Run-time Principals in Information-flow Type Systems

Stephen Tse Steve Zdancewic

Technical Report (MS-CIS-03-39)
University of Pennsylvania

Abstract

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 term 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 information-flow 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.

1 Introduction

Information-flow type systems are a promising approach for enforcing strong end-to-end confidentiality and integrity policies [27]. 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 means of integrating the policies expressed by the type system with external notions of principals such as that from public key infrastructure (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:

Stephen Tse (`stse@cis.upenn.edu`) and Steve Zdancewic (`stevez@cis.upenn.edu`). This research was supported in part by NSF grant CCR-0311204, *Dynamic Security Policies.* Appears in IEEE Symposium on Security and Privacy, 2004.
Last update: March 11, 2004.

- 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 [20] and FlowCaml [29].
- 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 [33, 32].
- We investigate the implementation of run-time principals via public key infrastructure. 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 static security label `{user:}`. In the decentralized label model [21, 22], 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 indicates that `String`s 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). 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`. 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 allows 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 [14, 2, 25, 36]. It should be possible to extend our results to Java-like languages by using the techniques of Banerjee and Naumann [6, 7].

The rest of the paper is organized as follows. The next section describes our language with run-time principals, including its type system and the noninterference proof. Section 3 considers adding declassification in the context of run-time principals. Section 4 suggests how the security policies admitted by our language may be integrated with traditional public key infrastructure and gives an extended example. The last section discusses related work and conclusions.

2 Information-flow type systems

2.1 Decentralized label model

The security model considered in this paper is a version of the decentralized label model (DLM) developed by Myers and Liskov [21, 22]. However, the labels in this paper include integrity constraints in addition to confidentiality constraints, because integrity constraints allow robust declassification (see Section 3).

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:

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

The confidentiality requirements of the DLM are composed of *reader policy components* of the form $p\mathtt{:}s$, 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 $\mathit{Alice}\mathtt{:}\mathit{Bob}, \mathit{Charles}$ says that *Alice*'s policy is that only *Bob* and *Charles* (and implicitly *Alice*) may read data with this label. The confidentiality part of the label consists of a set of policy components such that *all* of their restrictions must be obeyed—the principals able to read the data must be in the intersection of the reader permissions. For example, a data labeled with the two reader permissions $\mathit{Alice}\mathtt{:}\mathit{Bob}, \mathit{Charles}$ and $\mathit{Bob}\mathtt{:}\mathit{Charles}, \mathit{Eve}$ will be readable only by *Charles* and *Bob*.[1]

The information-flow type system described below ensures that data with a given confidentiality label will only flow to destinations that are at least that restrictive. 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.

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 '`!`':

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

Acts-for hierarchy The decentralized label model also includes *delegation* embodied by a binary *acts-for* relation between principals. This relation is reflexive and transitive, yielding a partial order 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 hierarchy must be taken into account when determining the restrictions imposed by a label. For example, consider the labels $\mathtt{\{}\mathit{Alice}\mathtt{:!}\mathit{Alice}\mathtt{\}}$ and $\mathtt{\{}\mathit{Bob}\mathtt{:!}\mathit{Bob}\mathtt{\}}$. Ignoring the acts-for hierarchy, these labels describe data readable and trusted only by *Alice* and *Bob*, respectively. However, if the relation

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

[2] It 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 [18].

$Alice \preceq Bob$ is in the acts-for hierarchy, then data with label $\mathtt{\{}Alice\,\mathtt{:}\,\mathtt{!}Alice\mathtt{\}}$ will be readable by *Bob*—because *Bob* acts for *Alice*, anything *Alice* can read *Bob* can too. Note that *Bob* does *not* trust the integrity of data with label $\mathtt{\{}Alice\,\mathtt{:}\,\mathtt{!}Alice\mathtt{\}}$—*Alice*'s trust in the data does not imply *Bob*'s trust. *Alice does* trust data with label $\mathtt{\{}Bob\,\mathtt{:}\,\mathtt{!}Bob\mathtt{\}}$, again because *Bob* acts for *Alice*, anything *Bob* trusts *Alice* does too.

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 $p \preceq q$.

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

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

$$\mathtt{\{}d_1\,\mathtt{!}\,s_1\mathtt{\}} \sqcup \mathtt{\{}d_2\,\mathtt{!}\,s_2\mathtt{\}} \stackrel{\text{def}}{=} \mathtt{\{}d_1 \cup d_2\,\mathtt{!}\,s_1 \cap s_2\mathtt{\}}$$

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 [22] but extended to include integrity sets):

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

$$\frac{\Delta \vdash p_1 \preceq p_2 \quad \forall p_2' \in s_2.\ \exists p_1' \in s_1.\ \Delta \vdash p_1' \preceq p_2'}{\Delta \vdash p_1\,\mathtt{:}\,s_1 \sqsubseteq p_2\,\mathtt{:}\,s_2}$$

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 acts-for hierarchy.

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 hold:

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

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

2.2 λ_{RP} and 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} omits several

$t \ ::= \ u_l$	Secure types	$e \ ::=$	Terms	$v \ ::=$	Values
$u \ ::=$	Base types	v	value	$*$	unit
$\mathtt{1}$	unit	x	variable	$\mathtt{inl}\ v$	left injection
$t + t$	sum	$\mathtt{inl}\ e$	left injection	$\mathtt{inr}\ v$	right injection
$t \rightarrow t$	function	$\mathtt{inr}\ e$	right injection	$\lambda x\!:\!t.\ e$	function
$\mathtt{P}_p$	principal	$\mathtt{case}\ e\ v\ v$	sum case	X	principal
$\forall \alpha \preceq p.\ t$	universal	$e\ e$	application	$\Lambda \alpha \preceq p.\ e$	polymorphism
		$\mathtt{if}\ (e \preceq e)\ e\ e$	if delegation		
		$e\ [p]$	instantiation		

Figure 1: Syntax of types, terms, and values for λ_{RP}

features which are important for practical programming. 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 [31, 4, 28, 25, 36].

