

Run-time Principals in Information-flow Type Systems

STEPHEN TSE and STEVE ZDANCEWIC

University of Pennsylvania

Information-flow type systems are a promising approach for enforcing strong end-to-end confidentiality and integrity policies. Such policies, however, are usually specified in terms of static information—data is labeled *high* or *low* security at compile time. In practice, the confidentiality of data may depend on information available only while the system is running.

This paper studies language support for *run-time principals*, a mechanism for specifying security policies that depend on which principals interact with the system. We establish the basic property of noninterference for programs written in such language, and use run-time principals for specifying run-time authority in downgrading mechanisms such as declassification.

In addition to allowing more expressive security policies, run-time principals enable the integration of language-based security mechanisms with other existing approaches such as Java stack inspection and public key infrastructures. We sketch an implementation of run-time principals via public keys such that principal delegation is verified by certificate chains.

Categories and Subject Descriptors: D.3.3 [Programming Languages]: Language Constructs and Features

General Terms: Design, Languages, Security

Additional Key Words and Phrases: decentralized label model, dynamic principals, information-flow, noninterference, run-time principals, security-typed, soundness, type systems

Stephen Tse (stse@cis.upenn.edu) and Steve Zdancewic (stevez@cis.upenn.edu).

Authors' address: 3330 Walnut Street, Computer and Information Science Dept., University of Pennsylvania, Philadelphia, PA 19104, US.

An earlier version of this paper with the same title appears in IEEE Symposium on Security and Privacy, 2004 [Tse and Zdancewic 2004].

Permission to make digital/hard copy of all or part of this material without fee for personal or classroom use provided that the copies are not made or distributed for profit or commercial advantage, the ACM copyright/server notice, the title of the publication, and its date appear, and notice is given that copying is by permission of the ACM, Inc. To copy otherwise, to republish, to post on servers, or to redistribute to lists requires prior specific permission and/or a fee.

© 2004 ACM 0164-0925/99/0100-0111 \$0.75

Contents

1	Introduction	2
2	Decentralized label model	4
2.1	Principals and labels	4
2.2	Acts-for hierarchy	5
2.3	Label lattice	6
3	Run-time principals	7
3.1	Dynamic semantics	9
3.2	Static semantics	12
3.3	Noninterference	13
4	Declassification and authority	20
4.1	Run-time authority and capabilities	21
4.2	Endorsement and delegation	27
4.3	Acquiring capabilities	28
5	Type-safety	28
5.1	Progress	29
5.2	Preservation	32
6	PKI and application	35
6.1	Public key infrastructures	35
6.2	Application to distributed banking	37
7	Discussion	39
7.1	Related work	39
7.2	Conclusions	40
A	Full syntax	43

1. INTRODUCTION

Information-flow type systems are a promising approach for enforcing strong end-to-end confidentiality and integrity policies [Sabelfeld and Myers 2003]. However, most previous work on these security-typed languages has used simplistic ways of specifying policies: the programmer specifies during program development what data is confidential and what data is public. These information-flow policies constrain which principals have access either directly, or indirectly, to the labeled data.

In practice, however, policies are more complex—the principals that own a piece of data may be unknown at compile time or may change over time, and the security policy itself may require such run-time information to downgrade confidential data. This paper addresses these shortcomings and studies *run-time principals* in the context of information-flow policies.

Run-time principals are first-class data values representing users, groups, etc. During its execution, a program may inspect a run-time principal to determine policy information not available when the program was compiled. The key problem

is designing the language in such a way that the dynamic checks required to implement run-time principals introduce no additional covert channels. Moreover, while adding run-time principals permits new kinds of security policies, the new policies should still interact well with the static type checking.

Run-time principals provide a way of integrating the policies expressed by the type system with external notions of principals such as those found in public key infrastructures (PKI). This integration allows language-based security mechanisms to interoperate with existing machinery such as the access control policies enforced by a file system or the authentication provided by an OS.

This paper makes the following three contributions:

- We formalize run-time principals in a simple security-typed language based on the λ -calculus and show that the type system enforces *noninterference*, a strong information-flow guarantee. This type system is intended to serve as a theoretical foundation for realistic languages such as Jif [Myers et al. 1999] and FlowCaml [Simonet 2003].
- We consider the problems of *downgrading* and *delegation* in the presence of run-time principals and propose the concept of *run-time authority* to temper their use. Declassification, and other operations that reveal information owned by a run-time principal, may only be invoked when the principal has granted the system appropriate rights. These capabilities must be verified at runtime, leading to a mechanism reminiscent of (but stronger than) Java’s stack inspection [Wallach and Felten 1998; Wallach et al. 2000].
- We investigate the implementation of run-time principals via public key infrastructures. Run-time principals are represented by public keys, run-time authority corresponds to digitally signed capabilities, and the delegation relation between principals can be determined from certificate chains.

As an example of an information-flow policy permitted by run-time principals, consider this program that manipulates data confidential to both a company manager and to less privileged employees:

```

1    class C {
2        final principal user = Runtime.getUser();
3        void print(String{user:} s) {...}
4        void printIfManager(String{Manager:} s) {
5            actsFor (user, Manager) {
6                print(s);
7            }
8        }}

```

This program, written in a Java-like notation, calls the `print` routine to display a string on the terminal. The run-time principal `user`, whose value is determined dynamically (`Runtime.getUser`), represents the user that initiated the program. Note that, in addition to ordinary datatypes such as Java’s `String` objects, there is a new basic type, `principal`; values of type `principal` are run-time principals.

Lines 3-4 illustrate how information-flow type systems constrain information-flows using labels. The argument to the `print` method is a `String` object `s` that has the security label `{user:}`. In the decentralized label model [Myers and Liskov

1998; 2000], this annotation indicates that `s` is *owned* by the principal `user` and that the policy of `user` is that no other principals can *read* the contents of `s`. This policy annotation might be appropriate when the `Strings` passed to the `print` method are output on a terminal visible to the principal `user`. More importantly, confidential information such as `Manager`'s password, which `user` is *not* permitted to see, cannot be passed to the `print` method (either directly or indirectly). Here, `Manager` is a principal *constant* (a fixed value determined at compile time), and `user` is a principal *variable* (a dynamic value to be determined at run time). The type system of the programming language enforces such information-flow policies at compile time without run-time penalty.

The `printIfManager` method illustrates how run-time principals can allow for more expressive security policies. This method also takes a `String` as input but, unlike `print`, requires the string to have the label `{Manager:}`, meaning that the data is owned and readable only by the principal `Manager` and principals that act for `Manager`. The body of this method performs a run-time test to determine whether the `user` principal that has initiated the program is in fact acting for the `Manager` principal. If so, then `s` is printed to the terminal, which is secure because the `user` has the privileges of `Manager`. Otherwise `s` is not printed. Without such a run-time test, an information-flow type system would prevent a `String{Manager:}` object from being sent to the `print` routine because it expects a `String{user:}` object. Run-time principals allow such security policies that depend on the execution environment.

Although this example has been explained in terms of Java-like syntax, we carry out our formal analysis of run-time principals in terms of a typed λ -calculus. This choice allows us to emphasize the new features of run-time principals and to use established proof techniques for noninterference [Heintze and Riecke 1998; Abadi et al. 1999; Pottier and Simonet 2002; Zdancewic and Myers 2002]. It should be possible to extend our results to Java-like languages by using the techniques of Banerjee and Naumann [2002; 2003].

The rest of the paper is organized as follows. The next section introduces the decentralized label model as the background of our development. Section 3 describes our language with run-time principals, including its type system and the noninterference proof. Section 4 considers adding declassification in the context of run-time principals. Section 5 contains the detailed proof of type-safety for the full language. Section 6 suggests how the security policies admitted by our language can be integrated with traditional public key infrastructures and gives an extended example. The last section discusses related work and conclusions.

2. DECENTRALIZED LABEL MODEL

The security model considered in this paper is a version of the decentralized label model (DLM) developed by Myers and Liskov [1998; 2000]. However, the labels in this paper include integrity constraints in addition to confidentiality constraints, because integrity constraints allow robust declassification (see Section 4).

2.1 Principals and labels

Policies in the DLM are described in terms of a set of *principal names*. We use capitalized words like *Alice*, *Bob*, *Manager*, etc., to distinguish principal names

from other syntactic classes of the language. We use meta-variable X to range over such names.

To accommodate run-time principals, it is necessary to write policies that refer to principals whose identities are not known statically. Thus, the policy language includes *principal variables*, ranged over by α . Principal variables may be instantiated with principal names, as described below. In the example from the introduction, **Manager** is a principal name and the use of **user** in the label is a principal variable. We also need sets of principals, s , written as (unordered) comma-separated lists of principals. The empty set (of principals and other syntactic classes), written ‘ \cdot ’, will often be elided. In summary:

$$p ::= X \mid \alpha \qquad s ::= \cdot \mid p, s$$

Using principals and principal sets, the DLM builds *labels* that describe both *confidentiality* requirements, which restrict the principals that may read the data, and *integrity* requirements, which restrict which principals trust the data.

The confidentiality requirements of the DLM are composed of *reader policy components* of the form $p:s$, where p is the *owner* of the permissions and s is a set of principals permitted by p to read the data. For example, the component $Alice:Bob, Charles$ says that *Alice*’s policy is that only *Bob* and *Charles* (and implicitly *Alice*) may read data with this label. The confidentiality part of a label consists of a set of policy components, each of which must be obeyed—the principals able to read the data must be in the intersection of the reader permissions. For example, data labeled with the two reader permissions $Alice:Bob, Charles$ and $Bob:Charles, Eve$ will be readable only by *Charles* and *Bob*.¹

The information-flow type system described below ensures that data with a given confidentiality policy will only flow to destinations with labels that are at least as restrictive as the policy (see the discussion of the label lattice below). This label model is decentralized in the sense that each principal may specify reader sets independently.

The integrity part of a label consists of a set of principals that *trust* the data.² For integrity, the information-flow analysis ensures that less trusted data (trusted by fewer principals) is never used where more trusted data is necessary. For example, data whose integrity label is the set $Alice, Bob$ is trusted by *Alice* and by *Bob* but not by *Charles*.

Collecting the descriptions above, we arrive at the following formal syntax for reader policies c , confidentiality policy sets d , and labels l . The integrity part of a label is separated from the confidentiality part by ‘!’:

$$c ::= p:s \qquad d ::= \cdot \mid c;d \qquad l ::= \{d!s\}$$

2.2 Acts-for hierarchy

The decentralized label model also includes *delegation* embodied by a binary *acts-for* relation \preceq between principals. This relation is reflexive and transitive, yielding

¹Or, more precisely, principals that can act for *Charles* or *Bob*; see the discussion of the acts-for hierarchy in Section 2.2.

²It would be possible to give a version of integrity fully dual to the owners–readers model by using an owners–writers model, but there do not seem to be compelling reasons to do so [Li et al. 2003].

a preorder on principals. The notation $p \preceq q$ indicates that principal q acts for principal p , or, conversely, that p delegates to q .

The acts-for relation is taken into account when determining the restrictions imposed by a label. For example, consider the labels $\{Alice: !Alice\}$ and $\{Bob: !Bob\}$. These labels describe data readable and trusted only by *Alice* and *Bob*, respectively. However, if the relation $Alice \preceq Bob$ is in the acts-for relation, then data with label $\{Alice: !Alice\}$ will be readable by *Bob*—because *Bob* acts for *Alice*, *Bob* can read anything that *Alice* can read. *Bob* does *not* trust the integrity of data with label $\{Alice: !Alice\}$ —*Alice*'s trust in the data does not imply *Bob*'s trust. *Alice* does trust data with label $\{Bob: !Bob\}$, again because *Bob* acts for *Alice*, anything *Bob* trusts *Alice* does too.

Formally, an *acts-for hierarchy* Δ is a set of $p \preceq q$ constraints. Δ is *closed* if it contains no principal variables. To make it easier to distinguish closed acts-for hierarchies from potentially open ones, we use the notation \mathcal{A} rather than Δ to mean a closed hierarchy.

We write $\Delta \vdash p \preceq q$ if principal q acts for principal p according to hierarchy Δ , or formally, if the reflexive, transitive closure of Δ contains $p \preceq q$. The notation $\Delta \vdash s_1 \preceq s_2$ extends this delegation relation to sets of principals: the set of principals s_1 can act for the set of principals s_2 if for each principal $p \in s_1$ there exists a principal $q \in s_2$ such that $\Delta \vdash p \preceq q$. We write $\Delta \vdash s_1 \sqcap s_2$ to be the meet of s_1 and s_2 , that is, the largest s_3 such that $\Delta \vdash s_3 \preceq s_1$ and $\Delta \vdash s_3 \preceq s_2$.

Furthermore, we assume the existence of a unique most powerful principal \top (called *top*) that acts for all other principals. For all principals p and all hierarchies Δ , we have $\Delta \vdash p \preceq \top$.

2.3 Label lattice

A label l_1 is less restrictive than a label l_2 according to an acts-for hierarchy Δ , written $\Delta \vdash l_1 \sqsubseteq l_2$, when l_1 permits more readers and is at least as trusted. Formally, this relation is defined in according to these two rules (adapted from Myers and Liskov [2000] but extended to include integrity sets):

$$\frac{\forall c_1 \in d_1. \exists c_2 \in d_2. \Delta \vdash c_1 \sqsubseteq c_2 \quad \Delta \vdash s_2 \preceq s_1}{\Delta \vdash \{d_1!s_1\} \sqsubseteq \{d_2!s_2\}}$$

$$\frac{\Delta \vdash p_1 \preceq p_2 \quad \Delta \vdash s_2 \preceq s_1}{\Delta \vdash p_1:s_1 \sqsubseteq p_2:s_2}$$

We write $\Delta \vdash l_1 \not\sqsubseteq l_2$ if it is not the case that $\Delta \vdash l_1 \sqsubseteq l_2$. This negation is well defined because the problem of determining the \sqsubseteq relation is (efficiently) decidable—it reduces to a graph reachability problem over the finite acts-for hierarchy.

The labels of the DLM form a distributive, join-semi lattice, with join operation given by

$$\Delta \vdash \{d_1!s_1\} \sqcup \{d_2!s_2\} \stackrel{\text{def}}{=} \{d_1 \cup d_2!s_3\} \quad \text{if } \Delta \vdash s_1 \sqcap s_2 = s_3$$

The intuition is that the \sqsubseteq relation describes legal information flows, and the $\not\sqsubseteq$ relation describes the illegal information flows that should not be permitted in a secure program. According to these rules, the following example label inequalities

can be derived:

$$\begin{aligned}
& \cdot \vdash \{Alice:Bob!\} \sqsubseteq \{Alice:!\} \\
& \cdot \vdash \{Alice:!\} \not\sqsubseteq \{Alice:Bob!\} \\
& \cdot \vdash \{!Alice, Bob\} \sqsubseteq \{!Alice\} \\
& \cdot \vdash \{!Alice\} \not\sqsubseteq \{!Alice, Bob\} \\
Alice \preceq Bob & \vdash \{Alice:!\} \sqsubseteq \{Bob:!\} \\
Alice \preceq Bob & \vdash \{Bob:!\} \not\sqsubseteq \{Alice:!\} \\
\Delta \vdash \{!\top\} & \sqsubseteq l \quad (\text{for all } \Delta \text{ and } l) \\
\Delta \vdash l & \sqsubseteq \{\top:!\} \quad (\text{for all } \Delta \text{ and } l)
\end{aligned}$$

These inequalities show that there is a top-most label $\{\top:!\}$ (owned by \top , readable and trusted by no principals) and that the bottom of the label lattice is $\{!\top\}$ (completely unconstrained readers, trusted by all principals). Data with a less restrictive label may always be treated as having a more restrictive label.

3. RUN-TIME PRINCIPALS

This section describes the language λ_{RP} , a variant of the typed λ -calculus with information-flow policies drawn from the label lattice described above. In order to focus on run-time principals, λ_{RP} does not address several aspects of information flow. First, all programs in λ_{RP} terminate, thus it precludes termination channels. Second, λ_{RP} does not have state, so no information channels may arise through the shared memory. Third, the analysis presented here does not consider timing channels. The type system could be extended to remove all of these limitations using known techniques [Volpano et al. 1996; Agat 2000; Sabelfeld and Sands 2001; Pottier and Simonet 2002; Zdancewic and Myers 2002].

Security types, plain types, program terms and values of the language are defined according to the grammars in Figure 1. Like in previous information-flow languages, computation in λ_{RP} is described by security types (t), which are plain types (u) annotated with a label (l).

The unit, sum, and function types are standard [Pierce 2002]. There is only one value, written $*$, of type 1. Sum values are created by tagging another value v with either the left or right tag: $\mathbf{inl} \ v$ and $\mathbf{inr} \ v$, respectively. The **case** expression branches on the tag of a sum value. Function values, of type $t_1 \rightarrow t_2$ are λ -abstractions of the form $\lambda x:t. e$, where x is the formal parameter that is bound within expression e , the body of the function. Function application is written by juxtaposition of expressions.

By convention, if the label is omitted from a plain type, we take it to be the minimal label, $\{!\top\}$. For example, the type $1_{\{!\top\}}$ can be written 1. We encode Booleans with label l to be $\mathbf{bool}_l \stackrel{\text{def}}{=} (1 + 1)_l$ such that $\mathbf{true} \stackrel{\text{def}}{=} \mathbf{inl} \ *$ and $\mathbf{false} \stackrel{\text{def}}{=} \mathbf{inr} \ *$. The expression **if** (e) $e_1 \ e_2$ is encoded as **case** $e \ (\lambda x_1:1. e_1) \ (\lambda x_2:1. e_2)$, for some fresh names x_1 and x_2 .

The last two kinds of types, P_p and $\forall \alpha \preceq p. t$, are the new features related to run-time principals. The run-time representation of a principal such as *Alice* may be a public key or some other structured data, but for now we treat these representations as abstract. The only value of type P_{Alice} is the constant *Alice*. That is, P_p is a *singleton type* [Aspinall 1994]; such types have previously been used to represent other kinds of run-time type information [Crary et al. 2002]. A

$t ::= u_l$	Secure types
$u ::=$ $\mathbf{1}$ $t + t$ $t \rightarrow t$ \mathbf{P}_p $\forall \alpha \preceq p. t$ $\exists \alpha \preceq p. t$	Plain types unit sum function principal universal existential
$e ::=$ v x $\mathbf{inl} \ e$ $\mathbf{inr} \ e$ $\mathbf{case} \ e \ v \ v$ $e \ e$ $\mathbf{if} \ (e \preceq e) \ e \ e$ $e \ [p]$ $\mathbf{open} \ (\alpha, x) = e \ \mathbf{in} \ e$	Terms value variable left injection right injection sum case application if delegation instantiation opening
$v ::=$ $*$ $\mathbf{inl} \ v$ $\mathbf{inr} \ v$ $\lambda x:t. e$ X $\Lambda \alpha \preceq p. e$ $\mathbf{pack} \ (p \preceq p, v)$	Values unit left injection right injection function principal name polymorphism packing
$E ::=$ $\mathbf{inl} \ E \mid \mathbf{inr} \ E \mid \mathbf{case} \ E \ v \ v$ $\mid E \ e \mid v \ E$ $\mid E \ [p] \mid \mathbf{open} \ (\alpha, x) = E \ \mathbf{in} \ e$ $\mid \mathbf{if} \ (E \preceq e) \ e \ e \mid \mathbf{if} \ (v \preceq E) \ e \ e$	Evaluation contexts

Fig. 1. Syntax of types, terms, and values for λ_{RP}

program can perform a dynamic test of the acts-for relation between *Alice* and *Bob* using the expression $\mathbf{if} \ (Alice \preceq Bob) \ e_1 \ e_2$.

