sim A$.*

2. *If $\Delta; \Gamma \vdash_\Sigma \texttt{read}\ [I][P](M) \ @\ \mathbf{w} \sim \texttt{string}$, then $\Delta \vdash_\Sigma P :: \texttt{mayrd}(\mathbf{w}, I), \Delta \vdash_\Sigma I :: \texttt{RES}$, and $\Delta; \Gamma \vdash_\Sigma M : \texttt{Res}[I]$.*

3. *If* $\Delta;\Gamma \vdash_\Sigma \texttt{write}\ [I][P](M_1)(M_2) \ @\ \mathbf{w} \sim \mathbf{1}$, *then* $\Delta \vdash_\Sigma I :: \texttt{RES}$, $\Delta;\Gamma \vdash_\Sigma M_1 : \texttt{Res}[I]$, $\Delta \vdash_\Sigma P :: \texttt{maywt}(\mathbf{w}, I)$, *and* $\Delta;\Gamma \vdash_\Sigma M_2 : \texttt{string}$

4. *If* $\Delta;\Gamma \vdash_\Sigma \texttt{letac}\ x = M\ \texttt{in}\ C \ @\ \mathbf{w} \sim A$, *then* $\Delta;\Gamma, x{:}A_1 \vdash_\Sigma C \ @\ \mathbf{w} \sim A$, $\Delta;\Gamma \vdash_\Sigma M : \texttt{AC}[\mathbf{w}]A_1$.

5. *If* $\Delta;\Gamma \vdash_\Sigma \texttt{return}\ M \ @\ \mathbf{w} \sim A$, *then* $\Delta;\Gamma \vdash_\Sigma M : A$.

6. *If* $\Delta;\Gamma \vdash_\Sigma \texttt{su}[\mathbf{w}'](M)\{\alpha.C_1 \mid C_2\} \ @\ \mathbf{w} \sim A$, *then* $\Delta;\Gamma \vdash_\Sigma M : \texttt{string}$, $\Delta, \alpha :: \mathbf{w}' \leq \mathbf{w};\Gamma \vdash_\Sigma C_1 \ @\ \mathbf{w} \sim A$ *and* $\Delta;\Gamma \vdash_\Sigma C_2 \ @\ \mathbf{w} \sim A$.

7. *If* $\Delta;\Gamma \vdash_\Sigma \texttt{proverd}[I][\mathbf{w}']\{\alpha.C_1 \mid C_2\} \ @\ \mathbf{w} \sim A$, *then* $\Delta, \alpha :: \texttt{mayrd}(\mathbf{w}', I);\Gamma \vdash_\Sigma C_1 \ @\ \mathbf{w} \sim A$, $\Delta;\Gamma \vdash_\Sigma C_2 \ @\ \mathbf{w} \sim A$.

8. *If* $\Delta;\Gamma \vdash_\Sigma \texttt{provewt}[I][\mathbf{w}']\{\alpha.C_1 \mid C_2\} \ @\ \mathbf{w} \sim A$, *then* $\Delta, \alpha :: \texttt{maywt}(\mathbf{w}', I);\Gamma \vdash_\Sigma C_1 \ @\ \mathbf{w} \sim A, \Delta;\Gamma \vdash_\Sigma C_2 \ @\ \mathbf{w} \sim A$,

9. *If* $\Delta;\Gamma \vdash_\Sigma \mathit{Ins}; x.C \ @\ \mathbf{w} \sim A$, *then* $\Delta;\Gamma \vdash_\Sigma \mathit{Ins} \ @\ \mathbf{w} \sim A', \Delta;\Gamma, x : A' \vdash_\Sigma C \ @\ \mathbf{w} \sim A$

10. *If* $\Delta;\Gamma \vdash_\Sigma \lambda x{:}A.M : A_1 \to A_2$, *then* $\Delta;\Gamma, x{:}A_1 \vdash_\Sigma M : A_2$.

11. *If* $\Delta;\Gamma \vdash_\Sigma M_1 M_2 : A_2$, *then* $\Delta;\Gamma \vdash_\Sigma M_1 : A_1 \to A_2$, *and* $\Delta;\Gamma \vdash_\Sigma M_2 : A_1$

12. *If* $\Delta;\Gamma \vdash_\Sigma \texttt{ac}[\mathbf{w}]C : \texttt{AC}[\mathbf{w}]A$, *then* $\Delta;\Gamma \vdash_\Sigma C \ @\ \mathbf{w} \sim A$.

13. *If* $\Delta;\Gamma \vdash_\Sigma \langle\rangle : A$, *then* $A = \mathbf{1}$.