Security types, base 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 base types (u) annotated with a label (l).

The unit, sum, and function types are standard [23]. There is only one value, written $*$, of type $\mathtt{1}$. Sum values are created by tagging another value v with either the left or right tag: $\mathtt{inl}\ v$ and $\mathtt{inr}\ v$, respectively. The $\mathtt{case}$ 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 base type, we take it to be the minimal label, $\mathtt{\{!}\top\mathtt{\}}$. For example, the type $\mathtt{1}_{\mathtt{\{!}\top\mathtt{\}}}$ can be written $\mathtt{1}$. We define the type of Booleans with label l to be $\mathtt{bool}_l \overset{\mathrm{def}}{=} (\mathtt{1} + \mathtt{1})_l$ with values $\mathtt{true} \overset{\mathrm{def}}{=} \mathtt{inl}\ *$ and $\mathtt{false} \overset{\mathrm{def}}{=} \mathtt{inr}\ *$. The expression $\mathtt{if}\ (e)\ e_1\ e_2$ is encoded as $\mathtt{case}\ e\ (\lambda x_1 \!:\! \mathtt{1}.\ e_1)\ (\lambda x_1 \!:\! \mathtt{1}.\ e_2)$, for some fresh names x_1 and x_2.

The last two kinds of types, $\mathtt{P}_p$ 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 $\mathtt{P}_{Alice}$ is the constant *Alice*. That is, $\mathtt{P}_p$ is a *singleton type* [5]; such types have previously been used to represent other kinds of run-time type information [9]. A program can perform a dynamic test of the acts-for relation between *Alice* and *Bob* using the expression $\mathtt{if}\ (Alice \preceq Bob)\ e_1\ e_2$.

The type $\forall \alpha \preceq p.\ t$ is a form of *bounded quantification* [23] over principals. This type introduces a principal variable, and it describes programs for which the static information about principal α is that the acts-for relation $\alpha \preceq p$ holds. For example, the type $t_0 = \forall \alpha \preceq Alice.\ \mathtt{bool}_{\mathtt{\{}\alpha\mathtt{:!\}}} \rightarrow \mathtt{bool}_{\mathtt{\{}\alpha\mathtt{:!\}}}$ describes functions whose parameter and return types are Booleans owned by any principal for whom *Alice* may act.

Term-level expressions bind 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 $f\ [Bob]\ \mathtt{true}$. 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 $\forall \alpha.\ t \overset{\mathrm{def}}{=} \forall \alpha \preceq \top.\ t$ and $\Lambda \alpha.\ e \overset{\mathrm{def}}{=} \Lambda \alpha \preceq \top.\ e$.

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:

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

$$\mathcal{A}, (\lambda x\!:\!t.\ e)\ v \longrightarrow \mathcal{A}, e\{v/x\} \qquad \text{(E-AppFun)}$$

$$\mathcal{A}, (\Lambda\alpha \preceq p.\ e)\ [X] \longrightarrow \mathcal{A}, e\{X/\alpha\} \qquad \text{(E-PAppAll)}$$

$$\mathcal{A}, \texttt{case}\ (\texttt{inl}\ v)\ v_1\ v_2 \longrightarrow \mathcal{A}, v_1\ v \qquad \text{(E-CaseInl)}$$

$$\mathcal{A}, \texttt{case}\ (\texttt{inr}\ v)\ v_1\ v_2 \longrightarrow \mathcal{A}, v_2\ v \qquad \text{(E-CaseInr)}$$

$$\frac{\mathcal{A} \vdash X_1 \preceq X_2}{\mathcal{A}, \texttt{if}\ (X_1 \preceq X_2)\ e_3\ e_4 \longrightarrow \mathcal{A}, e_3} \qquad \text{(E-IfDelYes)}$$

$$\frac{\mathcal{A} \vdash X_1 \not\preceq X_2}{\mathcal{A}, \texttt{if}\ (X_1 \preceq X_2)\ e_3\ e_4 \longrightarrow \mathcal{A}, e_4} \qquad \text{(E-IfDelNo)}$$

$$\frac{\begin{array}{c}\Delta; \Gamma \vdash e : (t_1 + t_2)_l \\ \Delta; \Gamma \vdash v_1 : (t_1 \to t)_l \\ \Delta; \Gamma \vdash v_2 : (t_2 \to t)_l\end{array}}{\Delta; \Gamma \vdash \texttt{case}\ e\ v_1\ v_2 : t \sqcup l} \qquad \text{(T-Case)}$$

$$\frac{\Delta \vdash l}{\Delta; \Gamma \vdash X : (\mathtt{P}_X)_l} \qquad \text{(T-PName)}$$

$$\frac{\Delta, \alpha \preceq p; \Gamma \vdash e : t \quad \alpha \notin \mathrm{dom}(\Delta) \quad \Delta \vdash l}{\Delta; \Gamma \vdash \Lambda\alpha \preceq p.\ e : (\forall\alpha \preceq p.\ t)_l} \qquad \text{(T-All)}$$

$$\frac{\begin{array}{c}\Delta; \Gamma \vdash e_1 : (\mathtt{P}_p)_l \quad \Delta; \Gamma \vdash e_2 : (\mathtt{P}_q)_l \\ \Delta, p \preceq q; \Gamma \vdash e_3 : t \quad \Delta; \Gamma \vdash e_4 : t\end{array}}{\Delta; \Gamma \vdash \texttt{if}\ (e_1 \preceq e_2)\ e_3\ e_4 : t \sqcup l} \qquad \text{(T-IfDel)}$$

$$\frac{\Delta; \Gamma \vdash e : (\forall\alpha \preceq q.\ t)_l \quad \Delta \vdash p \preceq q}{\Delta; \Gamma \vdash e\ [p] : t\{p/\alpha\} \sqcup l} \qquad \text{(T-PApp)}$$

Figure 2: Evaluation and typing rules

This function is parameterized by the principal variable α. The next parameter is a run-time principal *user* that has type $\mathtt{P}_\alpha$, 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.