The type $\forall \alpha \preceq p. t$ is a form of *bounded quantification* [Pierce 2002] over principals.³ This type introduces a principal variable, and it describes programs for which the static information about principal α is that the acts-for relation $\alpha \preceq p$

³It might be useful in practice to add lower bounds or multiple bounds, but we do not investigate them here.

holds. For example, the type $t_0 = \forall\alpha \preceq Alice. \mathbf{bool}_{\{\alpha:!\}} \rightarrow \mathbf{bool}_{\{\alpha:!\}}$ describes functions whose parameter and return types are Booleans owned by any principal for whom *Alice* may act.

Term-level expressions generalize the principal variable α using the syntax $\Lambda\alpha \preceq p. e$. If f is such a function of the type t_0 given above, and if the acts-for hierarchy establishes that $Bob \preceq Alice$, we may call f by instantiating α with *Bob* by $f [Bob] \mathbf{true}$. A bound of \top in a polymorphic type, as in $\forall\alpha \preceq \top. t$, expresses a policy parameterized by *any* principal, because all principals satisfy the constraint $p \preceq \top$. For convenience, we define the syntactic sugar $\forall\alpha. t \stackrel{\text{def}}{=} \forall\alpha \preceq \top. t$ and $\Lambda\alpha. e \stackrel{\text{def}}{=} \Lambda\alpha \preceq \top. e$.

This kind of polymorphism over principals, in conjunction with the singleton principal types, provides a connection between the static type system and the program's run-time tests of the acts-for hierarchy. Consider the following program g , which is similar to the `printIfManager` example in Section 1:

$$\begin{aligned} g &: \forall\alpha. P_\alpha \rightarrow (\mathbf{bool}_{\{\alpha:!\}} \rightarrow 1) \rightarrow \mathbf{bool}_{\{M:!\}} \rightarrow 1 \\ g &= \Lambda\alpha. \lambda user:P_\alpha. \lambda print:\mathbf{bool}_{\{\alpha:!\}} \rightarrow 1. \\ &\quad \lambda s:\mathbf{bool}_{\{M:!\}}. \mathbf{if} (M \preceq user) (print s) * \end{aligned}$$

This function is parameterized by the principal variable α . The next parameter is a run-time principal *user* that has type P_α , meaning that the static name associated with the run-time principal *user* is α . The next two arguments to g are a function called *print*, which expects an argument owned by α , and a Boolean value s , owned by the principal M (here abbreviating *Manager*). The body of g performs a run-time test to determine whether *user* acts for M . If so, the first branch of the conditional is taken, and the *print* function is applied to the secret s . Otherwise, the unit value $*$ is returned.

Another form of quantification is existential types $\exists\alpha \preceq p. t$, which are useful for encapsulating the run-time identity of some principal.⁴ For example, the Java API `Runtime.getUser` in Section 1 can now be given the type $1 \rightarrow \exists\alpha \preceq \top. P_\alpha$, which means that the value is a package containing the identity of a principal but its static type is encapsulated and only its upper bound \top is revealed.

These existential types have been traditionally used for encoding modules or packages [Pierce 2002]. An expression `pack` ($p_1 \preceq p_2, e$) hides the principal p_1 inside e , revealing only the upper bound p_2 of the delegation to the rest of the program. Programmers can then use the new expression `open` ($\alpha, x = e_1$ in e_2) to interact with the package e_1 inside the scope of e_2 .

The main feature of existential packages is that they are first-class values which can be freely passed around. The distributed banking example in Figure 11 illustrates such practical use of existential types.

3.1 Dynamic semantics

The operational semantics of λ_{RP} , shown in Figure 2, is standard [Pierce 2002], except for the addition of the acts-for hierarchy and the if-acts-for test. We use the notation $\mathcal{A}, e \longrightarrow \mathcal{A}, e'$ to mean that an acts-for hierarchy \mathcal{A} and a program e

⁴Existential types can be encoded in terms of universal types, following Reynolds' encoding (but requires a whole program analysis). For clarity, we include existential types here as primitives.

$$\begin{array}{l}
\mathcal{A}, \text{case } (\text{inl } v) v_1 v_2 \longrightarrow \mathcal{A}, v_1 v \quad (\text{E-CaseInl}) \\
\mathcal{A}, \text{case } (\text{inr } v) v_1 v_2 \longrightarrow \mathcal{A}, v_2 v \quad (\text{E-CaseInr}) \\
\mathcal{A}, (\lambda x:t. e) v \longrightarrow \mathcal{A}, e\{v/x\} \quad (\text{E-Fun}) \\
\mathcal{A}, (\Lambda\alpha \preceq p. e) [X] \longrightarrow \mathcal{A}, e\{X/\alpha\} \quad (\text{E-All}) \\
\mathcal{A}, \text{open } (\alpha, x) = (\text{pack } (X_1 \preceq X_2, v)) \text{ in } e \longrightarrow \mathcal{A}, e\{X_1/\alpha, v/x\} \quad (\text{E-Some}) \\
\frac{\mathcal{A} \vdash X_1 \preceq X_2}{\mathcal{A}, \text{if } (X_1 \preceq X_2) e_3 e_4 \longrightarrow \mathcal{A}, e_3} \quad (\text{E-IfDelYes}) \\
\frac{\mathcal{A} \vdash X_1 \not\preceq X_2}{\mathcal{A}, \text{if } (X_1 \preceq X_2) e_3 e_4 \longrightarrow \mathcal{A}, e_4} \quad (\text{E-IfDelNo}) \\
\frac{\mathcal{A}, e_1 \longrightarrow \mathcal{A}, e_2}{\mathcal{A}, E[e_1] \longrightarrow \mathcal{A}, E[e_2]} \quad (\text{E-Context})
\end{array}$$

Fig. 2. Evaluation rules of λ_{RP}

make a small step of evaluation to become \mathcal{A} and e' . The evaluation of a program is the reflexive and transitive closure of the small-step evaluation. Note that \mathcal{A} is used but never changed here; Section 4.2 considers run-time modification of \mathcal{A} via delegation.

In Figure 2, E-Fun says that, if an abstraction $\lambda x:t. e$ is applied to a value v , then v is substituted for x in e . Similarly, by E-All, if a polymorphic term $\Lambda\alpha \preceq p. e$ is instantiated to a principal X , then X is substituted for α in e . E-Some does both the term and the type substitutions when opening up a package. We use the notation $e\{v/x\}$ and $e\{X/\alpha\}$ for capture-avoiding substitutions.

E-CaseInl and E-CaseInr are rules for conditional test of tagged values: If the test condition is a left-injection $\text{inl } v$, the first branch is applied to v . For example, using the Boolean encoding described earlier,

$$\begin{array}{l}
\text{if } (\text{true}) \text{ Alice Bob} \\
\stackrel{\text{def}}{=} \text{case } (\text{inl } *) (\lambda y:1. \text{ Alice}) (\lambda y:1. \text{ Bob}) \\
\longrightarrow (\lambda y:1. \text{ Alice}) * \\
\longrightarrow \text{Alice}
\end{array}$$

E-IfDelYes and E-IfDelNo, unlike the other rules above, use the acts-for hierarchy \mathcal{A} to check delegation at run-time. If \mathcal{A} proves that principal X_1 delegates to principal X_2 , the result of an if-acts-for term is the first branch; otherwise, the result is the second branch.

E-Context specifies the congruence rules for evaluation with contexts.

$$\begin{array}{c}
\frac{x : t \in \Gamma}{\Delta; \Gamma \vdash x : t} \quad (\text{T-Var}) \\
\\
\frac{\Delta \vdash l}{\Delta; \Gamma \vdash * : 1_l} \quad (\text{T-Unit}) \\
\\
\frac{\Delta; \Gamma \vdash e : t_1 \quad \Delta \vdash l}{\Delta; \Gamma \vdash \text{inl } e : (t_1 + t_2)_l} \quad (\text{T-Inl}) \\
\\
\frac{\Delta; \Gamma \vdash e : t_2 \quad \Delta \vdash l}{\Delta; \Gamma \vdash \text{inr } e : (t_1 + t_2)_l} \quad (\text{T-Inr}) \\
\\
\frac{\Delta; \Gamma \vdash e : (t_1 + t_2)_l \quad \Delta; \Gamma \vdash v_1 : (t_1 \rightarrow t)_l \quad \Delta; \Gamma \vdash v_2 : (t_2 \rightarrow t)_l}{\Delta; \Gamma \vdash \text{case } e \ v_1 \ v_2 : t \sqcup l} \quad (\text{T-Case}) \\
\\
\frac{\Delta; \Gamma, x : t_1 \vdash e : t_2 \quad \Delta \vdash l}{\Delta; \Gamma \vdash \lambda x : t_1. e : (t_1 \rightarrow t_2)_l} \quad (\text{T-Fun}) \\
\\
\frac{\Delta; \Gamma \vdash e_1 : (t_1 \rightarrow t_2)_l \quad \Delta; \Gamma \vdash e_2 : t_1 \quad \Delta \vdash \pi_2 \preceq (\pi_1 | l)}{\Delta; \Gamma; \pi_1 \vdash e_1 \ e_2 : t_2 \sqcup l} \quad (\text{T-App}) \\
\\
\frac{\Delta; \Gamma \vdash e_1 : (\mathbb{P}_p)_l \quad \Delta; \Gamma \vdash e_2 : (\mathbb{P}_q)_l \quad \Delta, p \preceq q; \Gamma \vdash e_3 : t \quad \Delta; \Gamma \vdash e_4 : t}{\Delta; \Gamma \vdash \text{if } (e_1 \preceq e_2) \ e_3 \ e_4 : t \sqcup l} \quad (\text{T-IfDel}) \\
\\
\frac{\Delta \vdash l}{\Delta; \Gamma \vdash X : (\mathbb{P}_X)_l} \quad (\text{T-Name}) \\
\\
\frac{\Delta, \alpha \preceq p; \Gamma \vdash e : t \quad \alpha \notin \text{ftv}(\Gamma) \quad \Delta \vdash l}{\Delta; \Gamma \vdash \Lambda \alpha \preceq p. e : (\forall \alpha \preceq p. t)_l} \quad (\text{T-All}) \\
\\
\frac{\Delta; \Gamma \vdash e : (\forall \alpha \preceq q. t)_l \quad \Delta \vdash p \preceq q}{\Delta; \Gamma \vdash e [p] : t\{p/\alpha\} \sqcup l} \quad (\text{T-Inst}) \\
\\
\frac{\Delta; \Gamma \vdash e : t\{p/\alpha\} \quad \Delta \vdash p \preceq q \quad \Delta \vdash l}{\Delta; \Gamma \vdash \text{pack } (p \preceq q, e) : (\exists \alpha \preceq q. t)_l} \quad (\text{T-Pack}) \\
\\
\frac{\Delta; \Gamma \vdash e_1 : (\exists \alpha \preceq p. t_1)_l \quad \Delta, \alpha \preceq p; \Gamma, x : t_1 \vdash e_2 : t_2 \quad \alpha \notin \text{ftv}(\Gamma) \cup \text{ftv}(t_2)}{\Delta; \Gamma \vdash \text{open } (\alpha, x) = e_1 \ \text{in } e_2 : t_2 \sqcup l} \quad (\text{T-Open}) \\
\\
\frac{\Delta; \Gamma \vdash e : t_1 \quad \Delta \vdash t_1 \leq t_2}{\Delta; \Gamma \vdash e : t_2} \quad (\text{T-Sub})
\end{array}$$

Fig. 3. Typing rules of λ_{RP}

$$\begin{array}{c}
\frac{\Delta \vdash u \leq u' \quad \Delta \vdash l \sqsubseteq l'}{\Delta \vdash u_l \leq u'_l} \quad (\text{S-Label}) \\
\Delta \vdash u \leq u \quad (\text{S-Refl}) \\
\frac{\Delta \vdash u \leq u' \quad \Delta \vdash u' \leq u''}{\Delta \vdash u \leq u''} \quad (\text{S-Trans}) \\
\frac{\Delta \vdash t_1 \leq t'_1 \quad \Delta \vdash t_2 \leq t'_2}{\Delta \vdash (t_1 + t_2) \leq (t'_1 + t'_2)} \quad (\text{S-Sum}) \\
\frac{\Delta \vdash t'_1 \leq t_1 \quad \Delta \vdash t_2 \leq t'_2}{\Delta \vdash (t_1 \rightarrow t_2) \leq (t'_1 \rightarrow t'_2)} \quad (\text{S-Fun}) \\
\frac{\Delta \vdash p' \leq p \quad \Delta, \alpha \leq p' \vdash t \leq t'}{\Delta \vdash (\forall \alpha \leq p. t) \leq (\forall \alpha \leq p'. t')} \quad (\text{S-All})
\end{array}$$

Fig. 4. Subtyping rules of λ_{RP}

3.2 Static semantics

Our type system, shown in Figure 3, is similar to those previously proposed [Heintze and Riecke 1998; Pottier and Conchon 2000; Zdancewic and Myers 2002], except for the addition of rules for run-time principals. The notation $\Delta; \Gamma \vdash e : t$ means that a program e has type t under the hierarchy Δ and the term environment Γ .

To explain how the type system keeps track of information flow, consider the typing rule T-Case for a case term. The test condition has type $(t_1 + t_2)_l$, the first branch must be a function of type $t_1 \rightarrow t$, and the second branch must be a function of type $t_2 \rightarrow t$. This typing rule matches the operational semantics of E-CaseInl and E-CaseInr mentioned above. The label of the inputs (the test condition and the branches) will be folded into the label of the output as in $t \sqcup l$. We define $t \sqcup l = (u_l) \sqcup l = u_{(l' \sqcup l)}$ so that the output always has a label as high as the input's label. For all elimination forms (T-App, T-IfDel and T-Inst), this restriction on the output label is used to rule out implicit information flows [Heintze and Riecke 1998; Zdancewic and Myers 2002].

T-Var (variables), T-Unit (units), T-Inl (left injections), T-Inr (right injections), T-Fun (functions), T-App (applications), T-Sub (subsumption) are standard rules for lambda calculus with subtyping [Pierce 2002].⁵ Figure 4 shows the subtyping rules of λ_{RP} . Note the absence of subtyping for singletons; it is unsound to combine subtyping and singletons [Aspinall 1994].

By T-Name, a principal constant X has type $(P_X)_l$. This *singleton property* ties the static type information and the run-time identity of principals—if a program expression has type $(P_X)_l$, it is guaranteed to evaluate to the constant X (because of

⁵By Barendregt's variable convention, a binding variable is chosen fresh with respect to the current context. In particular, x does not occur free in Γ and α does not occur free in Δ in rules T-Fun, T-All and T-Open.

type preservation and canonical forms). The extra condition $\Delta \vdash l$ checks that the label l is well-formed under hierarchy Δ , meaning that all free principal variables of l are contained in Δ .

T-All indicates that a polymorphic term $\Lambda\alpha \preceq p. e$ is well-typed if the body e is well-typed under hierarchy Δ extended with the additional delegation $\alpha \preceq p$. The extra condition $\alpha \notin \text{ftv}(\Gamma)$ ensures the well-formedness of the environment— α is a fresh variable. T-Inst requires the left term to be a polymorphic term and that the delegation constraint $\Delta \vdash p \preceq q$ on the instantiated principal is known statically.

T-Pack and T-Open are similar to T-All and T-Inst. The additional restriction $\alpha \notin \text{ftv}(t_2)$ prevents the escape of the lexically-scoped type variable α . This restriction is an important detail for a sound type system with existential types [Pierce 2002].

T-IfDel is similar to T-All in that it extends Δ with $\alpha \preceq p$, but it does the extension only for the first branch. This matches the operational semantics of E-IfDelYes and E-IfDelNo in Figure 2. Extending Δ for the first branch reflects the run-time information that the branch is run only when $\alpha \preceq p$ holds at run-time. For example, when type-checking the program g at the beginning of this section, the function application `print s` will be type-checked in a context where $M \preceq \alpha$. Because $M \preceq \alpha \vdash \{M:!\} \sqsubseteq \{\alpha:!\}$ the function application is permitted—inside the first branch of the if-acts-for, a value of type $\text{bool}_{\{M:!\}}$ can be treated as though it has type $\text{bool}_{\{\alpha:!\}}$.

The following shows the safety of the type system with respect to the operational semantics.

THEOREM 1 (TYPE-SAFETY).

- (1) *Progress:* If $\mathcal{A} \vdash e : t$, then $e = v$ or $\mathcal{A}, e \longrightarrow \mathcal{A}, e'$ for some e' .
- (2) *Preservation:* If $\mathcal{A} \vdash e : t$ and $\mathcal{A}, e \longrightarrow \mathcal{A}, e'$, then $\mathcal{A} \vdash e' : t$.

The proof for this theorem is similar to those for languages with subtyping [Pierce 2002]. Since type-safety also holds for our full language with declassification and authority (to be introduced in Section 4), we present the full proof altogether as Theorem 6 in Section 5. Theorem 1 here is a specialized version of Theorem 6 where the authority π is empty.

3.3 Noninterference

This section proves a noninterference theorem [Goguen and Meseguer 1982], which is the first main theoretical result of this paper. The intuition is that in secure programs, high-security inputs do not interfere with low-security outputs.

Formally, the noninterference theorem states that if a Boolean program e of low security l is well-typed and contains a free variable x of high security l' , and if values v and v' have the same type and security as x , then substituting either v or v' for x in e will evaluate to the same Boolean value v_0 . We use Boolean so that the equivalence of the final values can be observed syntactically. This result means that a low-security observer cannot use a well-typed program e to learn information about the high-security input x .

$$\begin{array}{c}
\frac{\mathcal{A}, e \longrightarrow^* \mathcal{A}, v \quad \mathcal{A}, e' \longrightarrow^* \mathcal{A}, v' \quad \mathcal{A} \vdash v \sim_{\zeta} v' : t}{\mathcal{A} \vdash e \approx_{\zeta} e' : t} \quad (\text{R-Term}) \\
\\
\frac{\mathcal{A} \vdash l \not\sqsubseteq \zeta}{\mathcal{A} \vdash v \sim_{\zeta} v' : u_l} \quad (\text{R-Label}) \\
\\
\mathcal{A} \vdash * \sim_{\zeta} * : 1_l \quad (\text{R-Unit}) \\
\\
\frac{\mathcal{A} \vdash v \sim_{\zeta} v' : t_1}{\mathcal{A} \vdash \text{inl } v \sim_{\zeta} \text{inl } v' : (t_1 + t_2)_l} \quad (\text{R-Inl}) \\
\\
\frac{\mathcal{A} \vdash v \sim_{\zeta} v' : t_2}{\mathcal{A} \vdash \text{inr } v \sim_{\zeta} \text{inr } v' : (t_1 + t_2)_l} \quad (\text{R-Inr}) \\
\\
\frac{\forall (\mathcal{A} \vdash v_2 \sim_{\zeta} v'_2 : t_1). \mathcal{A} \vdash (v \ v_2) \approx_{\zeta} (v' \ v'_2) : t_2 \sqcup l}{\mathcal{A} \vdash v \sim_{\zeta} v' : (t_1 \rightarrow t_2)_l} \quad (\text{R-Fun}) \\
\\
\mathcal{A} \vdash X \sim_{\zeta} X : (\text{P}_X)_l \quad (\text{R-Name}) \\
\\
\frac{\forall (\mathcal{A} \vdash X \preceq p). \mathcal{A} \vdash (v \ [X]) \approx_{\zeta} (v' \ [X]) : t\{X/\alpha\} \sqcup l}{\mathcal{A} \vdash v \sim_{\zeta} v' : (\forall \alpha \preceq p. t)_l} \quad (\text{R-All}) \\
\\
\frac{\mathcal{A} \vdash v \sim_{\zeta} v' : t\{X_1/\alpha\}}{\mathcal{A} \vdash \text{pack } (X_1 \preceq X_2, v) \sim_{\zeta} \text{pack } (X_1 \preceq X_2, v') : (\exists \alpha \preceq X_2. t)_l} \quad (\text{R-Some})
\end{array}$$

Fig. 5. Logical relations for types with labels

THEOREM 2 (NONINTERFERENCE). *If $\mathcal{A}; x : u_l \vdash e : \text{bool}_l$, $\mathcal{A} \vdash l' \not\sqsubseteq l$, $\mathcal{A} \vdash v : u_l'$ and $\mathcal{A} \vdash v' : u_l'$ then*

$$\mathcal{A}, e\{v/x\} \longrightarrow^* \mathcal{A}, v_0 \quad \text{iff} \quad \mathcal{A}, e\{v'/x\} \longrightarrow^* \mathcal{A}, v_0$$

The proof requires a notion of equivalence with respect to observers of different security labels. To reason about equivalence of higher-order functions and polymorphism, we use the standard technique of logical relations [Mitchell 1996]. However, we parameterize the relations with an upper-bound ζ (“zeta”) of the observer’s security label, capturing the dependence of the terms’ equivalence on the observer’s label.