C Access-control safety

In order to talk about access-control, we need to show the existence of proofs in the access-control theory whenever the corresponding propositional kind in the programming language is inhabited. Often it is easy to construct a proof from the constructor. For this we define a mapping ($\llcorner\cdot\lrcorner$) from constructors to proofs and from kinds to propositions in the logic.

In the case of kinds, we map all the kinds that are in the image of the embedding $\ulcorner\cdot\urcorner$ back to their preimages under $\ulcorner\cdot\urcorner$. We call such kinds *propositional*. On the other hand, we will refer to those kinds that do not have a propositional subkind as *non-propositional*. Note that there may be kinds which are neither propositional nor non-propositional.

$$
\begin{array}{lcl}
\llcorner \texttt{TYPE} \lrcorner & = & \top \\
\llcorner \texttt{RES} \lrcorner & = & \top \\
\llcorner \mathbf{w}_1 \leq \mathbf{w}_2 \lrcorner & = & \top \\
\llcorner \texttt{mayrd}(\mathbf{w}, I) \lrcorner & = & \texttt{mayrd}(\mathbf{w}, I) \\
\llcorner \texttt{maywt}(\mathbf{w}, I) \lrcorner & = & \texttt{maywt}(\mathbf{w}, I) \\
\llcorner K_1 \to K_2 \lrcorner & = & \llcorner K_1 \lrcorner \supset \llcorner K_2 \lrcorner \\
\llcorner K_1 \times K_2 \lrcorner & = & \llcorner K_1 \lrcorner \wedge \llcorner K_2 \lrcorner \\
\llcorner \top \lrcorner & = & \top \\
\llcorner \mathbf{c} \lrcorner & = & \mathbf{c} \\
\llcorner \alpha \lrcorner & = & \alpha \\
\llcorner A_1 \to A_2 \lrcorner & = & \langle\rangle_{\mathcal{L}} \\
\llcorner \texttt{AC}[\mathbf{w}]A \lrcorner & = & \langle\rangle_{\mathcal{L}} \\
\llcorner \texttt{Res}[I] \lrcorner & = & \langle\rangle_{\mathcal{L}} \\
\llcorner \langle\rangle \lrcorner & = & \langle\rangle_{\mathcal{L}} \\
\llcorner \star \lrcorner & = & \langle\rangle_{\mathcal{L}} \\
\llcorner \langle P_1, P_2 \rangle \lrcorner & = & \langle \llcorner P_1 \lrcorner, \llcorner P_2 \lrcorner \rangle_{\mathcal{L}} \\
\llcorner \texttt{fst}\ P \lrcorner & = & \texttt{fst}_{\mathcal{L}} \llcorner P \lrcorner \\
\llcorner \texttt{snd}\ P \lrcorner & = & \texttt{snd}_{\mathcal{L}} \llcorner P \lrcorner \\
\llcorner \lambda\alpha{::}K.A \lrcorner & = & \lambda_{\mathcal{L}}\alpha{:}\llcorner K \lrcorner.\llcorner A \lrcorner \\
\llcorner A_1 A_2 \lrcorner & = & \texttt{app}_{\mathcal{L}} \llcorner A_1 \lrcorner \llcorner A_2 \lrcorner
\end{array}
$$

Non-propositional kinds are defined inductively by the judgment K nonp.

$$\texttt{TYPE}\ \text{nonp} \qquad \texttt{RES}\ \text{nonp} \qquad \mathbf{w}_1 \leq \mathbf{w}_2\ \text{nonp} \qquad \frac{K_1\ \text{nonp} \quad K_2\ \text{nonp}}{K_1 \to K_2\ \text{nonp}} \qquad \frac{K_1\ \text{nonp} \quad K_2\ \text{nonp}}{K_1 \times K_2\ \text{nonp}}$$

Lemma C.1. *If K nonp, then $\llcorner K \lrcorner \in p_\top$, where $p_\top$ is defined by the grammar:*

$$p_\top ::= \top \mid p_\top \supset p_\top \mid p_\top \wedge p_\top$$

Proof. By induction on the derivation of K nonp. □

$p_\top$ is a subset of trivially true propositions. Any hypothesis regarding a proposition in $p_\top$ can be safely omitted while still maintaining completeness of deductions.

It is easy to verify that $\llcorner\cdot\lrcorner$ is the right-inverse of $\ulcorner\cdot\urcorner$.