2.3 Evaluation and typing rules

The operational semantics for λ_{RP} formalizes program evaluation, and the type system keeps track of invariants, which can be statically checked. In this subsection we show that the type system of λ_{RP} is sound by proving the progress and the preservation theorems. The noninterference theorem of λ_{RP} uses the soundness property to establish that program security can be checked statically. Figure 2 shows the rules for evaluation and typing.

Operational semantics The operational semantics of λ_{RP} is standard [23], 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 make a small step of evaluation to become $\mathcal{A}$ and e'. The full 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 3.2 considers run-time modification of $\mathcal{A}$ via delegation.

In Figure 2, E-AppFun 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-PAppAll, if a polymorphic term $\Lambda\alpha \preceq p.\ e$ is instantiated to a principal X, then X is substituted for α in e. 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 left-injection $\texttt{inl}\ v$, the first branch is applied to v. For example, using the Boolean encoding described earlier,

$$\begin{array}{rl} & \texttt{if}\ (\texttt{true})\ \mathit{Alice}\ \mathit{Bob} \\ \overset{\text{def}}{=} & \texttt{case}\ (\texttt{inl}\ \texttt{*})\ (\lambda y\!:\!\mathtt{1}.\ \mathit{Alice})\ (\lambda y\!:\!\mathtt{1}.\ \mathit{Bob}) \\ \longrightarrow & (\lambda y\!:\!\mathtt{1}.\ \mathit{Alice})\ \texttt{*} \\ \longrightarrow & \mathit{Alice} \end{array}$$

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.

Type system The type system is similar to those previously proposed [14, 36, 24], 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 $t_1 \to t$, and the second branch must be a function of type $t_2 \to t$. 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-PApp), this restriction on the output label is used to rule out implicit information flows [14, 36].

By T-PName, only a principal constant X has type $(\mathtt{P}_X)_l$. This *singleton property* ties the static type information and the run-time identity of principals—if a program expression has type $(\mathtt{P}_X)_l$ it is guaranteed to evaluate to the constant X. 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 $\alpha \notin \mathrm{dom}(\Delta)$ ensures the well-formedness of the environment—α is a fresh variable. T-PApp 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-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 mentioned above. 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 from above, 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 $\mathtt{bool}_{\{M:!\}}$ can be treated as though it has type $\mathtt{bool}_{\{\alpha:!\}}$.

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

Theorem 1 (Soundness). *(1) Progress: If $\mathcal{A} \vdash e : t$, then $e = v$ or $\mathcal{A}, e \longrightarrow \mathcal{A}, e'$. (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 standard for languages with subtyping [23]. Appendix contains the complete proof, which uses the following substitution lemma. The lemma says that if an open term e has type t, then the substituted term $\gamma\delta(e)$ has the substituted type $\delta(t)$—this result is also needed to prove noninterference later (Theorem 3 and Lemma 4). Substitution also respects subtyping for types, principals, labels and policies [30]. The notation $\delta \models \Delta$ denotes a substitution δ that assigns each free principal variable α in hierarchy Δ to a principal name X. Similarly, $\mathcal{A} \vdash \gamma \models \delta(\Gamma)$ denotes a term substitution γ that assigns each free term variable x in environment Γ to a value such that the assignment respects the typing $x : t$ in Γ.

Lemma 2 (Substitution for typing).
If $\Delta; \Gamma \vdash e : t$, $\delta \models \Delta$, $\mathcal{A} = \delta(\Delta)$ and $\mathcal{A} \vdash \gamma \models \delta(\Gamma)$, then $\mathcal{A} \vdash \gamma\delta(e) : \delta(t)$.

2.4 Noninterference

This section proves a noninterference theorem [12], 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 closed and well-typed but 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 program e to learn information about input x.

$$\frac{\begin{array}{c}\mathcal{A} \vdash \Gamma \quad \mathcal{A} \vdash \gamma \models \Gamma \quad \mathcal{A} \vdash \gamma' \models \Gamma \\ \forall (x : t \in \Gamma).\ \mathcal{A} \vdash \gamma(x) \sim_\zeta \gamma'(x) : t\end{array}}{\mathcal{A} \vdash \gamma \approx_\zeta \gamma' : \Gamma} \quad \text{(R-Sub)} \qquad \frac{\mathcal{A} \vdash l \not\sqsubseteq \zeta}{\mathcal{A} \vdash v \sim_\zeta v' : u_l} \quad \text{(R-Label)}$$

$$\frac{\begin{array}{c}\mathcal{A}, e \longrightarrow^* \mathcal{A}, v \quad \mathcal{A}, e' \longrightarrow^* \mathcal{A}, v' \\ \mathcal{A} \vdash e : t \quad \mathcal{A} \vdash e' : t \quad \mathcal{A} \vdash v \sim_\zeta v' : t\end{array}}{\mathcal{A} \vdash e \approx_\zeta e' : t} \quad \text{(R-Term)} \qquad \mathcal{A} \vdash \mathtt{*} \sim_\zeta \mathtt{*} : \mathtt{1}_l \quad \text{(R-Unit)}$$