Logical relations. Figure 5 shows the formal definition of the logical relation. We use the notation $\mathcal{A} \vdash e \approx_{\zeta} e' : t$ to denote two related computations, and $\mathcal{A} \vdash v \sim_{\zeta} v' : t$ to denote two related values. These relations are parameterized by a type t , an acts-for hierarchy \mathcal{A} , and an upper-bound ζ of the observer’s security label.

R-Term indicates that two terms are related at type t if they evaluate to values which are related at type t . R-Label is the crucial definition for logical relations

with labels. It relates *any* two values at type u_l as long as the label l is not lower than the observer's label ζ . If R-Label does not apply, values are related only by one of the following syntax-directed rules.

By R-Unit, $*$ is related only to itself and, similarly, by R-Name, X is related only to itself (because they are both singleton types). R-Inl says that two values are related at $(t_1 + t_2)_l$ if they both are left-injections of the form $\text{inl } v$ and $\text{inl } v'$, and if v and v' are related at t . By R-Fun, two values are related at $(t_1 \rightarrow t_2)_l$ if their applications to *all* values related at t_1 are related at $t_2 \sqcup l$. Lastly, R-All indicates that two values are related at $(\forall \alpha \preceq p. t)_l$ if their instantiations with *all* principals acting for p are related at $t \sqcup l$.

We use the notation $\mathcal{A} \vdash \gamma \approx_\zeta \gamma' : \Gamma$ to denote two related substitutions, meaning that $\text{dom}(\gamma) = \text{dom}(\gamma') = \text{dom}(\Gamma)$ and that, for all $x : t \in \Gamma$, we have $\mathcal{A}; \cdot \vdash \gamma(x) \approx_\zeta \gamma'(x) : t$. Note that a substituted value $\gamma(x)$ or $\gamma'(x)$ is always closed, as the logical relations relate only closed values to closed values, and closed expressions to closed expressions.

We use the notation $\delta \models \Delta$ to denote a type substitution δ modeling a type environment Δ , meaning that $\text{dom}(\delta) = \text{dom}(\Delta)$ and that, for all $\alpha \preceq p \in \Delta$, we have $\Delta \vdash \delta(\alpha) \preceq p$. That is, δ substitutes all free principal variables in Δ and, δ respects all delegation constraints in Δ .

Similarly, we use the notation $\mathcal{A} \vdash \gamma \models \Gamma$ to denote a term substitution γ modelling a term environment Γ under an acts-for hierarchy \mathcal{A} , meaning that $\text{dom}(\gamma) = \text{dom}(\Gamma)$ and that, for all $x : t \in \Gamma$, we have $\mathcal{A}; \Gamma \vdash \gamma(x) : t$. At last, the notation $\mathcal{A} = \delta(\Delta)$ is a point-wise extension of $\delta(t)$ such that $\text{dom}(\mathcal{A}) = \text{dom}(\Delta)$ and that, for all $\alpha \preceq p \in \Delta$, we have $\alpha \preceq \delta(p) \in \mathcal{A}$.

REMARK 3. *We do not deal with parametricity of polymorphic functions [Wadler 1989] nor the behavioral equivalence of existential packages [Pitts 1998]. That is, our model assumes that an observer can differentiate different representations of polymorphic functions or different implementations of existential packages. This assumption simplifies the equivalence relations, and is the key difference between noninterference and parametricity.*

Using these definitions, we strengthen the induction hypothesis of noninterference so that Theorem 2 (Noninterference) follows as a special case of this substitution lemma. In essence, the lemma states that substitution of related values yields related results.

LEMMA 4 (SUBSTITUTION FOR LOGICAL RELATIONS). *If*

- (1) $\Delta; \Gamma \vdash e : t$
- (2) $\delta \models \Delta$
- (3) $\mathcal{A} = \delta(\Delta)$
- (4) $\mathcal{A} \vdash \gamma \models \Gamma$
- (5) $\mathcal{A} \vdash \gamma' \models \Gamma$
- (6) $\mathcal{A} \vdash \gamma \approx_\zeta \gamma' : \delta(\Gamma)$

then

$$\mathcal{A} \vdash \gamma \delta(e) \approx_\zeta \gamma' \delta(e) : \delta(t)$$

PROOF. By induction on the typing derivations.

$$\text{—T-Var: } \frac{x : t \in \Gamma}{\Delta; \Gamma \vdash x : t}$$

By the assumption $\mathcal{A} \vdash \gamma \approx_{\zeta} \gamma' : \delta(\Gamma)$ and the definition of related substitutions.

$$\text{—T-Unit: } \frac{\Delta \vdash l}{\Delta; \Gamma \vdash * : 1_l}$$

By the definition of substitution, $\gamma\delta(e) = \gamma'\delta(e) = *$. Their evaluated values are related by R-Unit. The result then follows by R-Term.

$$\text{—T-Inl: } \frac{\Delta; \Gamma \vdash e_1 : t_1 \quad \Delta \vdash l}{\Delta; \Gamma \vdash \text{inl } e_1 : (t_1 + t_2)_l}$$

By the definition of substitution,

$$\gamma\delta(\text{inl } e_1) = \text{inl } \gamma\delta(e_1) \tag{1}$$

$$\gamma'\delta(\text{inl } e_1) = \text{inl } \gamma'\delta(e_1) \tag{2}$$

$$\delta((t_1 + t_2)_l) = (\delta(t_1) + \delta(t_2))_{\delta(l)} \tag{3}$$

By IH on e_1 , we have $\mathcal{A} \vdash \gamma\delta(e_1) \approx_{\zeta} \gamma'\delta(e_1) : \delta(t_1)$ with $\mathcal{A}, \gamma\delta(e_1) \longrightarrow^* \mathcal{A}, v$ and $\mathcal{A}, \gamma'\delta(e_1) \longrightarrow^* \mathcal{A}, v'$. By the evaluation under a context, we have $\mathcal{A}, \text{inl } \gamma\delta(e_1) \longrightarrow^* \mathcal{A}, \text{inl } v$ and $\mathcal{A}, \text{inl } \gamma'\delta(e_1) \longrightarrow^* \mathcal{A}, \text{inl } v'$. The result follows by (1)-(3), R-Inl and R-Term.

—T-Inr: symmetric to T-Inl.

$$\text{—T-Case: } \frac{\Delta; \Gamma \vdash e_0 : (t_1 + t_2)_l \quad \Delta; \Gamma \vdash v_1 : (t_1 \rightarrow t_0)_l \quad \Delta; \Gamma \vdash v_2 : (t_2 \rightarrow t_0)_l}{\Delta; \Gamma \vdash \text{case } e_0 v_1 v_2 : t_0 \sqcup l}$$

By IH on e_0 , we have $\mathcal{A} \vdash \gamma\delta(e_0) \approx_{\zeta} \gamma'\delta(e_0) : \delta((t_1 + t_2)_l)$ with $\mathcal{A}, \gamma\delta(e_0) \longrightarrow^* \mathcal{A}, v_0$ and $\mathcal{A}, \gamma'\delta(e_0) \longrightarrow^* \mathcal{A}, v'_0$. By the definition of substitution, we have $\gamma\delta(\text{case } e_0 v_1 v_2) = \text{case } \gamma\delta(e_0) \gamma\delta(v_1) \gamma\delta(v_2)$.

By the inversion of $\mathcal{A} \vdash v_0 \sim_{\zeta} v'_0 : \delta((t_1 + t_2)_l)$, there are three subcases to be considered:

- (1) R-Label with $\mathcal{A} \vdash l \not\sqsubseteq \zeta$: let $t_0 \sqcup l = u_{\nu}$. Since $\mathcal{A} \vdash l \not\sqsubseteq \zeta$, we have $\mathcal{A} \vdash l' \not\sqsubseteq \zeta$. Hence any two values of type u_{ν} are then trivially related by R-Label. The result follows by R-Term.
- (2) R-Inl with $v_0 = \text{inl } v$ and $v'_0 = \text{inl } v'$ with $\mathcal{A} \vdash v \sim_{\zeta} v' : \delta(t_1)$: by E-CaseInl,

$$\begin{aligned} \mathcal{A}, \text{case } v_0 \gamma\delta(v_1) \gamma\delta(v_2) &\longrightarrow^* \mathcal{A}, \gamma\delta(v_1) v \\ \mathcal{A}, \text{case } v'_0 \gamma'\delta(v'_1) \gamma'\delta(v'_2) &\longrightarrow^* \mathcal{A}, \gamma'\delta(v'_1) v' \end{aligned}$$

By IH, the definition of substitution and R-Fun, we have

$$\mathcal{A} \vdash (\gamma\delta(v_1) v) \approx_{\zeta} (\gamma'\delta(v'_1) v') : \delta(t_0 \sqcup l)$$

By R-Term, we have related values for the two application terms. By the evaluation under a context, we have related values for the result terms.

- (3) R-Inr with $v_0 = \text{inr } v$ and $v'_0 = \text{inr } v'$: symmetric to the previous case.

$$\frac{\Delta; \Gamma, x : t_1 \vdash e_0 : t_2 \quad \Delta \vdash l}{\text{—T-Fun: } \Delta; \Gamma \vdash \lambda x : t_1. e_0 : (t_1 \rightarrow t_2)_l}$$

By the definition of substitution,

$$\gamma\delta(\lambda x : t_1. e_0) = \lambda x : \delta(t_1). \gamma\delta(e_0) \quad (1)$$

$$\gamma'\delta(\lambda x : t_1. e_0) = \lambda x : \delta(t_1). \gamma'\delta(e_0) \quad (2)$$

$$\delta((t_1 \rightarrow t_2)_l) = (\delta(t_1) \rightarrow \delta(t_2))_{\delta(l)} \quad (3)$$

By (1)-(3), R-Fun and R-Term, it remains to show that $\forall \mathcal{A} \vdash v \sim_{\zeta} v' : \delta(t_1)$,

$$\mathcal{A} \vdash ((\lambda x : \delta(t_1). \gamma\delta(e_0)) v) \approx_{\zeta} ((\lambda x : \delta(t_1). \gamma'\delta(e_0)) v') : \delta(t_2 \sqcup l)$$

By the evaluation under a context with E-Fun,

$$\mathcal{A}, (\lambda x : \delta(t_1). \gamma\delta(e_0)) v \longrightarrow^* \mathcal{A}, \gamma\delta(e_0)\{v/x\} \quad (4)$$

$$\mathcal{A}, (\lambda x : \delta(t_1). \gamma'\delta(e_0)) v' \longrightarrow^* \mathcal{A}, \gamma'\delta(e_0)\{v'/x\} \quad (5)$$

Let $\gamma_0 = \gamma, x \mapsto v$ and $\gamma'_0 = \gamma', x \mapsto v'$ such that

$$\gamma\delta(e_0)\{v/x\} = \gamma_0\delta(e_0) \quad (6)$$

$$\gamma'\delta(e_0)\{v'/x\} = \gamma'_0\delta(e_0) \quad (7)$$

$$\mathcal{A} \vdash \gamma_0 \approx_{\zeta} \gamma'_0 : \delta(\Gamma, x : t_1) \quad (8)$$

By IH with $\mathcal{A} \vdash \gamma_0 \models \Gamma, x : t_1$ and $\mathcal{A} \vdash \gamma'_0 \models \Gamma, x : t_1$ and (8), we have $\mathcal{A} \vdash \gamma_0\delta(e_0) \approx_{\zeta} \gamma'_0\delta(e_0) : \delta(t_2 \sqcup l)$. Then, the result follows by (4)-(7) and R-Term.

$$\frac{\Delta; \Gamma \vdash e_1 : (t_1 \rightarrow t_2)_l \quad \Delta; \Gamma \vdash e_2 : t_1}{\text{—T-App: } \Delta; \Gamma \vdash e_1 e_2 : t_2 \sqcup l}$$

By IH on e_1 and e_2 ,

$$(1) \mathcal{A} \vdash \gamma\delta(e_1) \approx_{\zeta} \gamma'\delta(e_1) : \delta((t_1 \rightarrow t_2)_l) \text{ with } \mathcal{A}, \gamma\delta(e_1) \longrightarrow^* \mathcal{A}, v_1 \text{ and } \mathcal{A}, \gamma'\delta(e_1) \longrightarrow^* \mathcal{A}, v'_1$$

$$(2) \mathcal{A} \vdash \gamma\delta(e_2) \approx_{\zeta} \gamma'\delta(e_2) : \delta(t_1) \text{ with } \mathcal{A}, \gamma\delta(e_2) \longrightarrow^* \mathcal{A}, v_2 \text{ and } \mathcal{A}, \gamma'\delta(e_2) \longrightarrow^* \mathcal{A}, v'_2$$

The result then follows by R-Fun and R-Term.

$$\frac{\Delta \vdash l}{\text{—T-Name: } \Delta; \Gamma \vdash X : (\mathbb{P}_X)_l}$$

By the definition of substitution, $\gamma\delta(e) = \gamma'\delta(e) = X$ and $\delta((\mathbb{P}_X)_l) = (\mathbb{P}_X)_{\delta(l)}$.

The result then follows by R-Name and R-Term.

$$\frac{\Delta; \Gamma \vdash e_1 : (\mathbb{P}_{p_1})_l \quad \Delta; \Gamma \vdash e_2 : (\mathbb{P}_{p_2})_l \quad \Delta, p_1 \preceq p_2; \Gamma \vdash e_3 : t_0 \quad \Delta; \Gamma \vdash e_4 : t_0}{\text{—T-IfDel: } \Delta; \Gamma \vdash \text{if } (e_1 \preceq e_2) e_3 e_4 : t_0 \sqcup l}$$

By IH on e_1, e_2, e_3 and e_4 ,

$$(1) \mathcal{A} \vdash \gamma\delta(e_1) \approx_{\zeta} \gamma'\delta(e_1) : \delta((\mathbb{P}_{p_1})_l) \text{ with } \mathcal{A}, \gamma\delta(e_1) \longrightarrow^* \mathcal{A}, v_1 \text{ and } \mathcal{A}, \gamma'\delta(e_1) \longrightarrow^* \mathcal{A}, v'_1$$

$$(2) \mathcal{A} \vdash \gamma\delta(e_2) \approx_{\zeta} \gamma'\delta(e_2) : \delta((\mathbb{P}_{p_2})_l) \text{ with } \mathcal{A}, \gamma\delta(e_2) \longrightarrow^* \mathcal{A}, v_2 \text{ and } \mathcal{A}, \gamma'\delta(e_2) \longrightarrow^* \mathcal{A}, v'_2$$

- (3) $\mathcal{A} \vdash \gamma\delta(e_3) \approx_\zeta \gamma'\delta(e_3) : \delta(t_0)$ with $\mathcal{A}, \gamma\delta(e_3) \longrightarrow^* \mathcal{A}, v_3$ and $\mathcal{A}, \gamma'\delta(e_3) \longrightarrow^* \mathcal{A}, v'_3$
- (4) $\mathcal{A} \vdash \gamma\delta(e_4) \approx_\zeta \gamma'\delta(e_4) : \delta(t_0)$ with $\mathcal{A}, \gamma\delta(e_4) \longrightarrow^* \mathcal{A}, v_4$ and $\mathcal{A}, \gamma'\delta(e_4) \longrightarrow^* \mathcal{A}, v'_4$

By the definition of substitution and the evaluation under a context,

$$\begin{aligned} \gamma\delta(\text{if } (e_1 \preceq e_2) e_3 e_4) &= \text{if } (\gamma\delta(e_1) \preceq \gamma\delta(e_2)) \gamma\delta(e_3) \gamma\delta(e_4) \\ \gamma'\delta(\text{if } (e_1 \preceq e_2) e_3 e_4) &= \text{if } (\gamma'\delta(e_1) \preceq \gamma'\delta(e_2)) \gamma'\delta(e_3) \gamma'\delta(e_4) \\ \mathcal{A}, \text{if } (\gamma\delta(e_1) \preceq \gamma\delta(e_2)) \gamma\delta(e_3) \gamma\delta(e_4) &\longrightarrow^* \mathcal{A}, \text{if } (v_1 \preceq v_2) \gamma\delta(e_3) \gamma\delta(e_4) \\ \mathcal{A}, \text{if } (\gamma'\delta(e_1) \preceq \gamma'\delta(e_2)) \gamma'\delta(e_3) \gamma'\delta(e_4) &\longrightarrow^* \mathcal{A}, \text{if } (v'_1 \preceq v'_2) \gamma'\delta(e_3) \gamma'\delta(e_4) \end{aligned}$$

By the inversion of $\mathcal{A} \vdash v_1 \sim_\zeta v'_1 : \delta((\mathbb{P}_{p_1})_l)$ and $\mathcal{A} \vdash v_2 \sim_\zeta v'_2 : \delta((\mathbb{P}_{p_2})_l)$,

- (1) R-Label with $\mathcal{A} \vdash l \not\sqsubseteq \zeta$: by R-Label, we have related values for the result terms.
- (2) R-Name with $v_1 = v'_1 = \delta(p_1)$, $v_2 = v'_2 = \delta(p_2)$: if $\mathcal{A} \vdash \delta(p_1) \preceq \delta(p_2)$, then by the evaluation under a context with E-IfDelYes,

$$\begin{aligned} \mathcal{A}, \text{if } (v_1 \preceq v_2) \gamma\delta(e_3) \gamma\delta(e_4) &\longrightarrow^* \mathcal{A}, \gamma\delta(e_3) \\ \mathcal{A}, \text{if } (v'_1 \preceq v'_2) \gamma'\delta(e_3) \gamma'\delta(e_4) &\longrightarrow^* \mathcal{A}, \gamma'\delta(e_3) \end{aligned}$$

Otherwise, if $\mathcal{A} \vdash \delta(p_1) \not\preceq \delta(p_2)$, then by the evaluation under a context and E-IfDelNo,

$$\begin{aligned} \mathcal{A}, \text{if } (v_1 \preceq v_2) \gamma\delta(e_3) \gamma\delta(e_4) &\longrightarrow^* \mathcal{A}, \gamma\delta(e_4) \\ \mathcal{A}, \text{if } (v'_1 \preceq v'_2) \gamma'\delta(e_3) \gamma'\delta(e_4) &\longrightarrow^* \mathcal{A}, \gamma'\delta(e_4) \end{aligned}$$

In both cases, by the evaluation under a context, we have related values for the result terms at type $\delta(t_0)$. By Lemma 5 (Subtyping for logical relations) and Lemma 12 (substitution for subtyping), they are also related at type $\delta(t_0 \sqcup l)$. The result then follows by R-Term.