Lemma C.2. 1. *If $\Delta \vdash_\Sigma K\ \texttt{kind}$, then $\llcorner K \lrcorner$ is a proposition.*

2. *If $\Delta \vdash_\Sigma A :: K$, then $\llcorner \Delta \lrcorner, \llcorner \Sigma \lrcorner \vdash_{\mathcal{L}} \llcorner A \lrcorner : \llcorner K \lrcorner$.*

Proof. By induction on derivation of $\Delta \vdash_\Sigma K\ \texttt{kind}$ and $\Delta \vdash_\Sigma A :: K$ resp. □

Theorem C.1. (Safety of resource-access):

1. *If $\cdot \vdash_\Sigma C \mathbin{@} \mathbf{w} \sim A$, $\cdot \vdash_\Sigma \xi\ \texttt{ok}[\pi]$ and $\cdot; \xi; \cdot \rhd C \mathbin{@} \mathbf{w} \mapsto^\star_{\pi;\Sigma} \sigma; \xi'; \kappa \rhd \texttt{read}\ [I][P](M) \mathbin{@} \mathbf{w}'$, then there exists a proof p such that $\pi \vdash_{\mathcal{L}} p : \texttt{mayrd}(\mathbf{w}', I)$.*

2. *If $\cdot \vdash_\Sigma C \mathbin{@} \mathbf{w} \sim A$, $\cdot \vdash_\Sigma \xi\ \texttt{ok}[\pi]$ and $\cdot; \xi; \cdot \rhd C \mathbin{@} \mathbf{w} \mapsto^\star_{\pi;\Sigma} \sigma; \xi'; \kappa \rhd \texttt{write}\ [I][P](M)(\ @\ )\mathbf{w}'$, then there exists a proof p such that $\pi \vdash_{\mathcal{L}} p : \texttt{maywt}(\mathbf{w}', I)$.*

Proof. We specifically assume that proofs of logical propositions are introduced into the language only through the access-control theory. That is to say, for all $c :: K \in \Sigma$, K nonp.

We now sketch the proof of part(1). The proof for the second part is similar.

$\cdot \vdash_\Sigma C \mathbin{@} \mathbf{w} \sim A$	Assumption
$\cdot \vdash_\Sigma \xi\ \texttt{ok}[\pi]$	Assumption
$\cdot; \xi; \cdot \rhd C \mathbin{@} \mathbf{w}\ \texttt{ok}[\pi]$	By rule Abs-C-ok
$\cdot; \xi; \cdot \rhd C \mathbin{@} \mathbf{w} \mapsto^\star_{\pi;\Sigma} \sigma; \xi'; \kappa \rhd \texttt{read}\ [I][P](M) \mathbin{@} \mathbf{w}'$	Assumption
$\sigma; \xi'; \kappa \rhd \texttt{read}\ [I][P](M) \mathbin{@} \mathbf{w}'\ \texttt{ok}[\pi]$	By preservation
$\cdot \vdash_{\Sigma, \ulcorner\pi\urcorner} P :: \texttt{mayrd}(\mathbf{w}', I)$	By inversion of Abs-C-ok, and of typing derivation
$\llcorner\Sigma\lrcorner, \llcorner\ulcorner\pi\urcorner\lrcorner \vdash_{\mathcal{L}} \llcorner P \lrcorner : \texttt{mayrd}(\mathbf{w}', I)$	By Lemma C.2
$\llcorner\Sigma\lrcorner, \pi \vdash_{\mathcal{L}} \llcorner P \lrcorner : \texttt{mayrd}(\mathbf{w}', I)$	$\llcorner\cdot\lrcorner$ is the right inverse of $\ulcorner\cdot\urcorner$

Having obtained a $\llcorner P \lrcorner$, we can easily transform it to a proof in the context π (instead of $\llcorner\Sigma\lrcorner, \pi$) by replacing all the constants $c : K$ from Σ in $\llcorner P \lrcorner$ by canonical proofs of $\llcorner K \lrcorner$. For example, the canonical proof of $\top$ is $\langle\rangle_{\mathcal{L}}$, that of $\top \to (\top \to \top)$ is $\lambda_{\mathcal{L}} u{:}\top.\lambda_{\mathcal{L}} v{:}\top.\top$. The substitution lemma B.7 ensures that we get the proof of the same proposition. □

Theorem C.2. (Access-control safety): *If $\pi, \Delta; \cdot \vdash_\Sigma C \mathbin{@} \mathbf{w} \sim A$ and $\cdot; \xi; \cdot \rhd C \mathbin{@} \mathbf{w} \mapsto^\star_{\pi;\Sigma}$ Abs, then for each $\mathbf{w}_i$ active in Abs, either $\mathbf{w}_i = w$, or there exist sequences $\langle \mathbf{w}_1, \ldots, \mathbf{w}_{i_n} \rangle, \langle P_1, \ldots, P_{i_n - 1} \rangle$, such that $\mathbf{w}_1 = w, \mathbf{w}_{i_n} = \mathbf{w}_i$ and $\sigma, \Delta \vdash_{\Sigma, \ulcorner\pi\urcorner} P_j :: \mathbf{w}_{j+1} \leq \mathbf{w}_j$ for $1 \leq j < i_n$, where σ is the Su-permissions component of Abs.*

Proof. We proceed by induction on the number of steps of the operational semantics it takes to reach Abs from $\cdot; \xi; \cdot \rhd C \mathbin{@} \mathbf{w}$.