$$\frac{\forall (\mathcal{A} \vdash v_2 \sim_\zeta v_2' : t_1).\ \mathcal{A} \vdash (v\ v_2) \approx_\zeta (v'\ v_2') : t_2 \sqcup l}{\mathcal{A} \vdash v \sim_\zeta v' : (t_1 \to t_2)_l} \quad \text{(R-Fun)} \qquad \mathcal{A} \vdash X \sim_\zeta X : (\mathtt{P}_X)_l \quad \text{(R-PName)}$$

$$\frac{\forall (\mathcal{A} \vdash X \preceq p).\ \mathcal{A} \vdash (v\ [X]) \approx_\zeta (v'\ [X]) : t \sqcup l}{\mathcal{A} \vdash v \sim_\zeta v' : (\forall \alpha \preceq p.\ t)_l} \quad \text{(R-All)} \qquad \frac{\mathcal{A} \vdash v \sim_\zeta v' : t_1}{\mathcal{A} \vdash \mathtt{inl}\ v \sim_\zeta \mathtt{inl}\ v' : (t_1 + t_2)_l} \quad \text{(R-Inl)}$$

Figure 3: Logical relations for types with labels

Theorem 3 (Noninterference). *If $\mathcal{A}; x : u_{l'} \vdash e : \mathtt{bool}_l$, $\mathcal{A} \vdash l' \not\sqsubseteq l$, $\mathcal{A} \vdash v : u_{l'}$ and $\mathcal{A} \vdash v' : u_{l'}$ then*

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

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 [19]. 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 3 shows the complete definition of the logical relation. We use the notation $\mathcal{A} \vdash \gamma \approx_\zeta \gamma' : \Gamma$ to denote two related substitutions, $\mathcal{A} \vdash e \approx_\zeta e' : t$ to denote two related computations, and $\mathcal{A} \vdash v \sim_\zeta v' : t$ to denote two related values. They are parameterized by a type t, an acts-for hierarchy $\mathcal{A}$ and an upper-bound ζ of the observer's security label.

By R-Subs, two substitutions are related at environment Γ if Γ is closed and if the substitutions assign all variables in environment Γ to related values. R-Term indicates that two terms are related at type t if they both have type t and 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, $\mathtt{*}$ is related only to itself and, similarly, by R-PName, 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 $\mathtt{inl}\ v$ and $\mathtt{inl}\ v'$, and if v and v' are related at t. By R-Fun, two values are related at $(t_1 \to t_2)_l$ 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 $t \sqcup l$.

Using these definitions, we strengthen the induction hypothesis of noninterference so that the theorem 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 $\Delta; \Gamma \vdash e : t$, $\delta \models \Delta$, $\mathcal{A} = \delta(\Delta)$ and $\mathcal{A} \vdash \gamma \approx_\zeta \gamma' : \delta(\Gamma)$, then $\mathcal{A} \vdash \gamma\delta(e) \approx_\zeta \gamma'\delta(e) : \delta(t)$.

Proof. We only give a proof sketch here; a complete proof can be found in Appendix. By Lemma 2, the terms $\gamma\delta(e)$ and $\gamma'\delta(e)$ are well-typed. It remains to show that $\mathcal{A}, \gamma\delta(e) \longrightarrow^* \mathcal{A}, v$ and $\mathcal{A}, \gamma'\delta(e') \longrightarrow^* \mathcal{A}, v'$ and $\mathcal{A} \vdash v \sim_\zeta v' : \delta(t)$, which we prove by induction on the typing derivations: For T-PName, the result follows by R-PName because $\gamma\delta(e) = \gamma'\delta(e) = X$ and $\delta((\mathtt{P}_X)_l) = (\mathtt{P}_X)_{\delta(l)}$.

For T-IfDel, the two terms in the condition are related by the induction hypothesis. By inversion, either $\mathcal{A} \vdash l \not\sqsubseteq \zeta$ or they are both related using R-PName. In the former the result follows trivially by R-Label. In the latter, the test conditions evaluate to X_1 and X_2. Then, both terms step to the same branch depending on whether $\mathcal{A} \vdash X_1 \preceq X_2$. The result follows because both branches are related by the induction hypothesis.

For T-All, $\gamma\delta(\Lambda\alpha \preceq p.\, e_0)$ evaluates to $\Lambda\alpha \preceq \delta(p).\ \gamma\delta(e_0)$ while $\gamma'\delta(\Lambda\alpha \preceq p.\, e_0)$ evaluates to $\Lambda\alpha \preceq \delta(p).\ \gamma'\delta(e_0)$. It remains to show that $\forall(\mathcal{A} \vdash X \preceq \delta(p))$:

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

By E-PAppAll, these two applications step to $\gamma\delta(e_0)\{X/\alpha\} = \gamma\delta_0(e_0)$ and $\gamma'\delta(e_0)\{X/\alpha\} = \gamma'\delta_0(e_0)$, where $\delta_0 = \delta, \alpha \mapsto X$. The result follows by the induction hypothesis because $\delta_0 \models \Delta, \alpha \preceq p$.

For T-PApp, the two terms on the left are related by the induction hypothesis. The two principals on the right are both $\delta(p_1)$ and, by $\Delta \vdash p_1 \preceq p_2$ and Lemma 2, we have $\mathcal{A} \vdash \delta(p_1) \preceq \delta(p_2)$. The result then follows by the definition of R-All.