$$\text{—T-All: } \frac{\Delta, \alpha \preceq p; \Gamma \vdash e_0 : t_0 \quad \alpha \notin \text{ftv}(\Gamma) \quad \Delta \vdash l}{\Delta; \Gamma \vdash \Lambda\alpha \preceq p. e_0 : (\forall\alpha \preceq p. t_0)_l}$$

By the definition of substitution,

$$\gamma\delta(\Lambda\alpha \preceq p. e_0) = \Lambda\alpha \preceq \delta(p). \gamma\delta(e_0) \quad (1)$$

$$\gamma'\delta(\Lambda\alpha \preceq p. e_0) = \Lambda\alpha \preceq \delta(p). \gamma'\delta(e_0) \quad (2)$$

$$\delta((\forall\alpha \preceq p. t_0)_l) = (\forall\alpha \preceq p. \delta(t_0))_{\delta(l)} \quad (3)$$

By (1)-(3), R-All and R-Term, it remains to show that $\forall\mathcal{A} \vdash X \preceq \delta(p)$,

$$\mathcal{A} \vdash ((\Lambda\alpha \preceq \delta(p). \gamma\delta(e_0)) [X]) \approx_\zeta ((\Lambda\alpha \preceq \delta(p). \gamma'\delta(e_0)) [X]) : \delta(t_0 \sqcup l)$$

By the evaluation under a context with E-All,

$$\mathcal{A}, (\Lambda\alpha \preceq \delta(p). \gamma\delta(e_0)) [X] \longrightarrow^* \mathcal{A}, \gamma\delta(e_0)\{X/\alpha\} \quad (4)$$

$$\mathcal{A}, (\Lambda\alpha \preceq \delta(p). \gamma'\delta(e_0)) [X] \longrightarrow^* \mathcal{A}, \gamma'\delta(e_0)\{X/\alpha\} \quad (5)$$

Let $\delta_0 = \delta, \alpha \mapsto X$ such that

$$\gamma\delta(e_0)\{X/\alpha\} = \gamma\delta_0(e_0) \quad (6)$$

$$\gamma'\delta(e_0)\{X/\alpha\} = \gamma'\delta_0(e_0) \quad (7)$$

$$\delta_0 \models \Delta, \alpha \preceq p \quad (8)$$

By IH with (8), we have $\mathcal{A} \vdash \gamma\delta(e_0) \approx_\zeta \gamma'\delta(e_0) : \delta(t_0 \sqcup l)$. Then, the result follows by (4)-(7) and R-Term.

$$\frac{\Delta; \Gamma \vdash e_0 : (\forall \alpha \preceq p. t_0)_l}{\Delta; \Gamma \vdash e_0 [p] : t_0 \sqcup l}$$

—T-Inst: $\Delta; \Gamma \vdash e_0 [p] : t_0 \sqcup l$

By IH on e_0 , we have $\mathcal{A} \vdash \gamma\delta(e_0) \approx_\zeta \gamma'\delta(e_0) : \delta((\forall \alpha \preceq p. t_0)_l)$ with $\mathcal{A}, \gamma\delta(e_0) \longrightarrow^* \mathcal{A}, v$ and $\mathcal{A}, \gamma'\delta(e_0) \longrightarrow^* \mathcal{A}, v'$. The result then follows by R-All and R-Term.

$$\frac{\Delta; \Gamma \vdash e_0 : t\{q/\alpha\} \quad \Delta \vdash p \preceq q \quad \Delta \vdash l}{\Delta; \Gamma \vdash \text{pack}(p \preceq q, e_0) : (\exists \alpha \preceq q. t)_l}$$

—T-Pack: $\Delta; \Gamma \vdash \text{pack}(p \preceq q, e_0) : (\exists \alpha \preceq q. t)_l$

By the definition of substitution,

$$\gamma\delta(\text{pack}(p \preceq q, e_0)) = \text{pack}(\delta(p) \preceq \delta(q), \gamma\delta(e_0)) \quad (1)$$

$$\gamma'\delta(\text{pack}(p \preceq q, e_0)) = \text{pack}(\delta'(p) \preceq \delta'(q), \gamma'\delta(e_0)) \quad (2)$$

$$\delta((\exists \alpha \preceq q. t)_l) = (\exists \alpha \preceq \delta(q). \delta(t))_{\delta(l)} \quad (3)$$

By IH on e_0 , we have $\mathcal{A} \vdash \gamma\delta(e_0) \approx_\zeta \gamma'\delta(e_0) : \delta(t\{q/\alpha\})$ with $\mathcal{A}, \gamma\delta(e_0) \longrightarrow^* \mathcal{A}, v$ and $\mathcal{A}, \gamma'\delta(e_0) \longrightarrow^* \mathcal{A}, v'$. By the evaluation under a context,

$$\text{pack}(\delta(p) \preceq \delta(q), \gamma\delta(e_0)) \longrightarrow^* \text{pack}(\delta(p) \preceq \delta(q), v)$$

$$\text{pack}(\delta'(p) \preceq \delta'(q), \gamma'\delta(e_0)) \longrightarrow^* \text{pack}(\delta'(p) \preceq \delta'(q), v')$$

The result follows by (1)-(3), R-Some and R-Term.

$$\frac{\Delta; \Gamma \vdash e_1 : (\exists \alpha \preceq p. t_1)_l \quad \Delta, \alpha \preceq p; \Gamma, x : t_1 \vdash e_2 : t_2 \quad \alpha \notin \text{ftv}(\Gamma) \cup \text{ftv}(t_2)}{\Delta; \Gamma \vdash \text{open}(\alpha, x) = e_1 \text{ in } e_2 : t_2 \sqcup l}$$

—T-Open:

$$\Delta; \Gamma \vdash \text{open}(\alpha, x) = e_1 \text{ in } e_2 : t_2 \sqcup l$$

By IH on e_1 , $\mathcal{A} \vdash \gamma\delta(e_1) \approx_\zeta \gamma'\delta(e_1) : \delta((\exists \alpha \preceq p. t_1)_l)$ with $\mathcal{A}, \gamma\delta(e_1) \longrightarrow^* \mathcal{A}, v_1$ and $\mathcal{A}, \gamma'\delta(e_1) \longrightarrow^* \mathcal{A}, v'_1$.

By the definition of substitution and the evaluation under a context,

$$\gamma\delta(\text{open}(\alpha, x) = e_1 \text{ in } e_2) = \text{open}(\alpha, x) = \gamma\delta(e_1) \text{ in } \gamma\delta(e_2)$$

$$\gamma'\delta(\text{open}(\alpha, x) = e_1 \text{ in } e_2) = \text{open}(\alpha, x) = \gamma'\delta(e_1) \text{ in } \gamma'\delta(e_2)$$

$$\mathcal{A}, \text{open}(\alpha, x) = \gamma\delta(e_1) \text{ in } \gamma\delta(e_2) \longrightarrow^* \mathcal{A}, \text{open}(\alpha, x) = v_1 \text{ in } \gamma\delta(e_2)$$

$$\mathcal{A}, \text{open}(\alpha, x) = \gamma'\delta(e_1) \text{ in } \gamma'\delta(e_2) \longrightarrow^* \mathcal{A}, \text{open}(\alpha, x) = v'_1 \text{ in } \gamma'\delta(e_2)$$

By the inversion of $\mathcal{A} \vdash v_1 \sim_\zeta v'_1 : \delta((\exists \alpha \preceq p. t_1)_l)$,

(1) R-Label with $\mathcal{A} \vdash l \not\sqsubseteq \zeta$: by R-Label and R-Term.

(2) R-Some with $v_1 = \text{pack}(X \preceq p, v)$ and $v'_1 = \text{pack}(X \preceq p, v')$ with $\mathcal{A} \vdash v \sim_\zeta v' : \delta(t\{X/\alpha\})$: by the evaluation under a context,

$$\mathcal{A}, \text{open}(\alpha, x) = v_1 \text{ in } \gamma\delta(e_2) \longrightarrow^* \mathcal{A}, \gamma\delta(e_2)\{X/\alpha, v/x\}$$

$$\mathcal{A}, \text{open}(\alpha, x) = v'_1 \text{ in } \gamma'\delta(e_2) \longrightarrow^* \mathcal{A}, \gamma'\delta(e_2)\{X/\alpha, v'/x\}$$

We can then finish in a similar way as T-Fun and T-All by extending $\gamma_0 = \gamma, x \mapsto v$ and $\gamma'_0 = \gamma', x \mapsto v'$ and $\delta_0 = \delta, \alpha \mapsto X$.

—T-Sub:
$$\frac{\Delta; \Gamma \vdash e_0 : t_1 \quad \Delta \vdash t_1 \leq t_2}{\Delta; \Gamma \vdash e_0 : t_2}$$
 By IH on e_0 , we have $\mathcal{A} \vdash \gamma\delta(e_0) \approx_\zeta \gamma'\delta(e_0) : \delta(t_1)$ with $\mathcal{A}, \gamma\delta(e_0) \longrightarrow^* \mathcal{A}, v$ and $\mathcal{A}, \gamma'\delta(e_0) \longrightarrow^* \mathcal{A}, v'$. The result is then a consequence of the following lemma and R-Term.

LEMMA 5 (SUBTYPING FOR LOGICAL RELATIONS). *If $\mathcal{A} \vdash v \sim_\zeta v' : t$ and $\mathcal{A} \vdash t \leq t'$, then $\mathcal{A} \vdash v \sim_\zeta v' : t'$.*

□

4. DECLASSIFICATION AND AUTHORITY

Although noninterference is useful as an idealized security policy, in practice most programs *do* intentionally release some confidential information. This section considers the interaction between run-time principals and declassification and suggests *run-time authority* as a practical approach to delimiting the effects of downgrading.

The basic idea of declassification is to add an explicit method for the programmer to allow information flows downward in the security lattice. The expression `declassify e t` indicates that e should be considered to have type t , which may relax some of the labels constraining e . Declassification is like a type-cast operation; operationally it has no run-time effect:

$$\mathcal{A}, \text{declassify } v \ t \longrightarrow \mathcal{A}, v \quad (\text{E-Dcls})$$

One key issue is how to constrain its use so that the declassification correctly implements a desired security policy. Ideally, each declassification would be accompanied by formal justification of why its use does not permit unwanted downward information flows. However, such a general approach reduces to proving that a program satisfies an arbitrary policy, which is undecidable for realistic programs.

An alternative is to give up on general-purpose declassification and instead build it into appropriate operations, such as encryption. Doing so essentially limits the security policies that can be expressed, which may be acceptable in some situations, but is not desirable for general-purpose information-flow type systems.

To resolve these tensions, the original decentralized label model proposed the use of *authority* to scope the use of declassification. Intuitively, if *Alice* is an owner of the data, then her authority is needed to relax the restrictions on its use. For example, to declassify data labeled $\{Alice:!\}$ to permit *Bob* as a reader (i.e. relax the label to $\{Alice:Bob!\}$) requires *Alice*'s permission. In the original DLM, a principal's authority is statically granted to a piece of code.

Zdancewic and Myers proposed a refinement of the DLM authority model called *robust declassification* [Zdancewic and Myers 2001; Zdancewic 2003; Myers et al. 2004]. Intuitively, robust declassification requires that the *decision* to release the confidential data be trusted by the principals whose policies are relaxed. In a programming language setting, robustness entails an *integrity* constraint on the program-counter (*pc*) label—the *pc* label is a security label associated with each program point; it approximates the information that may be learned by observing that the program execution has reached the program point.

For example, suppose that the variable x has type bool_l . Then the *pc* label at the program points at the start of the branches v_0 and v_1 of the conditional

expression `case` x v_0 v_1 satisfies $l \sqsubseteq pc$ because the branch taken depends on x —observing that the program counter has reached v_0 reveals that x is `true`. If x has low integrity and is untrusted by *Alice*, then the condition $l \sqsubseteq pc$ implies that the integrity of the pc labels in the branches are also untrusted by *Alice*. Robustness requires that *Alice* trusts the pc at the point of her declassification; even if she has granted her authority to this program, no declassification affecting her policies will be permitted to take place in v_0 or v_1 .

In the presence of run-time principals, however, the story is not so straightforward. To adopt the authority model, we must find a way to represent a run-time principal’s authority. Similarly, to enforce robust declassification, we must ensure that at runtime the integrity of the program counter is trusted by any run-time principals whose data is declassified. At the same time, we would like to ensure backward compatibility with the static notions of authority and robustness in previous work [Zdancewic and Myers 2001; Zdancewic 2003].

4.1 Run-time authority and capabilities

To address downgrading with run-time principals, we use *capabilities* (unforgeable tokens) to represent the run-time authority of a principal. The meta-variable i ranges over a set of *privilege identifiers* \mathcal{I} . We are interested in controlling the use of declassification, so we assume that \mathcal{I} contains at least the identifier `declassify`, but the framework is general enough to control arbitrary privileges. In Section 4.2, we consider using capabilities to regulate other privileged operations, such as endorsement and delegation.

Figure 6 to Figure 9 summarize the changes to the language needed to support run-time authority. Just as we separate the static principal names from their run-time representation, we separate the static authority granted by a principal from its representation. The former, static authority, is written $p \triangleright i$ to indicate that principal p grants permission for the program to use privilege i . For example, a program needs to have the authority $Alice \triangleright \text{declassify}$ to declassify on *Alice*’s behalf. The latter, run-time authority, is written $X\{i\}$ and represents an unforgeable capability created by principal X and authorizing privilege i . T-Cap in Figure 8 says that capabilities have static type \mathbf{C} .

A program can test a capability at run time to determine whether a principal has granted it privilege i using the expression `if` $(e_1 \Rightarrow e_2 \triangleright i)$ e_3 e_4 . Here, e_1 evaluates to a capability and e_2 evaluates to a run-time principal; if the capability implies that the principal permits i the first branch e_3 is taken, otherwise e_4 is taken.

Other expressions (`endorse` e p and `acquire` $e \triangleright i$) and other privileges (`endorse` and `delegate` _{$p \leq p$}) will be explained in the next subsections.

To retain the benefits of robust declassification, we generalize the pc label to be a set of static permissions, π . The function type constructor must also be extended to indicate a bound on the calling context’s pc . In our setting, the bound is the minimum authority needed to invoke the function. We write such types as $[\pi] t_1 \rightarrow t_2$. For example, if f has type $[Alice \triangleright \text{declassify}] \text{bool}_{\{Alice:!\}} \rightarrow \text{bool}_{\{\top\}}$ then the caller of f must have *Alice*’s authority to declassify— f may internally do some declassification of data owned by *Alice*. Therefore f , which takes data owned by *Alice* and returns public data, may reveal information about its argument. On the other hand, a function of type $[Alice \triangleright \text{declassify}] \text{bool}_{\{Bob:!\}} \rightarrow \text{bool}_{\{\top\}}$

$u ::= \dots$ $\quad [\pi] t \rightarrow t$ $\quad \mathbf{C}$	Plain types function capability
$\pi ::= \cdot \mid \pi, p \triangleright i$	Authority
$i ::=$ $\quad \mathbf{declassify}$ $\quad \mathbf{endorse}$ $\quad \mathbf{delegate}_{p \preceq p}$	Privileges declassification endorsement delegation
$e ::= \dots$ $\quad \mathbf{if} (e \Rightarrow e \triangleright i) e e$ $\quad \mathbf{declassify} e t$ $\quad \mathbf{endorse} e p$ $\quad \mathbf{let} (e \preceq e) \mathbf{in} e$ $\quad \mathbf{acquire} e \triangleright i$	Terms if certify declassify endorse let delegate acquire
$v ::= \dots$ $\quad \mathbf{let} (X_1 \preceq X_2) \mathbf{in} \lambda x:t. e$ $\quad \mathbf{let} (X_1 \preceq X_2) \mathbf{in} \Lambda \alpha \preceq p. e$ $\quad \mathbf{let} (X_1 \preceq X_2) \mathbf{in} \mathbf{pack} (p \preceq p, v)$ $\quad X\{i\}$	Values delegated function delegated polymorphism delegated package capability
$E ::= \dots$ $\quad \mid \mathbf{let} (E \preceq e) e$ $\quad \mid \mathbf{let} (v \preceq E) e$ $\quad \mid \mathbf{if} (E \Rightarrow e \triangleright i) e e$ $\quad \mid \mathbf{if} (v \Rightarrow E \triangleright i) e e$ $\quad \mid \mathbf{declassify} E t$ $\quad \mid \mathbf{endorse} E p$ $\quad \mid \mathbf{acquire} E \triangleright i$	Evaluation contexts

Fig. 6. λ_{RP} with run-time authority

cannot declassify the argument, which is owned by *Bob*, unless *Alice* acts for *Bob*. Note that the types accurately describe the security-relevant operations that may be performed by the function.