For the base case we analyze all possible cases of the computation C:

Case: $\texttt{return}\, M$

$\cdot; \xi; \cdot \rhd \texttt{return}\, M \mathbin{@} \mathbf{w} \mapsto_{\pi;\Sigma} \cdot; \xi; \cdot \parallel \texttt{return}\ \bullet\ \mathbin{@} \mathbf{w} \rhd M$

The only active principal after one step is $\mathbf{w}$.

Case: $\texttt{letac}\, x = M\ \texttt{in}\, C$.

$\cdot; \xi; \cdot \rhd \texttt{letac}\, x = M\ \texttt{in}\, C \mathbin{@} \mathbf{w} \mapsto_{\pi;\Sigma} \cdot; \xi; \cdot \parallel \texttt{letac}\, x = \bullet\ \texttt{in}\, C \mathbin{@} \mathbf{w} \rhd M$

The only active principal is $\mathbf{w}$.

Case: $\texttt{su}[\mathbf{w}'](M)\{\alpha.C_1 \mid C_2\}$

$\cdot; \xi; \cdot \rhd \texttt{su}[\mathbf{w}'](M)\{\alpha.C_1 \mid C_2\} \mathbin{@} \mathbf{w} \mapsto_{\pi;\Sigma} \cdot; \xi; \cdot \parallel \texttt{su}[\mathbf{w}'](\bullet)\{\alpha.C_1 \mid C_2\} \mathbin{@} \mathbf{w} \rhd M$.

The only active principal is $\mathbf{w}$.

We do not show rest of the cases. In all of them, the only active principal after one step is $\mathbf{w}$: the principal that the computation was started with.

Induction step: Assume that the induction hypothesis holds for transitions of length n.
Let $\cdot; \xi; \cdot \rhd C \mathbin{@} \mathbf{w}_0 \mapsto^n_{\pi;\Sigma} \text{Abs}_1 \mapsto_{\pi;\Sigma} \text{Abs}_2$, where Abs_1 is the state after n transitions. We now induct on the derivation of the transition step $\text{Abs}_1 \mapsto_{\pi;\Sigma} \text{Abs}_2$. Here we only show the interesting cases where the set of active principals changes. In all remaining cases, the set of active principals in Abs_2 is the same as those in Abs_1, and hence by induction hypothesis, the theorem holds in Abs_2.

Case: $\sigma; \xi; \kappa \rhd \texttt{sudo}[\mathbf{w}'][P](C') \mathbin{@} \mathbf{w} \mapsto_{\pi;\Sigma} \sigma; \xi; \kappa \rhd C' \mathbin{@} \mathbf{w}'$

By I.H., we have a chain of proofs from $\mathbf{w}_0$ to all active principals in κ. Also, by I.H., let $\langle P_1, P_2, \ldots, P_i\rangle$ be the chain of proofs from $\mathbf{w}_0$ to $\mathbf{w}$.

$\Delta; \cdot \vdash_{\Sigma, \ulcorner\pi\urcorner} C \mathbin{@} \mathbf{w}_0 \sim A$	Assumption
$\sigma; \xi; \kappa \rhd \texttt{sudo}[\mathbf{w}'][P](C') \mathbin{@} w\ \texttt{ok}[\pi]$	By preservation
$\sigma, \Delta; \cdot \vdash_{\Sigma, \ulcorner\pi\urcorner} \texttt{sudo}[\mathbf{w}'][P](C') \mathbin{@} \mathbf{w} \sim A'$	Inversion of the derivation of $\Delta \vdash$ Abs $\texttt{ok}[\pi]$
$\sigma, \Delta \vdash_{\Sigma, \ulcorner\pi\urcorner} P :: \mathbf{w}' \leq \mathbf{w}$	Inversion of typing derivation

Thus $\langle P_1, P_2, \ldots, P_i, P\rangle$ forms a chain of proofs of accessibility from $\mathbf{w}_0$ to $\mathbf{w}'$. The rest of the principals active in Abs_2 are all active in Abs_1 and we have chains of proofs for them by induction hypothesis.

□