For T-Sub, the result follows by Lemma 2 and the following properties of subtyping with respect to logical relations (which can be proved by induction on the subtyping derivations): (1) If $\mathcal{A} \vdash e \approx_\zeta e' : t$ and $\mathcal{A} \vdash t \leq t'$, then $\mathcal{A} \vdash e \approx_\zeta e' : t'$. (2) If $\mathcal{A} \vdash v \sim_\zeta v' : t$ and $\mathcal{A} \vdash t \leq t'$, then $\mathcal{A} \vdash v \sim_\zeta v' : t'$. □

3 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:

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

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* [35, 34]. 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 $\texttt{bool}_l$ 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

$$
\begin{array}{lll|lll|lll}
u & ::= \ldots & \text{Base types} & e & ::= \ldots & \text{Terms} & v & ::= \ldots & \text{Values} \\
 & [\pi]\ t \rightarrow t & \quad\text{function} & & \texttt{if}\ (e \Rightarrow e \triangleright i)\ e\ e & \quad\text{if certify} & & X\texttt{\{}i\texttt{\}} & \quad\text{capability} \\
 & \texttt{C} & \quad\text{capability} & & \texttt{declassify}\ e\ t & \quad\text{declassify} & & & \\
\pi & ::= \cdot \ \mid\ \pi, p \triangleright i & \text{Authority} & & & & & i \in \mathcal{I} & \text{Privileges}
\end{array}
$$

Figure 4: λ_{RP} with run-time authority

that x is `true`. If x has low integrity, for example, if it is untrusted by *Alice*, then $l \sqsubseteq pc$ 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 [35, 34].

3.1 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. Below, we consider using capabilities to regulate other privileged operations, such as delegation.

Figure 4 summarizes 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 $Alice \triangleright \texttt{declassify}$ to declassify on *Alice*'s behalf. The latter, run-time authority, is written $X\texttt{\{}i\texttt{\}}$ and represents an unforgeable capability created by principal X and authorizing privilege i. Capabilities have static type `C`.

A program can test a capability at run time to determine whether a principal has granted it privilege i using the expression $\texttt{if}\ (e_1 \Rightarrow e_2 \triangleright i)\ e_3\ e_4$. 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.

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 $[Alice \triangleright \texttt{declassify}]\ \texttt{bool}_{\texttt{\{}Alice\texttt{:!\}}} \rightarrow \texttt{bool}_{\texttt{\{!}\top\texttt{\}}}$ 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 $[Alice \triangleright \texttt{declassify}]\ \texttt{bool}_{\texttt{\{}Bob\texttt{:!\}}} \rightarrow \texttt{bool}_{\texttt{\{!}\top\texttt{\}}}$ 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:

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

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 $\texttt{declassify}\ e\ t$ 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 principal 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. For example, $\vdash \texttt{bool}_{\texttt{\{}Alice\texttt{:!\}}} - \texttt{bool}_{\texttt{\{}Alice\texttt{:}Bob\texttt{!\}}} = \texttt{\{}Alice\texttt{\}}$, because *Alice*'s authority is needed to add *Bob* as a reader. This judgment is used when typechecking the `declassify` expression:

$$\frac{\begin{array}{l}\Delta;\Gamma;\pi \vdash e : t_2\\ \Delta \vdash t_2 - t_1 = s\\ \Delta \vdash s \preceq \pi(\texttt{declassify})\end{array}}{\Delta;\Gamma;\pi \vdash \texttt{declassify}\, e\, t_1 : t_1}\ \text{(T-Dcls)}$$

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 $\Delta \vdash s \preceq \pi(\texttt{declassify})$ says that the set of `declassify`-granting principals in the static authority is sufficient to act for s.

For robustness, we must ensure that the integrity of the data is reflected in the set of static capabilities available. 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:

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

The restriction operator occurs in the typing rules of branching constructs. For example, this is the modified form of the `case` expression:

$$\frac{\begin{array}{l}\Delta;\Gamma;\pi_1 \vdash e : (t_1 + t_2)_l\\ \Delta;\Gamma;\pi_1|l \vdash v_1 : ([\pi_2]\ t_1 \to t)_l\\ \Delta;\Gamma;\pi_1|l \vdash v_2 : ([\pi_2]\ t_2 \to t)_l\\ \Delta \vdash \pi_2 \preceq (\pi_1|l)\end{array}}{\Delta;\Gamma;\pi_1 \vdash \texttt{case}\, e\ v_1\ v_2 : t \sqcup l}\ \text{(T-Case)}$$

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 (e_3 below):

$$\frac{\begin{array}{l}\Delta;\Gamma;\pi \vdash e_1 : \texttt{C}_l\\ \Delta;\Gamma;\pi \vdash e_2 : (\texttt{P}_p)_l\\ \Delta;\Gamma;(\pi, p \triangleright i)|l \vdash e_3 : t\\ \Delta;\Gamma;\pi|l \vdash e_4 : t\end{array}}{\Delta;\Gamma;\pi \vdash \texttt{if}\ (e_1 \Rightarrow e_2 \triangleright i)\ e_3\ e_4 : t \sqcup l}\ \text{(T-IfCert)}$$

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:

$$\frac{\mathcal{A} \vdash X_1\texttt{\{}i\texttt{\}} \Rightarrow X_2 \triangleright i}{\mathcal{A}, \texttt{if}\ (X_1\texttt{\{}i\texttt{\}} \Rightarrow X_2 \triangleright i)\ e_3\ e_4 \longrightarrow \mathcal{A}, e_3}\ \text{(E-CertYes)}$$

$$\frac{\mathcal{A} \vdash X_1\texttt{\{}i\texttt{\}} \not\Rightarrow X_2 \triangleright i}{\mathcal{A}, \texttt{if}\ (X_1\texttt{\{}i\texttt{\}} \Rightarrow X_2 \triangleright i)\ e_3\ e_4 \longrightarrow \mathcal{A}, e_4}\ \text{(E-CertNo)}$$

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: If $\mathcal{A} \vdash X_2 \preceq X_1$ then $\mathcal{A} \vdash X_1\texttt{\{}i\texttt{\}} \Rightarrow X_2 \triangleright i$.