The examples above use only static authority. To illustrate how run-time capabilities are used, consider this program:

$$\begin{aligned}
 h &: \forall \alpha. [\cdot] \mathbf{P}_\alpha \rightarrow [\cdot] \mathbf{C} \rightarrow [\cdot] \mathbf{bool}_{\{\alpha, !\}} \rightarrow \mathbf{bool}_{\{!\top\}} \\
 h &= \Lambda \alpha. \lambda user:\mathbf{P}_\alpha. \lambda cap:\mathbf{C}. \lambda data:\mathbf{bool}_{\{\alpha, !\}}. \\
 &\quad \mathbf{if} (cap \Rightarrow user \triangleright \mathbf{declassify}) \\
 &\quad \quad (\mathbf{declassify} data \mathbf{bool}_{\{!\top\}}) \\
 &\quad \mathbf{false}
 \end{aligned}$$

$$\begin{array}{c}
\frac{x : t \in \Gamma}{\Delta; \Gamma; \pi \vdash x : t} \quad (\text{T-Var}) \\
\\
\frac{\Delta \vdash l}{\Delta; \Gamma; \pi \vdash * : 1_l} \quad (\text{T-Unit}) \\
\\
\frac{\Delta; \Gamma; \pi \vdash e : t_1 \quad \Delta \vdash l}{\Delta; \Gamma; \pi \vdash \mathbf{inl} e : (t_1 + t_2)_l} \quad (\text{T-Inl}) \\
\\
\frac{\Delta; \Gamma; \pi \vdash e : t_2 \quad \Delta \vdash l}{\Delta; \Gamma; \pi \vdash \mathbf{inr} e : (t_1 + t_2)_l} \quad (\text{T-Inr}) \\
\\
\frac{\Delta; \Gamma; \pi_1 \vdash e : (t_1 + t_2)_l \quad \Delta; \Gamma; \pi_1 | l \vdash v_1 : ([\pi_2] t_1 \rightarrow t)_l \quad \Delta \vdash \pi_2 \preceq (\pi_1 | l) \quad \Delta; \Gamma; \pi_1 | l \vdash v_2 : ([\pi_2] t_2 \rightarrow t)_l}{\Delta; \Gamma; \pi_1 \vdash \mathbf{case} e v_1 v_2 : t \sqcup l} \quad (\text{T-Case}) \\
\\
\frac{\Delta; \Gamma, x : t_1; \pi \vdash e : t_2 \quad \Delta \vdash l}{\Delta; \Gamma; \cdot \vdash \lambda x : t_1. e : ([\pi] t_1 \rightarrow t_2)_l} \quad (\text{T-Fun}) \\
\\
\frac{\Delta; \Gamma; \pi_1 \vdash e_1 : ([\pi_2] t_1 \rightarrow t_2)_l \quad \Delta; \Gamma; \pi_1 \vdash e_2 : t_1 \quad \Delta \vdash \pi_2 \preceq (\pi_1 | l)}{\Delta; \Gamma; \pi_1 \vdash e_1 e_2 : t_2 \sqcup l} \quad (\text{T-App}) \\
\\
\frac{\Delta \vdash l}{\Delta; \Gamma; \pi \vdash X : (\mathbf{P}_X)_l} \quad (\text{T-Name}) \\
\\
\frac{\Delta, \alpha \preceq p; \Gamma; \pi \vdash e : t \quad \alpha \notin \text{dom}(\Delta) \quad \Delta \vdash l}{\Delta; \Gamma; \pi \vdash \Lambda \alpha \preceq p. e : (\forall \alpha \preceq p. t)_l} \quad (\text{T-All}) \\
\\
\frac{\Delta; \Gamma; \pi \vdash e : (\forall \alpha \preceq p_2. t)_l \quad \Delta \vdash p_1 \preceq p_2}{\Delta; \Gamma; \pi \vdash e [p_1] : t \sqcup l} \quad (\text{T-Inst}) \\
\\
\frac{\Delta; \Gamma; \pi \vdash e : t\{p/\alpha\} \quad \Delta \vdash p \preceq q \quad \Delta \vdash l}{\Delta; \Gamma; \pi \vdash \mathbf{pack} (p \preceq q, e) : (\exists \alpha \preceq q. t)_l} \quad (\text{T-Pack}) \\
\\
\frac{\Delta; \Gamma; \pi \vdash e_1 : (\exists \alpha \preceq p. t_1)_l \quad \Delta, \alpha \preceq p; \Gamma, x : t_1; \pi \vdash e_2 : t_2 \quad \alpha \notin \text{ftv}(\Gamma) \cup \text{ftv}(t_2)}{\Delta; \Gamma; \pi \vdash \mathbf{open} (\alpha, x) = e_1 \mathbf{in} e_2 : t_2 \sqcup l} \quad (\text{T-Open}) \\
\\
\frac{\Delta; \Gamma; \pi \vdash e : t_1 \quad \Delta \vdash t_1 \leq t_2}{\Delta; \Gamma; \pi \vdash e : t_2} \quad (\text{T-Sub})
\end{array}$$

Fig. 7. Modified typing rules of λ_{RP} with run-time authority

$$\begin{array}{c}
\frac{\Delta \vdash l}{\Delta; \Gamma; \pi \vdash X\{i\} : \mathbb{C}_l} \quad (\text{T-Cap}) \\
\\
\frac{\Delta; \Gamma; \pi \vdash e_1 : (\mathbb{P}_{p_1})_l \quad \Delta; \Gamma; \pi \vdash e_2 : (\mathbb{P}_{p_2})_l \quad \Delta, p_1 \preceq p_2; \Gamma; \pi \vdash e_3 : t \quad \Delta; \Gamma; \pi \vdash e_4 : t}{\Delta; \Gamma; \pi \vdash \text{if } (e_1 \preceq e_2) e_3 e_4 : t \sqcup l} \quad (\text{T-IfDel}) \\
\\
\frac{\Delta; \Gamma; \pi \vdash e_1 : (\mathbb{P}_{p_1})_l \quad \Delta; \Gamma; \pi \vdash e_2 : (\mathbb{P}_{p_2})_l \quad \Delta, p_1 \preceq p_2; \Gamma; \pi \vdash e_3 : t \quad \Delta \vdash p_1 \preceq \pi(\text{delegate}_{p_1 \preceq p_2})}{\Delta; \Gamma; \pi \vdash \text{let } (e_1 \preceq e_2) \text{ in } e_3 : t \sqcup l} \quad (\text{T-LetDel}) \\
\\
\frac{\Delta; \Gamma; \pi \vdash e_1 : \mathbb{C}_l \quad \Delta; \Gamma; \pi \vdash e_2 : (\mathbb{P}_p)_l \quad \Delta; \Gamma; (\pi, p \triangleright i) | l \vdash e_3 : t \quad \Delta; \Gamma; \pi | l \vdash e_4 : t}{\Delta; \Gamma; \pi \vdash \text{if } (e_1 \Rightarrow e_2 \triangleright i) e_3 e_4 : t \sqcup l} \quad (\text{T-IfCert}) \\
\\
\frac{\Delta; \Gamma; \pi \vdash e : t_2 \quad \Delta \vdash t_2 - t_1 = s \quad \Delta \vdash s \preceq \pi(\text{declassify})}{\Delta; \Gamma; \pi \vdash \text{declassify } e t_1 : t_1} \quad (\text{T-Dcls}) \\
\\
\frac{\Delta; \Gamma; \pi \vdash e : t \quad \Delta \vdash p \preceq \pi(\text{endorse})}{\Delta; \Gamma; \pi \vdash \text{endorse } e p : t \sqcap \{!p\}} \quad (\text{T-Endr}) \\
\\
\frac{\Delta; \Gamma; \pi \vdash e : (\mathbb{P}_p)_l}{\Delta; \Gamma; \pi \vdash \text{acquire } e \triangleright i : (\mathbb{C}_l + 1_l)_l} \quad (\text{T-Acq})
\end{array}$$

Fig. 8. Additional typing rules of λ_{RP} with run-time authority

The type of h is parameterized by a principal α , and the authority constraint $[\cdot]$ indicates that no static authority is needed to call this function. Instead, h takes a run-time principal $user$ (whose static name is α), a capability cap , and some data private to α . The body of the function tests whether capability cap provides evidence that $user$ has granted the program the **declassify** privilege. If so, the first branch is taken and the data is declassified to the bottom label. Otherwise h simply returns **false**.

The program h illustrates the use of the **declassify** $e t$ expression, which declassifies the expression e of type t' to have type t , where t' and t differ only in their security label annotations. The judgment $\Delta \vdash t_1 - t_2 = s$ indicates that under the acts-for hierarchy Δ , the type t_1 may be declassified to type t_2 using the authority of the principals in s . We call s the set of declassification requisites (Figure 10), which is computed by traversing through the types and collecting the authority in labels:

$$\frac{s' = \{p \mid \Delta \vdash d_2(p) \preceq d_1(p), \Delta \vdash d_1(p) \not\preceq d_2(p)\}}{\Delta \vdash \{d_1!s\} - \{d_2!s\} = s'}$$

For example, $\vdash \text{bool}_{\{Alice;! \}} - \text{bool}_{\{Alice;Bob!\}} = \{Alice\}$, because $Alice$'s authority is needed to add Bob as a reader. T-Dcls in Figure 8 is used when typechecking the **declassify** expression.

$$\begin{array}{c}
\frac{(\mathcal{A}, X_1 \preceq X_2), e_3 \longrightarrow (\mathcal{A}, X_1 \preceq X_2), e'_3}{\mathcal{A}, \text{let } (X_1 \preceq X_2) \text{ in } e_3 \longrightarrow \mathcal{A}, \text{let } (X_1 \preceq X_2) \text{ in } e'_3} \quad (\text{E-LetDel}) \\
\mathcal{A}, \text{let } X_1 \preceq X_2 \text{ in } * \longrightarrow \mathcal{A}, * \quad (\text{E-LetUnit}) \\
\mathcal{A}, \text{let } X_1 \preceq X_2 \text{ in inl } v \longrightarrow \mathcal{A}, \text{inl } v \quad (\text{E-LetInl}) \\
\mathcal{A}, \text{let } X_1 \preceq X_2 \text{ in inr } v \longrightarrow \mathcal{A}, \text{inr } v \quad (\text{E-LetInr}) \\
\mathcal{A}, \text{let } X_1 \preceq X_2 \text{ in } X \longrightarrow \mathcal{A}, X \quad (\text{E-LetName}) \\
\mathcal{A}, \text{let } X_1 \preceq X_2 \text{ in } X\{i\} \longrightarrow \mathcal{A}, X\{i\} \quad (\text{E-LetCap}) \\
\mathcal{A}, (\text{let } X_1 \preceq X_2 \text{ in } v_1) v_2 \longrightarrow \mathcal{A}, \text{let } X_1 \preceq X_2 \text{ in } (v_1 v_2) \quad (\text{E-LetApp}) \\
\mathcal{A}, (\text{let } X_1 \preceq X_2 \text{ in } v) [p] \longrightarrow \mathcal{A}, \text{let } X_1 \preceq X_2 \text{ in } (v [p]) \quad (\text{E-LetInst}) \\
\mathcal{A}, \text{open } (\alpha, x) = (\text{let } X_1 \preceq X_2 \text{ in } v) \text{ in } e \longrightarrow \mathcal{A}, \text{let } X_1 \preceq X_2 \text{ in } (\text{open } (\alpha, x) = v \text{ in } e) \quad (\text{E-LetOpen}) \\
\frac{\mathcal{A} \vdash X_1\{i\} \Rightarrow X_2 \triangleright i}{\mathcal{A}, \text{if } (X_1\{i\} \Rightarrow X_2 \triangleright i) e_3 e_4 \longrightarrow \mathcal{A}, e_3} \quad (\text{E-IfCertYes}) \\
\frac{\mathcal{A} \vdash X_1\{i\} \not\Rightarrow X_2 \triangleright i}{\mathcal{A}, \text{if } (X_1\{i\} \Rightarrow X_2 \triangleright i) e_3 e_4 \longrightarrow \mathcal{A}, e_4} \quad (\text{E-IfCertNo}) \\
\mathcal{A}, \text{declassify } v t \longrightarrow \mathcal{A}, v \quad (\text{E-Dcls}) \\
\mathcal{A}, \text{endorse } v p \longrightarrow \mathcal{A}, v \quad (\text{E-Endr}) \\
\frac{\mathcal{E} \vdash X\{i\}}{\mathcal{A}, \text{acquire } X \triangleright i \longrightarrow \mathcal{A}, \text{inl } X\{i\}} \quad (\text{E-AcqYes}) \\
\frac{\mathcal{E} \not\vdash X\{i\}}{\mathcal{A}, \text{acquire } X \triangleright i \longrightarrow \mathcal{A}, \text{inr } *} \quad (\text{E-AcqNo})
\end{array}$$

Fig. 9. Additional evaluation rules of λ_{RP} with run-time authority

The typing judgments for run-time authority are of the form $\Delta; \Gamma; \pi \vdash e : t$, where π is the set of static capabilities available within the expression e . Given static capabilities π , we write $\pi(i)$ for the set of principals that have granted the permission i ; so $\pi(i) = \{p \mid p \triangleright i \in \pi\}$. In the rule T-Dcls, s is the set of principals whose authority is needed to perform the declassification, therefore the condition $\Delta \vdash s \preceq \pi(\text{declassify})$ says that the set of **declassify**-granting principals in the static authority is sufficient to act for s .

$$\begin{array}{c}
\frac{\Delta \vdash u - u' = s_1 \quad \Delta \vdash l - l' = s_2}{\Delta \vdash u_l - u'_l = s_1 \cup s_2} \quad (\text{D-Label}) \\
\Delta \vdash 1 - 1 = \cdot \quad (\text{D-Unit}) \\
\frac{\Delta \vdash t_1 - t'_1 = s_1 \quad \Delta \vdash t_2 - t'_2 = s_2}{\Delta \vdash (t_1 + t_2) - (t'_1 + t'_2) = s_1 \cup s_2} \quad (\text{D-Sum}) \\
\frac{\Delta \vdash t'_1 - t_1 = s_1 \quad \Delta \vdash t_2 - t'_2 = s_2}{\Delta \vdash (t_1 \rightarrow t_2) - (t'_1 \rightarrow t'_2) = s_1 \cup s_2} \quad (\text{D-Fun}) \\
\Delta \vdash P_p - P_p = \cdot \quad (\text{D-Name}) \\
\frac{\Delta, \alpha \preceq p \vdash t - t' = s}{\Delta \vdash (\forall \alpha \preceq p. t) - (\forall \alpha \preceq p. t') = s} \quad (\text{D-All}) \\
\frac{s' = \{p \mid \Delta \vdash d_2(p) \preceq d_1(p), \Delta \vdash d_1(p) \not\preceq d_2(p)\}}{\Delta \vdash \{d_1!s\} - \{d_2!s\} = s'} \quad (\text{D-Label})
\end{array}$$

Fig. 10. Declassification requisites

For robustness [Zdancewic and Myers 2001; Zdancewic 2003; Myers et al. 2004], we must ensure that the integrity of the data is reflected in the set of static capabilities available.⁶ To do so, we define an operator $\pi|l$, that restricts the capabilities in π to just those whose owners have delegated to principals present in the integrity portion of the label l . With respect to hierarchy Δ , the formal definition is:

$$\pi|\{d!s\} = \{p \triangleright i \in \pi \mid \exists q \in s. \Delta \vdash p \preceq q\}$$

The restriction operator occurs in the typing rules of branching constructs. For example, T-Case in Figure 7 is the modified form of T-Case in Figure 3 for the `case` expression.

The rule for capability certification also uses the restriction operator, but it also adds the permission $p \triangleright i$ before checking the branch taken when the capability provides privilege i (T-IfCert in Figure 8).

Note that the restriction is applied *after* the permission is added, to prevent the specious amplification of rights based on untrustworthy capabilities. At run time, the validity of a capability under the current acts-for hierarchy determines which branch of the certification expression is taken (E-IfCertYes and E-IfCertNo in Figure 9).

To verify that a capability grants permission for principal X_2 to perform some privileged operation i , the run-time system determines whether the issuer X_1 of the capability acts for the principal X_2 wanting to use the capability. It can be

⁶Although we do not carry out a robustness proof here, we expect that the methodology developed by Myers et al. [2004] could be applied in this setting as well.

implemented simply as $\mathcal{A} \vdash X_2 \preceq X_1$, but we want to keep the operation abstract and show a PKI implementation in Section 6.

Function types capture the static capabilities that may be used in the body of the function, and the modified rule for typechecking function application requires that the static capabilities π of the calling context are sufficient to invoke the function (T-Fun and T-App in Figure 7).

Finer-grained control of declassification can be incorporated into this framework by refining the `declassify` privilege identifier with more information, for instance, to give upper bounds on the data that may be declassified or distinguish between declassify expressions applied for different reasons (see Section 6.2).

4.2 Endorsement and delegation

Endorsement is a downgrading mechanism for integrity policies, in the same way that declassification is a downgrading mechanism for confidentiality policies. An endorsement by principal p expresses the trust of p about the integrity of some data, and hence requires p 's permission. Such an operation, like type-casts and declassification, does not have any run-time effect. We use the privilege identifier `endorse` $\in \mathcal{I}$ for such a privilege and use the expression `endorse` e p to express that p endorses the integrity of e (see Figure 6). Its typing and evaluation (T-Endr and E-Endr in Figure 8 and Figure 9) rules parallel those for declassification (T-Dcls and E-Dcls).

Delegation, on the other hand, allows the acts-for hierarchy to change during program execution—so far, the operational semantics has been given in terms of a fixed \mathcal{A} . When p delegates to q , then q may read or declassify all data readable or owned by p ; therefore, delegation is a very powerful operation that should require permission from p .

We add a new expression `let` ($e_1 \preceq e_2$) `in` e_3 that allows programmers to extend the acts-for hierarchy in the scope of the expression e_3 . Here, e_1 and e_2 must evaluate to run-time principals. Assuming their static names are p and q , respectively, the body e_3 is checked with the additional assumption that $p \preceq q$.

Because delegation is a privileged operation, it needs the static authority of principal p . We extend the set of privileges \mathcal{I} to include additional identifiers of the form `delegate` _{$p \preceq q$} . The constraint $\Delta \vdash p \preceq \pi(\text{delegate}_{p \preceq q})$ ensures that the capability to extend the acts-for hierarchy has been granted by p (T-LetDel in Figure 8).

E-LetDel says that the body of a let-delegation term is evaluated to a value under the extended acts-for hierarchy, but the original acts-for hierarchy is restored afterwards. This ensures that the delegation is local to e_3 .

For simple values that are fully evaluated such as units, injections, principal names and capabilities (to be introduced in the next subsection), we can forget the extended delegation (see E-LetUnit, E-LetInl, E-LetInr, E-LetName and E-LetCap in Figure 9).

For closure values that may contain a delayed computation such as function closures, polymorphic generalizations and packages, the delayed computation may capture the delegation constraint. Therefore, for our dynamic semantics to be type-preserving (Theorem 14), `let` ($X_1 \preceq X_2$) `in` v for such values v becomes a closure value (see the definition of values in Figure 6) and auxiliary evaluation rules (see E-

LetApp, E-LetInst and E-LetOpen in Figure 9) are necessary to *lift* or *re-associate* such closures with each computation rule.

4.3 Acquiring capabilities

So far, this paper has not addressed how capability objects are obtained by the running program. Because capabilities represent privileges conferred to the program by run-time principals, they must be provided by the run-time system—they represent part of the dynamic execution environment. In practice, capabilities may be created in a variety of ways: the operating system may create an appropriate set of capabilities after authenticating a user. If the capabilities are implemented via digital certificates, then they may be obtained over the network using the underlying PKI. Capabilities may also be generated by the system in response to user input, for instance after prompting for user confirmation via a secure terminal.

To hide the details of the mechanism for producing capabilities, we model the external environment as a black box \mathcal{E} and write $\mathcal{E} \vdash X\{i\}$ to indicate that environment \mathcal{E} produces the capability $X\{i\}$. Using the expression `acquire $e \triangleright i$` , where e evaluates to a run-time principal, the program can query the environment to see whether a given capability is available. This operation either returns the corresponding capability object $X\{i\}$ or indicates failure by returning `*`. See T-Acq, E-AcqYes and E-AcqNo in Figure 8 and Figure 9.

A common programming idiom is to obtain a run-time capability using `acquire`, certify the capability, and, if both checks succeed, act using the newly acquired abilities:

```

case (acquire user  $\triangleright$  declassify)
   $\lambda cap:C$ . if (cap  $\Rightarrow$  user  $\triangleright$  declassify)
    (declassify data t) (...)
   $\lambda x:1$ . ...

```

When written in this way, there appears to be a lot of redundancy in these constructs. However, for the sake of modularity and flexibility, we separate the introduction of a capability (`acquire`) from its validation (the `if` test) and the use of the conferred privileges (the `declassify`). A surface language like Jif, would provide syntactic sugar that combines the first two, the last two, or even all three of these operations. Treating these features independently also allows more flexibility for the programmer. For instance, the ability to pass capabilities as first class objects is important in distributed settings, where one host may manufacture a capability and send it to a second host that can verify the capability and act using the privileges (see Section 6.2).

5. TYPE-SAFETY

As another theoretical contribution of this paper, we have extended the type-safety result (Theorem 1) in Section 2 to the full language with downgrading and authority. This property is specified as follows.

THEOREM 6 (TYPE-SAFETY).

- (1) *Progress*: If $\mathcal{A}; \pi \vdash e : t$, then $e = v$ or $\mathcal{A}, e \longrightarrow \mathcal{A}, e'$ for some e' .
- (2) *Preservation*: If $\mathcal{A}; \pi \vdash e : t$ and $\mathcal{A}, e \longrightarrow \mathcal{A}, e'$, then $\mathcal{A}'; \pi' \vdash e' : t$ for some \mathcal{A}' and π' such that $\mathcal{A} \preceq \mathcal{A}'$ and $\pi \preceq \pi'$.

REMARK 7. *We have not proved a corresponding noninterference result for λ_{RP} with the run-time authority because in this setting we are primarily concerned with regulating declassification, which intentionally breaks noninterference. Other work on a very similar language [Tse and Zdancewic 2005] formally proves that well-typed programs not containing declassification or delegation satisfy noninterference.*

The rest of this section sets up lemmas necessary to prove progress and preservation of our full language.

5.1 Progress

The canonical form of a value can be uniquely determined by the inversion of typing rules, except that any value can also be a delegation closure $\mathbf{let} X_1 \preceq X_2 \mathbf{in} v$ (as discussed at the end of Section 4.2).

LEMMA 8 (CANONICAL FORMS). *If $\mathcal{A}; \pi \vdash v : t$, then either $v = \mathbf{let} X_1 \preceq X_2 \mathbf{in} v'$, or one of the following holds:*

- (1) *If $\mathcal{A}; \pi \vdash v : (t_1 + t_2)_l$, then $v = \mathbf{inl} v_1$, or $v = \mathbf{inr} v_2$.*
- (2) *If $\mathcal{A}; \pi \vdash v : ([\pi_2] t_1 \rightarrow t_2)_l$, then $v = \lambda x : t. e$.*
- (3) *If $\mathcal{A}; \pi \vdash v : (\mathbf{P}_X)_l$, then $v = X$.*
- (4) *If $\mathcal{A}; \pi \vdash v : \mathbf{C}_l$, then $v = X\{i\}$.*
- (5) *If $\mathcal{A}; \pi \vdash v : (\forall \alpha \preceq p. t)_l$, then $v = \Lambda \alpha \preceq p'. e$, or $v = \mathbf{let} X_1 \preceq X_2 \mathbf{in} v'$.*
- (6) *If $\mathcal{A}; \pi \vdash v : (\exists \alpha \preceq p. t)_l$, then $v = \mathbf{pack} (p \preceq p', e)$.*
- (7) *If $\mathcal{A}; \pi \vdash v [p] : t$, then $p = X$*

THEOREM 9 (PROGRESS). *If $\mathcal{A}; \pi \vdash e : t$, then $e = v$ or $\mathcal{A}, e \longrightarrow \mathcal{A}, e'$.*

PROOF. By induction on typing derivations. In case of T-Unit, T-Name, T-Cap, T-Fun, or T-All, the expression is a value.

For the subcases when Lemma 8 (canonical forms) is applied, if the form is a delegation closure, then progress is satisfied with some lifting rule described at the end of Section 4.2. For example, if the term is $e = v_1 v_2$ where $v_1 = \mathbf{let} X_1 \preceq X_2 \mathbf{in} v'_1$, then $e' = \mathbf{let} X_1 \preceq X_2 \mathbf{in} (v'_1 v_2)$.

$\frac{x : t \in \cdot}{\text{---T-Var: } \mathcal{A}; \pi \vdash x : t}$

Since the empty term context \cdot cannot contain any variable binding $x : t$, this case does not apply.

$\frac{\mathcal{A}; \pi \vdash e_1 : t_1 \quad \vdash l}{\text{---T-Inl: } \mathcal{A}; \pi \vdash \mathbf{inl} e_1 : (t_1 + t_2)_l}$

By IH on e_1 ,

- (1) $e_1 = v$: $e = \mathbf{inl} v$ is a value.
- (2) $\mathcal{A}, e_1 \longrightarrow \mathcal{A}, e'_1$: by evaluation context $E = \mathbf{inl} E$, we have $e' = \mathbf{inl} e'_1$.

---T-Inr: symmetric to T-Inl.

$$\text{---T-Case: } \frac{\mathcal{A}; \pi \vdash e_0 : (t_1 + t_2)_l \quad \mathcal{A} \vdash \pi_2 \preceq (\pi|l) \quad \mathcal{A}; \pi \vdash v_1 : ([\pi_2] t_1 \rightarrow t_0)_l \quad \mathcal{A}; \pi \vdash v_2 : ([\pi_2] t_2 \rightarrow t_0)_l}{\mathcal{A}; \pi \vdash \text{case } e_0 \ v_1 \ v_2 : t_0 \sqcup l}$$

By IH on e_0 ,

- (1) $\mathcal{A}, e_0 \longrightarrow \mathcal{A}, e'_0$: by evaluation context $E = \text{case } E \ v_1 \ v_2$, we have $e' = \text{case } e'_0 \ v_1 \ v_2$.
- (2) $e_0 = v$: by Lemma 8 (canonical forms),
 - (a) $e_0 = \text{inl } v_0$: by E-CaseInl, $e' = v_1 \ v_0$.
 - (b) $e_0 = \text{inr } v_0$: by E-CaseInr, $e' = v_2 \ v_0$.

$$\text{---T-App: } \frac{\mathcal{A}; \pi \vdash e_1 : ([\pi_2] t_1 \rightarrow t_2)_l \quad \mathcal{A}; \pi \vdash e_2 : t_1 \quad \mathcal{A} \vdash \pi_2 \preceq (\pi|l)}{\mathcal{A}; \pi \vdash e_1 \ e_2 : t_2 \sqcup l}$$

By IH on e_1 and e_2 ,

- (1) $\mathcal{A}, e_1 \longrightarrow \mathcal{A}, e'_1$: by evaluation context $E = E \ e_2$, we have $e' = e'_1 \ e_2$.
- (2) $e_1 = v$ and $\mathcal{A}, e_2 \longrightarrow \mathcal{A}, e'_2$: by evaluation context $E = v \ E$, we have $e' = v \ e'_2$.
- (3) $e_1 = v_1$ and $e_2 = v_2$: by Lemma 8 (canonical forms), $e_1 = \lambda x : t. e_0$: by E-Fun, $e' = e_0\{v_2/x\}$.

$$\text{---T-IfDel: } \frac{\mathcal{A}; \pi \vdash e_1 : (\mathbb{P}_{p_1})_l \quad \mathcal{A}; \pi \vdash e_2 : (\mathbb{P}_{p_2})_l \quad \mathcal{A}, p_1 \preceq p_2; \pi \vdash e_3 : t_0 \quad \mathcal{A}; \pi \vdash e_4 : t_0}{\mathcal{A}; \pi \vdash \text{if } (e_1 \preceq e_2) \ e_3 \ e_4 : t_0 \sqcup l}$$

By IH on e_1 and e_2 ,

- (1) $\mathcal{A}, e_1 \longrightarrow \mathcal{A}, e'_1$: by evaluation context $E = \text{if } (E \preceq e_2) \ e_3 \ e_4$, we have $e' = \text{if } (e'_1 \preceq e_2) \ e_3 \ e_4$.
- (2) $e_1 = v$ and $\mathcal{A}, e_2 \longrightarrow \mathcal{A}, e'_2$: by evaluation context $E = \text{if } (v \preceq E) \ e_3 \ e_4$, we have $e' = \text{if } (v \preceq e'_2) \ e_3 \ e_4$.
- (3) $e_1 = v_1$ and $e_2 = v_2$: by Lemma 8 (canonical forms),
 - (a) $e_1 = X_1, e_2 = X_2$ and $\mathcal{A} \vdash X_1 \preceq X_2$: by E-IfDelYes, $e' = e_3$.
 - (b) $e_1 = X_1, e_2 = X_2$ and $\mathcal{A} \vdash X_1 \not\preceq X_2$: by E-IfDelNo, $e' = e_4$.

$$\text{---T-LetDel: } \frac{\mathcal{A}; \pi \vdash e_1 : (\mathbb{P}_{p_1})_l \quad \mathcal{A}; \pi \vdash e_2 : (\mathbb{P}_{p_2})_l \quad \mathcal{A}, p_1 \preceq p_2; \pi \vdash e_3 : t_0 \quad \mathcal{A} \vdash p_1 \preceq \pi(\text{delegate}_{p_1 \preceq p_2})}{\mathcal{A}; \pi \vdash \text{let } (e_1 \preceq e_2) \ \text{in } e_3 : t_0 \sqcup l}$$

By IH on e_1 and e_2 ,

- (1) $\mathcal{A}, e_1 \longrightarrow \mathcal{A}, e'_1$: by evaluation context $E = \text{let } (E \preceq e_2) \ \text{in } e_3$, we have $e' = \text{let } (e'_1 \preceq e_2) \ \text{in } e_3$.
- (2) $e_1 = v$ and $\mathcal{A}, e_2 \longrightarrow \mathcal{A}, e'_2$: by evaluation context $E = \text{let } (v \preceq E) \ \text{in } e_3$, we have $e' = \text{let } (v \preceq e'_2) \ \text{in } e_3$.
- (3) $e_1 = v_1$ and $e_2 = v_2$: by Lemma 8 (canonical forms), $e_1 = X_1$ and $e_2 = X_2$. Then, by IH on e_3 :
 - (a) $\mathcal{A}, e_3 \longrightarrow \mathcal{A}, e'_3$: by E-LetDel, $e' = \text{let } (X_1 \preceq X_2) \ \text{in } e'_3$.
 - (b) $e_3 = v$: $e = \text{let } (X_1 \preceq X_2) \ \text{in } v$ is a value.

- T-IfCert:
$$\frac{\mathcal{A} \vdash e_1 : \mathbf{C}_l \quad \mathcal{A} \vdash e_2 : (\mathbf{P}_p)_l \quad \mathcal{A}; ; (\pi, p \triangleright i) | l \vdash e_3 : t_0 \quad \mathcal{A}; ; \pi | l \vdash e_4 : t_0}{\mathcal{A} \vdash \mathbf{if} (e_1 \Rightarrow e_2 \triangleright i) e_3 e_4 : t_0 \sqcup l}$$
- By IH on e_1 and e_2 ,
- (1) $\mathcal{A}, e_1 \longrightarrow \mathcal{A}, e'_1$: by evaluation context $E = \mathbf{if} (E \Rightarrow e_2 \triangleright i) e_3 e_4$, we have $e' = \mathbf{if} (e'_1 \Rightarrow e_2 \triangleright i) e_3 e_4$.
 - (2) $e_1 = v$ and $\mathcal{A}, e_2 \longrightarrow \mathcal{A}, e'_2$: by evaluation context $E = \mathbf{if} (v \Rightarrow E \triangleright i) e_3 e_4$, we have $e' = \mathbf{if} (v \Rightarrow e'_2 \triangleright i) e_3 e_4$.
 - (3) $e_1 = v_1$ and $e_2 = v_2$: by Lemma 8 (canonical forms),
 - (a) $e_1 = X_1\{i\}$, $e_2 = X_2$ and $\mathcal{A} \vdash X_1\{i\} \Rightarrow X_2 \triangleright i$: by E-IfCertYes, $e' = e_3$.
 - (b) $e_1 = X_1\{i_1\}$, $e_2 = X_2$ and $\mathcal{A} \vdash X_1\{i_1\} \not\Rightarrow X_2 \triangleright i_2$: by E-IfCertNo, $e' = e_4$.

- T-Dcls:
$$\frac{\mathcal{A} \vdash e_0 : t_2 \quad \mathcal{A} \vdash t_2 - t_1 = s \quad \mathcal{A} \vdash s \preceq \pi(\mathbf{declassify})}{\mathcal{A} \vdash \mathbf{declassify} e_0 t_1 : t_1}$$
- By the evaluation context $E = \mathbf{declassify} E t_1$, or by E-Dcls.

- T-Endr:
$$\frac{\Delta; \Gamma; \pi \vdash e : t \quad \Delta \vdash p \preceq \pi(\mathbf{endorse})}{\Delta; \Gamma; \pi \vdash \mathbf{endorse} e p : t \sqcap \{!p\}}$$
- By the evaluation context $E = \mathbf{endorse} E t_1$, or by E-Endr.

- T-Acq:
$$\frac{\mathcal{A}; ; \pi \vdash e_0 : (\mathbf{P}_p)_l}{\mathcal{A}; ; \pi \vdash \mathbf{acquire} e_0 \triangleright i : (\mathbf{C}_l + 1_l)_l}$$
- By IH on e_0 ,
- (1) $\mathcal{A}, e_0 \longrightarrow \mathcal{A}, e'_0$: by evaluation context $E = \mathbf{acquire} E \triangleright i$, we have $e' = \mathbf{acquire} e'_0 \triangleright i$.
 - (2) $e_0 = v$: by Lemma 8 (canonical forms), $e_0 = X\{i\}$. Then, either
 - (a) $\mathcal{E} \vdash X\{i\}$: by E-AcqYes, $e' = \mathbf{inl} X\{i\}$.
 - (b) $\mathcal{E} \not\vdash X\{i\}$: by E-AcqNo, $e' = \mathbf{inr} *$.

- T-Inst:
$$\frac{\mathcal{A}; ; \pi \vdash e_0 : (\forall \alpha \preceq p. t_0)_l}{\mathcal{A}; ; \pi \vdash e_0 [p] : t_0 \sqcup l}$$
- By IH on e_0 ,
- (1) $\mathcal{A}, e_0 \longrightarrow \mathcal{A}, e'_0$: by evaluation context $E = E [p]$, we have $e' = e'_0 [p]$.
 - (2) $e_0 = v$: by Lemma 8 (canonical forms), $e_0 = \Lambda \alpha \preceq p. e_1$ and $p = X$: by E-All, $e' = e_1\{X/\alpha\}$.

- T-Pack:
$$\frac{\mathcal{A}; ; \pi \vdash e_0 : t\{q/\alpha\} \quad \Delta \vdash p \preceq q \quad \Delta \vdash l}{\mathcal{A}; ; \pi \vdash \mathbf{pack} (X_1 \preceq X_2, e_0) : (\exists \alpha \preceq q. t)_l}$$
- By IH on e_1 ,
- (1) $e_0 = v$: $e = \mathbf{pack} (X_1 \preceq X_2, v)$ is a value.
 - (2) $\mathcal{A}, e_0 \longrightarrow \mathcal{A}, e'_0$: by evaluation context $E = \mathbf{pack} (X_1 \preceq X_2, E)$, we have $e' = \mathbf{pack} (X_1 \preceq X_2, e'_0)$.

$$\frac{\mathcal{A}; \pi \vdash e_1 : (\exists \alpha \preceq p. t_1)_l \quad \mathcal{A}, \alpha \preceq p; x : t_1; \vdash e_2 : t_2 \quad \alpha \notin \text{ftv}(\Gamma) \cup \text{ftv}(t_2)}{\mathcal{A}; \pi \vdash \text{open}(\alpha, x) = e_1 \text{ in } e_2 : t_2 \sqcup l}$$

—T-Open:

By IH on e_1 ,

- (1) $\mathcal{A}, e_1 \longrightarrow \mathcal{A}, e'_1$: by evaluation context $E = \text{open}(\alpha, x) = E \text{ in } e_2$, we have $e' = \text{open}(\alpha, x) = e'_1 \text{ in } e_2$.
- (2) $e_1 = v$: by Lemma 8 (canonical forms), if $e_1 = \text{pack}(X_1 \preceq X_2, e)$, then by E-Some, $e' = e_2\{X_1/\alpha, v/x\}$.

$$\frac{\mathcal{A}; \pi \vdash e_0 : t_1 \quad \mathcal{A} \vdash t_1 \leq t_2}{\mathcal{A}; \pi \vdash e_0 : t_2}$$

—T-Sub:

By IH on e_0 .

□

5.2 Preservation

LEMMA 10 (INVERSION).

- (1) If $\Delta; \Gamma; \pi \vdash \text{inl } v : (t_1 + t_2)_l$, then $\Delta; \Gamma; \pi \vdash v : t_1$.
- (2) If $\Delta; \Gamma; \pi \vdash \text{inr } v : (t_1 + t_2)_l$, then $\Delta; \Gamma; \pi \vdash v : t_2$.
- (3) If $\Delta; \Gamma; \pi \vdash \lambda x : t_1. e : (t_1 \rightarrow t_2)_l$, then $\Delta; \Gamma, x : t_1 \vdash e : t_2$.
- (4) If $\Delta; \Gamma; \pi \vdash \Lambda \alpha \preceq p. e : (\forall \alpha \preceq p. t)_l$, then $\Delta, \alpha \preceq p; \Gamma; \pi \vdash e : t$.
- (5) If $\Delta; \Gamma; \pi \vdash \text{pack}(p \preceq q, e) : (\exists \alpha \preceq p. t)_l$, then $\Delta; \Gamma; \pi \vdash e : t\{q/\alpha\}$.

PROOF. By normalizing the typing derivations (via collapsing multiple applications of T-Sub into one application of T-Sub). □

LEMMA 11 (WEAKENING).

- (1) If $\Delta; \Gamma; \pi \vdash e : t$, then $(\Delta, p_1 \preceq p_2); (\Gamma, x : t'); (\pi, p \triangleright i) \vdash e : t$.
- (2) If $\Delta \vdash t_1 \leq t_2$, then $\Delta, p \preceq q \vdash t_1 \leq t_2$.
- (3) If $\Delta \vdash l_1 \sqsubseteq l_2$, then $\Delta, p \preceq q \vdash l_1 \sqsubseteq l_2$.
- (4) If $\Delta \vdash c_1 \sqsubseteq c_2$, then $\Delta, p \preceq q \vdash c_1 \sqsubseteq c_2$.
- (5) If $\Delta \vdash p_1 \preceq p_2$, then $\Delta, p \preceq q \vdash p_1 \preceq p_2$.

LEMMA 12 (SUBSTITUTION FOR SUBTYPING).

- (1) If $\Delta \vdash t_1 \leq t_2$, $\delta \models \Delta$ and $\mathcal{A} = \delta(\Delta)$, then $\mathcal{A} \vdash \delta(t_1) \leq \delta(t_2)$
- (2) If $\Delta \vdash l_1 \sqsubseteq l_2$, $\delta \models \Delta$ and $\mathcal{A} = \delta(\Delta)$, then $\mathcal{A} \vdash \delta(l_1) \sqsubseteq \delta(l_2)$
- (3) If $\Delta \vdash c_1 \sqsubseteq c_2$, $\delta \models \Delta$ and $\mathcal{A} = \delta(\Delta)$, then $\mathcal{A} \vdash \delta(c_1) \sqsubseteq \delta(c_2)$
- (4) If $\Delta \vdash p_1 \preceq p_2$, $\delta \models \Delta$ and $\mathcal{A} = \delta(\Delta)$, then $\mathcal{A} \vdash \delta(p_1) \preceq \delta(p_2)$
- (5) If $\Delta, \alpha \preceq p \vdash t_1 \leq t_2$ and $\Delta \vdash p' \preceq p$, then $\Delta\{p'/\alpha\} \vdash t_1\{p'/\alpha\} \leq t_2\{p'/\alpha\}$.
- (6) If $\Delta, \alpha \preceq p \vdash l_1 \sqsubseteq l_2$ and $\Delta \vdash p' \preceq p$, then $\Delta\{p'/\alpha\} \vdash l_1\{p'/\alpha\} \sqsubseteq l_2\{p'/\alpha\}$.
- (7) If $\Delta, \alpha \preceq p \vdash c_1 \sqsubseteq c_2$ and $\Delta \vdash p' \preceq p$, then $\Delta\{p'/\alpha\} \vdash c_1\{p'/\alpha\} \sqsubseteq c_2\{p'/\alpha\}$.
- (8) If $\Delta, \alpha \preceq p \vdash p_1 \preceq p_2$ and $\Delta \vdash p' \preceq p$, then $\Delta\{p'/\alpha\} \vdash p_1\{p'/\alpha\} \preceq p_2\{p'/\alpha\}$.

The last four rules are special cases of the first four. The first four rules are used in proving Lemma 4 (substitution for logical relations), while the last four are used in proving Lemma 13 (substitution for typing).

LEMMA 13 (SUBSTITUTION FOR TYPING).

- (1) If $\Delta; \Gamma; \pi \vdash e : t$, $\delta \models \Delta$, $\mathcal{A} = \delta(\Delta)$ and $\mathcal{A} \vdash \gamma \models \delta(\Gamma)$, then $\mathcal{A}; ; \pi \vdash \gamma\delta(e) : \delta(t)$.
- (2) If $\Delta; \Gamma, x : t'; \pi \vdash e : t$ and $\Delta; \Gamma; \pi \vdash v : t'$, then $\Delta; \Gamma; \pi \vdash e\{v/x\} : t$.
- (3) If $\Delta, \alpha \preceq p; \Gamma; \pi \vdash e : t$ and $\Delta \vdash X \preceq p$, then $\Delta\{X/\alpha\}; \Gamma\{X/\alpha\}; \pi\{X/\alpha\} \vdash e\{X/\alpha\} : t\{X/\alpha\}$.