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:

$$\frac{\Delta; \Gamma, x : t_1; \pi \vdash e : t_2 \quad \Delta \vdash l}{\Delta; \Gamma; \cdot \vdash \lambda x\!:\!t_1.\, e : ([\pi]\ t_1 \to t_2)_l} \text{ (T-Fun)}$$

$$\frac{\begin{array}{c}\Delta; \Gamma; \pi_1 \vdash e_1 : ([\pi_2]\ t_1 \to t_2)_l \\ \Delta; \Gamma; \pi_1 \vdash e_2 : t_1 \quad \Delta \vdash \pi_2 \preceq (\pi_1|l)\end{array}}{\Delta; \Gamma; \pi_1 \vdash e_1\ e_2 : t_2 \sqcup l} \text{ (T-App)}$$

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 4.2).

3.2 Delegation

Delegation allows the acts-for hierarchy to change during program execution—so far, the operational semantics have 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 p's permission.

We add a new expression $\texttt{let}\ (e_1 \preceq e_2)\ \texttt{in}\ e_3$ 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 $\texttt{delegate}_{p \preceq q}$. The constraint $\Delta \vdash p \preceq \pi(\texttt{delegate}_{p \preceq q})$ ensures that the capability to extend the acts-for hierarchy has been granted by p:

$$\frac{\begin{array}{l}\Delta; \Gamma; \pi \vdash e_1 : (\texttt{P}_p)_l \\ \Delta; \Gamma; \pi \vdash e_2 : (\texttt{P}_q)_l \\ \Delta, p \preceq q; \Gamma; \pi \vdash e_3 : t \\ \Delta \vdash p \preceq \pi(\texttt{delegate}_{p \preceq q})\end{array}}{\Delta; \Gamma; \pi \vdash \texttt{let}\ (e_1 \preceq e_2)\ \texttt{in}\ e_3 : t \sqcup l} \text{ (T-LetDel)}$$

As shown by the following evaluation rule E-LetDel, 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:

$$\frac{(\mathcal{A}, X_1 \preceq X_2), e_3 \longrightarrow (\mathcal{A}, X_1 \preceq X_2), e_3'}{\mathcal{A}, \texttt{let}\ (X_1 \preceq X_2)\ \texttt{in}\ e_3 \longrightarrow \mathcal{A}, \texttt{let}\ (X_1 \preceq X_2)\ \texttt{in}\ e_3'}$$

3.3 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 $\mathcal{E} \vdash X\texttt{\{}i\texttt{\}}$ to indicate that environment $\mathcal{E}$ produces the capability $X\texttt{\{}i\texttt{\}}$. Using the expression $\texttt{acquire}\ e \triangleright i$, 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 $X\texttt{\{}i\texttt{\}}$ or indicates failure by returning `*`. This behavior is captured by the following typechecking and evaluation rules (E-AcqNo, not shown, steps to $\texttt{inr}\ *$ when $\mathcal{E} \not\vdash X\texttt{\{}i\texttt{\}}$):

$$\frac{\Delta;\Gamma;\pi \vdash e : (\texttt{P}_p)_l}{\Delta;\Gamma;\pi \vdash \texttt{acquire}\ e \triangleright i : (\texttt{C}_l + \texttt{1}_l)_l}\ \text{(T-Acq)}$$

$$\frac{\mathcal{E} \vdash X\texttt{\{}i\texttt{\}}}{\mathcal{A}, \texttt{acquire}\ X \triangleright i \longrightarrow \mathcal{A}, \texttt{inl}\ X\texttt{\{}i\texttt{\}}}\ \text{(E-AcqYes)}$$

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 a 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 4.2).

3.4 Soundness

As a second theoretical contribution of this paper, we have extended the soundness result (Theorem 1) in Section 2 to the full language with authority and capability as follows. A complete proof can be found in Appendix.

Theorem 5 (Soundness). *(1) Progress: If $\mathcal{A};;\pi \vdash e : t$, then $e = v$ or $\mathcal{A}, e \longrightarrow \mathcal{A}, e'$. (2) Preservation: If $\mathcal{A};;\pi \vdash e : t$ and $\mathcal{A}, e \longrightarrow \mathcal{A}, e'$, then $\mathcal{A}';;\pi' \vdash e' : t$ such that $\mathcal{A} \preceq \mathcal{A}'$ and $\pi \preceq \pi'$.*

We have not proved a noninterference result for λ_{RP} with the run-time authority because we are primarily concerned with regulating declassification, which intentionally breaks noninterference. We conjecture that well-typed programs not containing `declassify` or delegation satisfy noninterference following a similar argument to that given in Section 2.4, but we leave the proof of this claim to future work.

4 PKI and application

4.1 Public key infrastructure

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 [15, 11]. We can interpret 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, where $\mathtt{K}_X$ is the public key corresponding to X and $\mathtt{K}_X^{-1}\texttt{\{}[\![i]\!]\texttt{\}}$ is a certificate signed using X's private key. The remaining constructs (the acts-for relation and the privileged operations) are interpreted as tuples:

$$\begin{array}{rcl}
[\![X]\!] & = & \mathtt{K}_X \\
[\![X_1 \preceq X_2]\!] & = & (\mathtt{K}_{X_1}, \mathtt{K}_{X_2}) \\
[\![X\texttt{\{}i\texttt{\}}]\!] & = & \mathtt{K}_X^{-1}\texttt{\{}[\![i]\!]\texttt{\}} \\
[\![\mathtt{declassify}]\!] & = & \mathtt{dcls} \\
[\![X \triangleright i]\!] & = & (\mathtt{K}_X, [\![i]\!]) \\
[\![\mathtt{delegate}_{X_1 \preceq X_2}]\!] & = & (\mathtt{del}, \mathtt{K}_{X_1}, \mathtt{K}_{X_2})
\end{array}$$

$$\frac{(\mathtt{K}_{X_2}, \mathtt{K}_{X_1}) \in [\![\mathcal{A}]\!]^*}{\mathcal{A} \vdash \mathtt{K}_{X_1}^{-1}\texttt{\{}[\![i]\!]\texttt{\}} \Rightarrow (\mathtt{K}_{X_2}, [\![i]\!])}$$

The interpretation of the acts-for hierarchy, $[\![\mathcal{A}]\!]^*$, is a binary relation on public keys—the reflexive, transitive closure of the pointwise 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: $\mathtt{verify}\ \mathtt{K}_{X_1}\ \mathtt{K}_{X_1}^{-1}\texttt{\{}[\![i]\!]\texttt{\}} = [\![i]\!]$. Note that in case of reflexive acts-for, we have $\mathtt{K}_{X_1} = \mathtt{K}_{X_2}$ and $\mathtt{K}_{X_1}^{-1}\texttt{\{}[\![i]\!]\texttt{\}} \Rightarrow (\mathtt{K}_{X_1}, [\![i]\!])$. 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 $[\![\mathcal{A}]\!]^*$ implies the existence of a valid certificate chain.

Now the universally trusted host $\top$ 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 $\mathtt{K}_X^{-1}\texttt{\{}[\![X \preceq \top]\!]\texttt{\}}$ and $(X, \top) \in [\![\mathcal{A}]\!]$ for each X—this information would be acquired by a host when it receives the principal X to key $\mathtt{K}_X$ binding from the certificate authority.

This interpretation permits flexibility in specifying security policies. Consider the following program that takes in two capabilities and some data owned by Alice and attempts to declassify it.

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

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

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

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 $(X, \top)$ pairs. Line 2 uses $[\![Alice]\!] = \mathtt{K}_{Alice}$ to verify the certificate $\mathcal{A} \vdash c_1 \Rightarrow (\mathtt{K}_{Alice}, [\![i]\!])$ where $[\![i]\!] = [\![Alice \triangleright \mathtt{delegate}_{Alice \preceq Bob}]\!] = (\mathtt{del}, \mathtt{K}_{Alice}, \mathtt{K}_{Bob})$. Since the acts-for hierarchy is otherwise empty, c_1 must be of the form $\mathtt{K}_{Alice}^{-1}\texttt{\{}[\![i]\!]\texttt{\}}$ or $\mathtt{K}_{\top}^{-1}\texttt{\{}[\![i]\!]\texttt{\}}$. The first certificate can be validated using only $\mathtt{K}_{Alice}$; the second can be validated starting from $\mathtt{K}_{Alice}$ by checking the certificate chain $\mathtt{K}^{-1}_{Alice}\mathtt{\{}\llbracket Alice \preceq \top\rrbracket\mathtt{\}} \leftrightarrow \mathtt{K}^{-1}_{\top}\mathtt{\{}\llbracket i\rrbracket\mathtt{\}}$. If one of these chains is valid, line 3 adds the delegation information into the hierarchy so that $(\mathtt{K}_{Alice}, \mathtt{K}_{Bob}) \in \llbracket \mathcal{A} \rrbracket$.

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

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

required by rule T-Dcls. If $c_2 = \mathtt{K}^{-1}_{Bob}\mathtt{\{dcls\}}$, the static condition holds at runtime because we can find the chain:

$$\mathtt{K}^{-1}_{Alice}\mathtt{\{}\llbracket Alice \preceq Bob\rrbracket\mathtt{\}} \leftrightarrow \mathtt{K}^{-1}_{Bob}\mathtt{\{dcls\}}$$

If $c_2 = \mathtt{K}^{-1}_{\top}\mathtt{\{dcls\}}$ we can find the chain:

$$\mathtt{K}^{-1}_{Alice}\mathtt{\{}\llbracket Alice \preceq Bob\rrbracket\mathtt{\}} \leftrightarrow \mathtt{K}^{-1}_{Bob}\mathtt{\{}\llbracket Bob \preceq \top\rrbracket\mathtt{\}} \leftrightarrow \mathtt{K}^{-1}_{\top}\mathtt{\{dcls\}}$$

In general, the justification for constraint $p_1 \preceq \pi(i)$ is the existence of some certificate chain of the form:

$$\mathtt{K}^{-1}_{p_1}\mathtt{\{}\llbracket p_1 \preceq p_2\rrbracket\mathtt{\}} \leftrightarrow \ldots \leftrightarrow \mathtt{K}^{-1}_{p_{n-1}}\mathtt{\{}\llbracket p_{n-1} \preceq p_n\rrbracket\mathtt{\}} \leftrightarrow \mathtt{K}^{-1}_{p_n}\mathtt{\{}\llbracket i\rrbracket\mathtt{\}}$$

4.2 Application to distributed banking

Figure 5 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 that are not in λ_{RP}, but could readily be added or encoded [23]. 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 left is the client code for *ATM*$_j$ (a particular ATM), on the right 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 $\mathtt{\{}Bank\mathtt{:}Bank\mathtt{!\}}$. 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. This example makes use of fine-grained `declassify` privileges to distinguish between the printing and network send 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, $\mathtt{\{}Bank\mathtt{:}Bank\mathtt{!\}}$ vs. $\mathtt{\{}agent\mathtt{:}agent\mathtt{!}agent\mathtt{\}}$ in the type of *request*.