The last two rules are special cases of the first. The first rule is used in proving Lemma 4 (substitution for logical relations), while the last two are used in proving Theorem 14 (preservation).

THEOREM 14 (PRESERVATION). *If $\mathcal{A}; ; \pi \vdash e : t$ and $\mathcal{A}, e \longrightarrow \mathcal{A}', e'$, then $\mathcal{A}'; ; \pi' \vdash e' : t$ for some \mathcal{A}' and π' such that $\mathcal{A} \preceq \mathcal{A}'$ and $\pi \preceq \pi'$.*

PROOF. By induction on the typing derivations. For steps taken with some evaluation context, the typing holds simply because of induction hypothesis. In case of T-Var, T-Unit, T-Fun, T-Name, T-Cap, T-Gen, or T-Pack, there is no evaluation.

For steps taken with some lifting rule described at the end of Section 4.2, the typing holds simply because of weakening. For example, in case of function applications, $\mathcal{A}, (\text{let } X_1 \preceq X_2 \text{ in } v_1) v_2 \longrightarrow \mathcal{A}, \text{let } X_1 \preceq X_2 \text{ in } (v_1 v_2)$. By Lemma 11 (weakening), $\mathcal{A}, X_1 \preceq X_2; ; \pi_1 \vdash v_2 : t_1$. The result follows by T-App.

$$\text{—T-Case: } \frac{\mathcal{A}; ; \pi \vdash e_0 : (t_1 + t_2)_l \quad \mathcal{A} \vdash \pi_2 \preceq (\pi|l) \quad \mathcal{A}; ; \pi|l \vdash v_1 : ([\pi_2] t_1 \rightarrow t_0)_l \quad \mathcal{A}; ; \pi|l \vdash v_2 : ([\pi_2] t_2 \rightarrow t_0)_l}{\mathcal{A}; ; \pi \vdash \text{case } e_0 v_1 v_2 : t_0 \sqcup l}$$

Case E-CaseInl: $\mathcal{A}, \text{case } (\text{inl } v) v_1 v_2 \longrightarrow \mathcal{A}, v_1 v$. By Lemma 10 (inversion), $\mathcal{A}; ; \pi \vdash v : t_1$. The result follows by Lemma 11 (weakening) and T-App.

Case E-CaseInr: symmetric to E-CaseInl.

$$\text{—T-App: } \frac{\mathcal{A}; ; \pi_1 \vdash e_1 : ([\pi_2] t_1 \rightarrow t_2)_l \quad \mathcal{A}; ; \pi_1 \vdash e_2 : t_1 \quad \mathcal{A} \vdash \pi_2 \preceq (\pi_1|l)}{\mathcal{A}; ; \pi_1 \vdash e_1 e_2 : t_2 \sqcup l}$$

Case E-Fun: $\mathcal{A}, (\lambda x : t_1. e_0) v \longrightarrow \mathcal{A}, e_0\{v/x\}$. By Lemma 10 (inversion), $\Delta; \Gamma, x : t_1 \vdash e_0 : t_2$. Then, by Lemma 13 (substitution for typing), $\mathcal{A}; ; \pi \vdash e_0\{v/x\} : t_2$. The result follows by T-Sub.

$$\text{—T-IfDel: } \frac{\mathcal{A}; ; \pi \vdash e_1 : (P_{p_1})_l \quad \mathcal{A}; ; \pi \vdash e_2 : (P_{p_2})_l \quad \mathcal{A}, p_1 \preceq p_2; ; \pi \vdash e_3 : t_0 \quad \mathcal{A}; ; \pi \vdash e_4 : t_0}{\mathcal{A}; ; \pi \vdash \text{if } (e_1 \preceq e_2) e_3 e_4 : t_0 \sqcup l}$$

Case E-IfDelYes: $\mathcal{A}, \text{if } (X_1 \preceq X_2) e_3 e_4 \longrightarrow \mathcal{A}, e_3$. Let $\mathcal{A}' = \mathcal{A}, p_1 \preceq p_2$. The result follows by Lemma 11 (weakening) and T-Sub.

Case E-IfDelNo: similar to E-IfDelYes.

$$\text{—T-LetDel: } \frac{\mathcal{A}; ; \pi \vdash e_1 : (P_{p_1})_l \quad \mathcal{A}; ; \pi \vdash e_2 : (P_{p_2})_l \quad \mathcal{A}; ; \pi \vdash e_3 : t_0}{\mathcal{A}; ; \pi \vdash \text{let } (e_1 \preceq e_2) \text{ in } e_3 : t_0 \sqcup l}$$

Case E-LetDel: by Lemma 11 (weakening) and T-Sub.

$$\text{—T-IfCert: } \frac{\mathcal{A}; \pi \vdash e_1 : \mathbb{C}_l \quad \mathcal{A}; \pi \vdash e_2 : (\mathbb{P}_p)_l \quad \mathcal{A}; (\pi, p \triangleright i) | l \vdash e_3 : t_0 \quad \mathcal{A}; \pi | l \vdash e_4 : t_0}{\mathcal{A}; \pi \vdash \text{if } (e_1 \Rightarrow e_2 \triangleright i) e_3 e_4 : t_0 \sqcup l}$$

Case E-IfCertYes: $\mathcal{A}, \text{if } (X_1\{i\} \Rightarrow X_2 \triangleright i) e_3 e_4 \longrightarrow \mathcal{A}, e_3$. Let $\pi' = \pi, p \triangleright i$. The result follows by T-Sub.

Case E-IfCertNo: similar to E-IfCertYes.

$$\text{—T-Dcls: } \frac{\mathcal{A}; \pi \vdash e_0 : t_2 \quad \mathcal{A} \vdash t_2 - t_1 = s \quad \mathcal{A} \vdash s \preceq \pi(\text{declassify})}{\mathcal{A}; \pi \vdash \text{declassify } e_0 t_1 : t_1}$$

Case E-Dcls: $\mathcal{A}, \text{declassify } v t_0 \longrightarrow \mathcal{A}, v$. Assume we only declassify a label at the top-level type, that is $\mathcal{A} \vdash u_{l'} - u_l = s$, where $t_2 = u_{l'}$ and $t_1 = u_l$. Since we can assign any label to the top-level type of a value (according to T-Unit, T-Inl, T-Inr, T-Fun, T-Name, T-Cap, T-All, and T-Pack), we can change the type of v from $\mathcal{A}; \pi \vdash v : u_{l'}$ to $\mathcal{A}; \pi \vdash v : u_l$.

If we declassify a label inside the structure of a type (in particular, the parameter type of a function), we need to weaken the theorem such that evaluation preserves types only in the erasure semantics. That is, if $\mathcal{A}; \pi \vdash e : t$ and $\mathcal{A}, [e] \longrightarrow \mathcal{A}, [e']$, then $\mathcal{A}; \pi \vdash [e'] : t$, where $[\cdot]$ is the type-erasure function. We omit the proof for this general case here.

$$\text{—T-Endr: } \frac{\Delta; \Gamma; \pi \vdash e : t \quad \Delta \vdash p \preceq \pi(\text{endorse})}{\Delta; \Gamma; \pi \vdash \text{endorse } e p : t \sqcap \{!p\}}$$

Case E-Endr: $\mathcal{A}, \text{endorse } v p \longrightarrow \mathcal{A}, v$. Similar to T-Dcls, we can change the type of v from $\mathcal{A}; \pi \vdash v : u_{l'}$ to $\mathcal{A}; \pi \vdash v : u_l$.

$$\text{—T-Acq: } \frac{\mathcal{A}; \pi \vdash e_0 : (\mathbb{P}_p)_l}{\mathcal{A}; \pi \vdash \text{acquire } e_0 \triangleright i : (\mathbb{C}_l + 1_l)_l}$$

Case E-AcqYes: $\mathcal{A}, \text{acquire } X \triangleright i \longrightarrow \mathcal{A}, \text{inl } X\{i\}$. By T-Cap and T-Inl.

Case E-AcqNo: similar to E-AcqYes.

$$\text{—T-Inst: } \frac{\mathcal{A}; \pi \vdash e_0 : (\forall \alpha \preceq p_2. t_0)_l \quad \mathcal{A} \vdash p_1 \preceq p_2}{\mathcal{A}; \pi \vdash e_0 [p_1] : t_0 \sqcup l}$$

Case E-All: $\mathcal{A}, (\Lambda \alpha \preceq p. e_0) [X] \longrightarrow \mathcal{A}, e_0\{X/\alpha\}$. By Lemma 10 (inversion), $\Delta; \Gamma, \alpha \vdash e_0 : t_0 \sqcup l$. Then, by Lemma 13 (substitution for typing), $\mathcal{A}; \pi \vdash e_0\{X/\alpha\} : t_0$. The result follows by Lemma 11 (weakening) and T-Sub.

$$\text{—T-Open: } \frac{\mathcal{A}; \pi \vdash e_1 : (\exists \alpha \preceq p. t_1)_l \quad \mathcal{A}, \alpha \preceq p; x : t_1; \vdash e_2 : t_2 \quad \alpha \notin \text{ftv}(\Gamma) \cup \text{ftv}(t_2)}{\mathcal{A}; \pi \vdash \text{open } (\alpha, x) = e_1 \text{ in } e_2 : t_2 \sqcup l}$$

Case E-Some: $\mathcal{A}, \text{open } (\alpha, x) = (\text{pack } (X_1 \preceq X_2, v)) \text{ in } e_2 \longrightarrow \mathcal{A}, e_2\{X_1/\alpha, v/x\}$.
 By Lemma 13 (substitution for typing), $\mathcal{A}; \pi \vdash e_2\{X_1/\alpha, v/x\} : t_2$. The result follows by T-Sub.

$$\text{—T-Sub: } \frac{\mathcal{A}; \pi \vdash e_0 : t_1 \quad \mathcal{A} \vdash t_1 \leq t_2}{\mathcal{A}; \pi \vdash e_0 : t_2}$$

By IH on e_0 .

□

6. PKI AND APPLICATION

6.1 Public key infrastructures

This section considers some possible implementations of run-time principals, concentrating on one interpretation in terms of a public key infrastructure.

If run-time principals are added to an information-flow type system whose programs are intended to run within a single, trusted execution environment, the implementation is straightforward: the trusted run time maintains an immutable (and persistent) mapping of principal names to unique identifiers, the acts-for hierarchy is a directed graph with nodes labeled by identifiers, and capabilities can be implemented as (unforgeable) handles to data structures created by the run-time system—this is the strategy currently taken by Jif.

If the programs are intended to run in a distributed setting, the implementation becomes more challenging. Fortunately, the appropriate machinery (principal names, delegation, and capabilities) has already been developed using public-key cryptography [Gasser and McDermott 1990; Howell and Kotz 2000]. We can interpret the acts-for hierarchy and run-time tests of λ_{RP} in terms of PKI as follows: run-time principals are implemented via public keys, the acts-for hierarchy is implemented via certificate chains, and capabilities are implemented as digitally signed certificates.

Formally, we have the following interpretation that maps the abstract syntax for privileges, capabilities, and the principal comparison tests of λ_{RP} into their concrete representations. Here, K_X is the public key corresponding to X and $K_X^{-1}\{\llbracket i \rrbracket\}$ is a certificate containing $\llbracket i \rrbracket$ signed using X 's private key. The strings **dcls** and **del** provide the content for the certificates, and they indicate the type of privilege granted by possession of the certificate. The remaining constructs (the acts-for relation and the privileged operations) are simply interpreted as tuples of data:

$$\begin{aligned} \llbracket X \rrbracket &= K_X \\ \llbracket X_1 \preceq X_2 \rrbracket &= (K_{X_1}, K_{X_2}) \\ \llbracket X \{i\} \rrbracket &= K_X^{-1}\{\llbracket i \rrbracket\} \\ \llbracket X \triangleright i \rrbracket &= (K_X, \llbracket i \rrbracket) \\ \llbracket \text{declassify} \rrbracket &= \text{dcls} \\ \llbracket \text{delegate}_{X_1 \preceq X_2} \rrbracket &= (\text{del}, K_{X_1}, K_{X_2}) \end{aligned}$$

The judgment $\mathcal{A} \vdash X_1 \Rightarrow X_2 \triangleright i$, which is used in the rule E-IfCertYes translates to the following rule:

$$\frac{(\mathbb{K}_{X_2}, \mathbb{K}_{X_1}) \in \llbracket \mathcal{A} \rrbracket^*}{\mathcal{A} \vdash \mathbb{K}_{X_1}^{-1}\{\llbracket i \rrbracket\} \Rightarrow (\mathbb{K}_{X_2}, \llbracket i \rrbracket)}$$

Here, the interpretation of the acts-for hierarchy, $\llbracket \mathcal{A} \rrbracket^*$, is a binary relation on public keys—the reflexive, transitive closure of the point-wise interpretation of the delegation pairs.

Given these definitions, it is clear how to interpret the capability verification—we use cryptographic primitives to verify that the digital certificate is signed by the corresponding public key. In the reflexive case, we need to check whether the capability $X_1\{i\}$ permits the privilege $X_2 \triangleright j$ (in this instance, the judgment is $\mathcal{A} \vdash X\{i\} \Rightarrow X \triangleright j$). The translation of the capability is $\mathbb{K}_{X_1}^{-1}\{\llbracket i \rrbracket\}$; the translation of the privilege is the pair $(\mathbb{K}_{X_2}, \llbracket j \rrbracket)$. Operationally, the implementation tries to use the key \mathbb{K}_{X_2} to verify the signature, and if the signature is valid (which implies that $\mathbb{K}_{X_1} = \mathbb{K}_{X_2}$) also checks to make sure that the contents of the certificate match (it checks whether $\llbracket i \rrbracket = \llbracket j \rrbracket$). If all of these checks succeed, the acts-for hierarchy permits the privilege (and evaluation proceeds according to rule E-IfCertYes). If the check fails, the implementation must check for a sequence of delegation certificates that permits the privilege.

The implementation uses graph reachability to test for transitive acts-for relations in \mathcal{A} . It is easy to show that the existence of a path in $\llbracket \mathcal{A} \rrbracket^*$ implies the existence of a valid certificate chain. In general, the justification for constraint $p_1 \preceq \pi(i)$ is the existence of some certificate chain of the form:

$$\mathbb{K}_{p_1}^{-1}\{\llbracket p_1 \preceq p_2 \rrbracket\} \leftrightarrow \mathbb{K}_{p_2}^{-1}\{\llbracket p_2 \preceq p_3 \rrbracket\} \leftrightarrow \dots \leftrightarrow \mathbb{K}_{p_{n-1}}^{-1}\{\llbracket p_{n-1} \preceq p_n \rrbracket\} \leftrightarrow \mathbb{K}_{p_n}^{-1}\{\llbracket i \rrbracket\}$$

Only key \mathbb{K}_{p_1} is needed to validate this chain, since each delegation certificate contains the key needed to validate the next certificate in the sequence. The PKI implementation must find such a chain at run time to justify granting privilege i .

The initial acts-for hierarchy contains only information relating principals to the universally trusted principal \top . The principal \top behaves as a certificate authority that generates private keys and issues certificates binding principal names to their corresponding public keys. To satisfy the axiom $\Delta \vdash X \preceq \top$, we assume that each host’s run-time is configured with $\mathbb{K}_X^{-1}\{\llbracket X \preceq \top \rrbracket\}$ and $(\mathbb{K}_X, \mathbb{K}_\top) \in \llbracket \mathcal{A} \rrbracket$ for each X —this information would be acquired by a host when it receives the binding between principal X and key \mathbb{K}_X from the certificate authority. The initial hierarchy consists of $(\mathbb{K}_X, \mathbb{K}_\top)$ pairs. As the program runs, additional delegation key pairs are added to the hierarchy by the **let** $(X_1 \preceq X_2)$ **in** \dots binding (corresponding to E-LetDel).

We illustrate this process by example. Consider the following program that takes in two capabilities and some data owned by Alice and attempts to declassify it.

```

1   $\lambda c_1 : \mathbb{C}. \lambda c_2 : \mathbb{C}. \lambda x : \text{bool}_{\{Alice\}}.$ 
2    if  $(c_1 \Rightarrow Alice \triangleright \text{delegate}_{Alice \preceq Bob})$ 
3      let  $(Alice \preceq Bob)$  in
4      if  $(c_2 \Rightarrow Bob \triangleright \text{declassify})$ 
5      declassify  $x \text{ bool}_{\{1\}}$ 
```

By the typing rule T-Dcls of declassification, line 5 needs the authority $p \triangleright$ **declassify** for some p acting for *Alice* because *Alice*'s policy is being weakened:

$$\vdash \text{bool}_{\{Alice : !\}} - \text{bool}_{\{!\}} = \{Alice\}$$

The PKI implementation justifies the presence of *Alice*'s authorization. Assume the acts-for hierarchy \mathcal{A} at line 1 is the default hierarchy consisting of only (K_X, K_\top) pairs. Line 2 uses $\llbracket Alice \rrbracket = K_{Alice}$ to verify the certificate $\mathcal{A} \vdash c_1 \Rightarrow (K_{Alice}, \llbracket i \rrbracket)$ where $\llbracket i \rrbracket = \llbracket \text{delegate}_{Alice \preceq Bob} \rrbracket = (\text{del}, K_{Alice}, K_{Bob})$. Since the acts-for hierarchy is otherwise empty, c_1 must be of the form $K_{Alice}^{-1}\{\llbracket i \rrbracket\}$ or $K_\top^{-1}\{\llbracket i \rrbracket\}$. The first certificate can be validated using only K_{Alice} ; the second can be validated starting from K_{Alice} by checking the certificate chain $K_{Alice}^{-1}\{\llbracket Alice \preceq \top \rrbracket\} \leftrightarrow K_\top^{-1}\{\llbracket i \rrbracket\}$. If one of these chains is valid, line 3 adds the delegation information into the hierarchy so that $(K_{Alice}, K_{Bob}) \in \llbracket \mathcal{A} \rrbracket$.

Similarly, there are two certificates c_2 that may justify the static condition

$$Alice \preceq \pi(\text{declassify}) = Alice \preceq Bob$$

required by rule T-Dcls in line 5. If $c_2 = K_{Bob}^{-1}\{\text{dcls}\}$, the static condition holds at runtime because we can find the chain:

$$K_{Alice}^{-1}\{\llbracket Alice \preceq Bob \rrbracket\} \leftrightarrow K_{Bob}^{-1}\{\text{dcls}\}$$

If $c_2 = K_\top^{-1}\{\text{dcls}\}$ we can find the chain:

$$K_{Alice}^{-1}\{\llbracket Alice \preceq Bob \rrbracket\} \leftrightarrow K_{Bob}^{-1}\{\llbracket Bob \preceq \top \rrbracket\} \leftrightarrow K_\top^{-1}\{\text{dcls}\}$$

6.2 Application to distributed banking

Figure 11 shows a more elaborate example λ_{RP} program that implements a distributed banking scenario in which a customer interacts with their bank through an ATM. The example uses a number of standard constructs such as integers, pairs, let-binding, and existential types with multiple arguments that are not in λ_{RP} , but could readily be added or encoded [Pierce 2002]. The main functions for the ATMs and the *Bank* are shown, along with the types of various auxiliary functions.

The static principals are *Bank* and ATM_1 through ATM_n , and there are two run-time principals, *user* and *agent*. The principal *user* is the customer at an ATM; *agent* is the *Bank*'s name for one of the n ATMs that may connect to the bank server. On the top of Figure 11 are the type declarations of the functions used, in the middle is the client code for ATM_j (a particular ATM), and at the bottom is the bank server code.

At the ATM_j , the customer logs in with the bank card and the password, revealing his identity $[user, user_{id}]$ and allowing ATM_j to act for him (represented by the capability c_{del}). Then ATM_j interacts with *user* to obtain his request such as withdrawing \$100. This interaction is modeled by the **acquire**. The ATM client packs the identities ATM_j and $user_{id}$ and the delegation c_{del} and the request c_{req} certificates into a message. To send the message over the channel to *Bank*, ATM_j gives up the ownership of the data by declassifying the message to have label $\{Bank : Bank!\}$. As a result of the transaction with the bank server, ATM_j obtains the new account balance of the customer. Finally, ATM_j prompts to determine whether the *user* wants a receipt, which requires a declassification certificate to print.

```

ATMj_main : [ATMj ▷ declassifynet]1 → 1
Bank_main : [Bank ▷ declassifynet]1 → 1
  request : ∃(agent, user). ((Pagent, Puser, C, C){Bank:Bank!}
    → int{agent:agent!agent})
  listen : 1 → ∃(agent, user).
    (Pagent, Puser, C, C, (int{agent:agent!} → 1)){Bank:Bank!Bank}
  login : 1 → (∃user. Puser, C){ATMj:ATMj!}
  print : int{!} → 1
  get : ∀user. Puser → int{Bank:Bank!}
  set : ∀user. Puser → int → 1

```

```

ATMj_main = λx : 1.
  open [user, (userid, cdel)] = login * in
  case (acquire userid ▷ withdraw100)
  λcreq : C. let message = (ATMj, userid, cdel, creq) in
    let data = declassifynet message (PATMj, Puser, C, C){Bank:Bank!} in
    let balance = request (pack ((ATMj, user), data) in
    case (acquire userid ▷ declassifyprt)
    λcprt : C. if (cprt ⇒ userid ▷ declassifyprt)
      let data = declassifyprt balance int{!} in
      print data
  ... // other banking options

```

```

Bank_main = λx : 1.
  open ((agent, user), (agentid, userid, cdel, creq, reply)) = listen * in
  if (cdel ⇒ userid ▷ delegateuser ≤ agent)
  let (userid ≤ agentid) in
  if (cdel ⇒ userid ▷ withdraw100)
  let old = get [user] userid in
  let balance = old - 100 in
  set [user] userid balance;
  let data = declassifynet balance int{user:user!} in
  reply data
  ... // other banking options

```

Fig. 11. A distributed banking example

This example makes use of fine-grained `declassify` privileges to distinguish between the printing (`declassifyprt`) and network send (`declassifynet`) uses of declassification. these variants have the same static and dynamic semantics as the `declassify` (as formulated in the last section), the subscripts are only annotations that explicitly distinguish different uses of declassification.

The bank server listens over the private channel and receives the message. The `listen` function also provides a `reply` channel so that the balance can be returned to

the same ATM. The server determines that *user* has logged in to ATM_j by verifying c_{del} , and if so, checks that the request capability is valid. If so, the server updates its database, and declassifies the resulting balance to be sent back to the ATM. In practice *Bank* will also want to log the certificates for auditing purposes.

In the functions *request* and *listen*, we assume the existence of a private network between ATM_j and *Bank*, which can be established using authentication and encryption. Since the network is private, the outgoing data must be readable only by the receiver; and, since the network is trusted, the incoming data has the integrity of the receiver. The labels of their types faithfully reflect this policy: for example, $\{Bank:Bank!\}$ vs. $\{agent:agent!agent\}$ in the type of *request*.

Note that the run-time authority for declassification and delegation are provided by the customer—they are acquired by the interaction of ATM_j and *user*. In contrast, in the types of ATM_j_main and $Bank_main$, the static capability requirements $[ATM_j \triangleright \text{declassify}_{net}]$ and $[Bank \triangleright \text{declassify}_{net}]$ indicate that the authorities to declassify to the network must be established from the caller.

Our type system does not prevent information leaks through computational effects such as printing or network input/output. Our ongoing research uses monads to incorporate such static analysis, in the same spirit as the work by Cray et al. [2004].

7. DISCUSSION

7.1 Related work

The work nearest to ours is the Jif project, by Myers et al. [1999]. The Jif compiler supports run-time principals but its type system has not been shown to be safe. Our noninterference proof for λ_{RP} is a step in that direction. Jif also supports *run-time labels* [Zheng and Myers 2004], which are run-time representations of label annotations, and a `switch label` construct that lets programs inspect the labels at runtime. Although it is desirable to support both run-time labels and run-time principals, the two features are mostly orthogonal.

While the core λ_{RP} presented here is not immediately suitable for use by programmers (more palatable syntax would be needed), λ_{RP} can serve as a typed intermediate representations for languages like Jif. Moreover, this approach improves on the current implementation of the decentralized label model (DLM) because Jif does not support declassification of data owned by run-time principals, nor does it provide language support for altering the acts-for hierarchy. Our separation of static principals from their run-time representations also clarifies the type checking rules.

The ability to perform acts-for tests at runtime is closely related to *intensional type analysis*, which permits programs to inspect the structure of types at runtime. Our use of singleton types like P_p to tie run-time tests to static types follows the work by Cray, Weirich, and Morrisett [2002]. Static capability sets π in our type system are a form of *effects* [Jouvelot and Gifford 1991], which have also been used to regulate the read and write privileges in type systems for memory management [Cray et al. 1999]. Simonet and Pottier [2004] have an interesting example of using *guarded algebraic datatypes* to express such run-time types and run-time tests.

The robustness condition on the set of run-time capabilities is very closely related to Java’s stack inspection model [Wallach and Felten 1998; Wallach et al. 2000; Fournet and Gordon 2002; Pottier et al. 2001]. In particular, the *enable-privilege* operation corresponds to our $\text{if } (e_1 \Rightarrow e_2 \triangleright i) e_3 e_4$ and the check-privileges operation corresponds to the constraint on π in the **declassify** rule. The restriction $\pi|l$ of capability sets in the type-checking rule for function application corresponds to taking the intersection of privilege sets in these type systems. However, stack inspection is *not* robust in the sense that data returned from an untrusted context can influence the outcome of privileged operations [Fournet and Gordon 2002]. In contrast, λ_{RP} tracks the integrity of data and restricts the capability sets according to the principals’ trust in the data—this is why the restriction $\pi|l$ appears in the typechecking rule for **case** expressions.

Banerjee and Naumann [2003] have previously shown how to mix stack inspection-style access control with information-flow analysis. They prove a noninterference result, which extends their earlier work on information-flow in Java-like languages [Banerjee and Naumann 2002]. Unlike their work, this paper considers run-time principals as well as run-time access control checks. Incorporating the principals used by the DLM into the privileges checked by stack inspection allows our type system to connect the information-flow policies to the access control policy, as seen in the typechecking rule for **declassify**.

We have proposed the use of public key infrastructures as a natural way to implement the authority needed to regulate declassification in the presence of run-time principals. Although the interpretation of principals as public keys and authorized actions as digitally signed certificates is not new, integrating these features in a language with static guarantees brings new insights to information-flow type systems. This approach should facilitate the development of software that interfaces with existing access-control mechanisms in distributed systems [Howell and Kotz 2000; Gasser and McDermott 1990].

Making the connection between PKI and the label model more explicit may have additional benefits. Myers and Liskov observed that the DLM acts-for relation is closely related to the speaks-for relation in the logical formulation of distributed access control by Abadi et al. [1993]. Adopting the local names of the SDSI/SPKI framework [Abadi 1998] may extend the analogy even further. Chothia et al. [2003] also use PKI to model typed cryptographic operations for distributed access control.

Lastly, although capability mechanism in λ_{RP} provides facilities for programming with static and run-time capabilities, we do not address the problem of *revocation*. It would be useful to find suitable language support for handling revocation, such as that found in the work by Jim and Gunter [2001; 2000], but we leave such pursuits to future work.

7.2 Conclusions

Information-flow type systems are a promising way to provide strong confidentiality and integrity guarantees. However, their practicality depends on their ability to interface with external security mechanisms, such as the access controls and authentication features provided by an operating system. Previous work has established noninterference only for information-flow policies that are determined at compile time, but such static approaches are not suitable for integration with run-time

security environments.

This paper addresses this problem in three ways: (1) We prove noninterference for an information-flow type system with run-time principals, which allow security policies to depend on the run-time identity of users. (2) We show how to safely extend this language with a robust access-control mechanism, a generalization of stack inspection, that can be used to control privileged operations such as declassification and delegation. (3) We sketch how the run-time principals and the acts-for hierarchy of the decentralized label model can be interpreted using public key infrastructures.

Our ongoing research attempts to use monads, in the same spirit as the dependency core calculus by Abadi et al. [1999], to simplify the design of the decentralized label model. In particular, we model all downgrading mechanisms uniformly as subtyping to allow a simple formulation and proof of a conditioned version of noninterference, even in the presence of downgrading.

ACKNOWLEDGMENT

The authors thank Steve Chong, Peng Li, Francois Pottier, Dimitrios Vytiniotis, Geoffrey Washburn, Stephanie Weirich, and the anonymous referees for their helpful suggestions and comments on earlier drafts of this work. This research was supported in part by NSF grant CCR-0311204, *Dynamic Security Policies*.

REFERENCES

- ABADI, M. 1998. On SDSI's linked local name spaces. *Journal of Computer Security* 6, 1-2, 3–21.
- ABADI, M., BANERJEE, A., HEINTZE, N., AND RIECKE, J. 1999. A core calculus of dependency. In *Proc. 26th ACM Symp. on Principles of Programming Languages (POPL)*. San Antonio, TX, 147–160.
- ABADI, M., BURROWS, M., LAMPSON, B. W., AND PLOTKIN, G. D. 1993. A calculus for access control in distributed systems. *Transactions on Programming Languages and Systems* 15, 4 (Sept.), 706–734.
- AGAT, J. 2000. Transforming out timing leaks. In *Proc. 27th ACM Symp. on Principles of Programming Languages (POPL)*. Boston, MA, 40–53.
- ASPINALL, D. 1994. Subtyping with Singleton Types. In *Computer Science Logic*. 1–15.
- BANERJEE, A. AND NAUMANN, D. A. 2002. Secure information flow and pointer confinement in a java-like language. In *Proc. of the 15th IEEE Computer Security Foundations Workshop*. 253–267.
- BANERJEE, A. AND NAUMANN, D. A. 2003. Using access control for secure information flow in a Java-like language. In *Proc. of the 16th IEEE Computer Security Foundations Workshop*. IEEE Computer Society Press, 155–169.
- CHOTHIA, T., DUGGAN, D., AND VITEK, J. 2003. Type-Based Distributed Access Control. In *Proc. of the IEEE Computer Security Foundations Workshop*. 170.
- CRARY, K., KLIGER, A., AND PFENNING, F. 2004. A monadic analysis of information flow security with mutable state. *Journal of Functional Programming* 15, 2 (Mar.), 249 – 291.
- CRARY, K., WALKER, D., AND MORRISSETT, G. 1999. Typed memory management in a calculus of capabilities. In *Proc. 26th ACM Symp. on Principles of Programming Languages (POPL)*. San Antonio, Texas, 262–275.
- CRARY, K., WEIRICH, S., AND MORRISSETT, G. 2002. Intensional polymorphism in type erasure semantics. *Journal of Functional Programming* 12, 6 (Nov.), 567–600.
- FOURNET, C. AND GORDON, A. 2002. Stack inspection: Theory and variants. In *Proc. 29th ACM Symp. on Principles of Programming Languages (POPL)*. 307–318.

- GASSER, M. AND MCDERMOTT, E. 1990. An architecture for practical delegation in a distributed system. In *Proc. IEEE Symposium on Security and Privacy*. IEEE Computer Society Press, 20–30.
- GOGUEN, J. A. AND MESEGUER, J. 1982. Security policies and security models. In *Proc. IEEE Symposium on Security and Privacy*. IEEE Computer Society Press, 11–20.
- GUNTER, C. A. AND JIM, T. 2000. Generalized certificate revocation. In *Proc. 27th ACM Symp. on Principles of Programming Languages (POPL)*. ACM Press, Boston, Massachusetts, 316–329.
- HEINTZE, N. AND RIECKE, J. G. 1998. The SLam calculus: Programming with secrecy and integrity. In *Proc. 25th ACM Symp. on Principles of Programming Languages (POPL)*. San Diego, California, 365–377.
- HOWELL, J. AND KOTZ, D. 2000. End-to-end authorization. In *Proc. USENIX Symp. on Operating Systems Design and Implementation (OSDI)*. 151–164.
- JIM, T. 2001. SD3: a trust management system with certificate revocation. In *IEEE Symposium on Security and Privacy*. 106–115.
- JOUVELOT, P. AND GIFFORD, D. K. 1991. Algebraic reconstruction of types and effects. In *ACM Symposium on Principles of Programming Languages*. 303–310.
- LI, P., MAO, Y., AND ZDANCEWIC, S. 2003. Information integrity policies. In *Proceedings of the Workshop on Formal Aspects in Security & Trust (FAST)*. 53–70.
- MITCHELL, J. C. 1996. *Foundations for Programming Languages*. Foundations of Computing Series. The MIT Press.
- MYERS, A. C., CHONG, S., NYSTROM, N., ZHENG, L., AND ZDANCEWIC, S. 1999. Jif: Java information flow.
- MYERS, A. C. AND LISKOV, B. 1998. Complete, safe information flow with decentralized labels. In *Proc. IEEE Symposium on Security and Privacy*. Oakland, CA, USA, 186–197.
- MYERS, A. C. AND LISKOV, B. 2000. Protecting privacy using the decentralized label model. *ACM Transactions on Software Engineering and Methodology* 9, 4, 410–442.
- MYERS, A. C., SABELFELD, A., , AND ZDANCEWIC, S. 2004. Enforcing Robust Declassification. In *Proc. of the IEEE Computer Security Foundations Workshop*. 172–186.
- PIERCE, B. C. 2002. *Types and Programming Languages*. MIT Press.
- PITTS, A. 1998. Existential Types: Logical Relations and Operational Equivalence. In *International Colloquium on Automata, Languages and Programming*. 309 – 326.
- POTTIER, F. AND CONCHON, S. 2000. Information flow inference for free. In *Proc. 5th ACM SIGPLAN International Conference on Functional Programming (ICFP)*. 46–57.
- POTTIER, F. AND SIMONET, V. 2002. Information flow inference for ML. In *Proc. 29th ACM Symp. on Principles of Programming Languages (POPL)*. Portland, Oregon, 319 – 330.
- POTTIER, F., SKALKA, C., AND SMITH, S. F. 2001. A Systematic Approach to Static Access Control. In *European Symposium on Programming*. 344 – 382.
- SABELFELD, A. AND MYERS, A. C. 2003. Language-based information-flow security. *IEEE Journal on Selected Areas in Communications* 21, 1 (Jan.), 5–19.
- SABELFELD, A. AND SANDS, D. 2001. A PER model of secure information flow in sequential programs. *Higher-Order and Symbolic Computation* 14, 1 (Mar.), 59–91.
- SIMONET, V. 2003. Flow caml in a nutshell. In *Proceedings of the first APPSEM-II workshop*, G. Hutton, Ed. 152–165.
- SIMONET, V. AND POTTIER, F. 2004. Constraint-Based Type Inference with Guarded Algebraic Data Types. Submitted to toplas.
- TSE, S. AND ZDANCEWIC, S. 2004. Run-time Principals in Information-flow Type Systems. In *IEEE Symposium on Security and Privacy*.
- TSE, S. AND ZDANCEWIC, S. 2005. Designing a Security-typed Language with Certificate-based Declassification. In *European Symposium on Programming*.
- VOLPANO, D., SMITH, G., AND IRVINE, C. 1996. A sound type system for secure flow analysis. *Journal of Computer Security* 4, 3, 167–187.
- WADLER, P. 1989. Theorems for Free! In *ACM Functional Programming Languages and Computer Architecture*. 347–359.

- WALLACH, D. S., APPEL, A. W., AND FELTEN, E. W. 2000. The security architecture formerly known as stack inspection: A security mechanism for language-based systems. *ACM Transactions on Software Engineering and Methodology* 9, 4 (Oct.), 341 – 378.
- WALLACH, D. S. AND FELTEN, E. W. 1998. Understanding Java stack inspection. In *Proc. IEEE Symposium on Security and Privacy*. Oakland, California, USA, 52–63.
- ZDANCEWIC, S. 2003. A type system for robust declassification. In *Proceedings of the Nineteenth Conference on the Mathematical Foundations of Programming Semantics*. Electronic Notes in Theoretical Computer Science.
- ZDANCEWIC, S. AND MYERS, A. C. 2001. Secure information flow and CPS. In *Proc. of the 10th European Symposium on Programming*. Lecture Notes in Computer Science, vol. 2028. 46–61.
- ZDANCEWIC, S. AND MYERS, A. C. 2002. Secure information flow via linear continuations. *Higher Order and Symbolic Computation* 15, 2/3, 209–234.
- ZHENG, L. AND MYERS, A. C. 2004. Dynamic Security Labels and Noninterference. In *Formal Aspects in Security and Trust*.

A. FULL SYNTAX

$p ::=$	α X	Principals variable name
$s ::=$	$\cdot \mid p, s$	Principal sets
$c ::=$	$\cdot \mid p : s$	Policies
$d ::=$	$\cdot \mid c; d$	Policy sets
$l ::=$	$\{d!s\}$	Labels
$\Delta ::=$	$\cdot \mid \Delta, p \preceq p$	Principal environments
$\mathcal{A} ::=$	$\cdot \mid \mathcal{A}, X \preceq X$	Acts-for hierarchies
$\Gamma ::=$	$\cdot \mid \Gamma, x : t$	Term environments
$\pi ::=$	$\cdot \mid \pi, p \triangleright i$	Authority
$\delta ::=$	$\cdot \mid \delta, \alpha \mapsto X$	Principal substitutions
$\gamma ::=$	$\cdot \mid \gamma, x \mapsto v$	Term substitutions
$t ::=$	u_l	Secure types
$u ::=$	1 $t + t$ $[\pi] t \rightarrow t$ $\forall \alpha \preceq p. t$ $\exists \alpha \preceq p. t$ P_p C	Plain types unit sum function universal existential principal capability
$v ::=$	$*$ $\text{inl } v$ $\text{inr } v$ $\lambda x : t. e$ $\Lambda \alpha \preceq p. e$	Values unit left injection right injection function generalization

	<code>pack</code> $(p \preceq q, e)$	packing
	X	principal constant
	<code>let</code> $(X_1 \preceq X_2)$ <code>in</code> v	let delegate
	$X\{i\}$	capability
$e ::=$		Terms
	v	value
	x	variable
	<code>inl</code> e	left injection
	<code>inr</code> e	right injection
	<code>case</code> $e v v$	sum case
	$e e$	application
	$e [p]$	instantiation
	<code>open</code> $(\alpha, x) = e$ <code>in</code> e	opening
	<code>if</code> $(e \preceq e)$ $e e$	if delegate
	<code>let</code> $(e \preceq e)$ <code>in</code> e	let delegate
	<code>if</code> $(e \Rightarrow e \triangleright i)$ $e e$	if certify
	<code>declassify</code> $e t$	declassify
	<code>endorse</code> $e p$	endorse
	<code>acquire</code> $e \triangleright i$	acquire
$i ::=$		Privileges
	<code>declassify</code>	declassification
	<code>endorse</code>	endorsement
	<code>delegate</code> _{$p \preceq p$}	delegation