Note 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 $[ATM_j \triangleright \mathtt{declassify}_{net}]$ and $[Bank \triangleright \mathtt{declassify}_{net}]$ indicate that the authorities to declassify to the network must be established from the caller.

$$
\begin{array}{rcl}
ATM_j\_main & : & [ATM_j \rhd \texttt{declassify}_{net}]\texttt{1} \to \texttt{1} \\
Bank\_main & : & [Bank \rhd \texttt{declassify}_{net}]\texttt{1} \to \texttt{1} \\
request & : & \forall(agent, user).\ (\texttt{P}_{agent}, \texttt{P}_{user}, \texttt{C}, \texttt{C})_{\{Bank:Bank!\}} \to \texttt{int}_{\{agent:agent!agent\}} \\
listen & : & \texttt{1} \to \exists(agent, user).\ (\texttt{P}_{agent}, \texttt{P}_{user}, \texttt{C}, \texttt{C}, (\texttt{int}_{\{agent:agent!\}} \to \texttt{1}))_{\{Bank:Bank!Bank\}} \\
login & : & \texttt{1} \to (\exists user.\ \texttt{P}_{user}, \texttt{C})_{\{ATM_j:ATM_j!\}} \\
print & : & \texttt{int\{!\}} \to \texttt{1} \\
get & : & \forall user.\ \texttt{P}_{user} \to \texttt{int}_{\{Bank:Bank!\}} \\
set & : & \forall user.\ \texttt{P}_{user} \to \texttt{int} \to \texttt{1}
\end{array}
$$

```
$ATM_j\_main = \lambda x : \texttt{1}.$
  let $[user, (user_{id}, c_{del})] = login$ * in
  case (acquire $user_{id} \rhd$ withdraw$_{100}$)
    $\lambda c_{req}$ : C. let $message = [(agent, user),$
        $(ATM_j, user_{id}, c_{del}, c_{req})]$ in
      let $data$ = declassify$_{net}$ $message$
        $(\texttt{P}_{ATM_j}, \texttt{P}_{user}, \texttt{C}, \texttt{C})_{\{Bank:Bank!\}}$ in
      let $balance = request\ [ATM_j, user]\ data$ in
      case (acquire $user_{id} \rhd$ declassify$_{prt}$)
        $\lambda c_{prt}$ : C. if ($c_{prt} \Rightarrow user_{id} \rhd$ declassify$_{prt}$)
          let $data$ = declassify$_{prt}$ $balance$ int$_{\{!\}}$ in
          $print\ data$
  $\cdots$   // other banking options
```

```
$Bank\_main = \lambda x : \texttt{1}.$
  let $[(agent, user), (agent_{id}, user_{id},$
    $c_{del}, c_{req}, reply)] = listen$ * in
  if ($c_{del} \Rightarrow user_{id} \rhd$ delegate$_{user \preceq agent}$)
    let $(user_{id} \preceq agent_{id})$ in
    if ($c_{del} \Rightarrow user_{id} \rhd$ withdraw$_{100}$)
      let $old = get\ [user]\ user_{id}$ in
      let $balance = old - 100$ in
      $set\ [user]\ user_{id}\ balance$;
      let $data$ = declassify$_{net}$
        $balance$ int$_{\{user:user!\}}$ in
      $reply\ data$
  $\cdots$   // other banking options
```

Figure 5: A distributed banking example

5 Discussion

5.1 Related work

The work nearest to ours is the Jif project, by Myers et al. [20]. Although the Jif compiler supports run-time principals, its type system has not been shown to be sound. Our noninterference proof for λ_{RP} is a step in that direction. Jif also supports *run-time labels*—run-time representations of the 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.

Although 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 $\texttt{P}_p$ to tie run-time tests to static types follows the work by Crary, Weirich, and Morrisett [9]. Static capability sets π in our type system are a form of *effects* [17], which have also been used to regulate the read and write privileges in type systems for memory management [8].

The robustness condition on the set of run-time capabilities is very closely related to Java's stack inspection model [33, 32, 10, 26]. In particular, the *enable-privilege* operation corresponds to our `if` $(e_1 \Rightarrow e_2 \rhd i)\ e_3\ e_4$ 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 the 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 [10]. 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 [7] 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 [6]. 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 infrastructure 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 [15, 11].

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. [3]. Adopting the local names of the SDSI/SPKI framework [1] may extend the analogy even further.

Lastly, although capabilities 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 the work by Jim and Gunter [16, 13], but we leave such pursuits to future work.

5.2 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 soundly 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 infrastructure.

Acknowledgments The authors thank Steve Chong, Dimitrios Vytiniotis, Geoffrey Washburn, Stephanie Weirich and anonymous referees for their helpful suggestions and comments on earlier drafts of this work.

References

[1] M. Abadi. On SDSI's linked local name spaces. *Journal of Computer Security*, 6(1-2):3–21, 1998.

[2] M. Abadi, A. Banerjee, N. Heintze, and J. Riecke. A core calculus of dependency. In *Proc. 26th ACM Symp. on Principles of Programming Languages (POPL)*, pages 147–160, San Antonio, TX, Jan. 1999.

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

[4] J. Agat. Transforming out timing leaks. In *Proc. 27th ACM Symp. on Principles of Programming Languages (POPL)*, pages 40–53, Boston, MA, Jan. 2000.

[5] D. Aspinall. Subtyping with Singleton Types. In *Computer Science Logic*, 1994.

[6] A. Banerjee and D. A. Naumann. Secure information flow and pointer confinement in a java-like language. In *Proc. of the 15th IEEE Computer Security Foundations Workshop*, 2002.

[7] A. Banerjee and D. A. Naumann. 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, June 2003.

[8] K. Crary, D. Walker, and G. Morrisett. Typed memory management in a calculus of capabilities. In *Proc. 26th ACM Symp. on Principles of Programming Languages (POPL)*, pages 262–275, San Antonio, Texas, Jan. 1999.

[9] K. Crary, S. Weirich, and G. Morrisett. Intensional polymorphism in type erasure semantics. *Journal of Functional Programming*, 12(6):567–600, Nov. 2002.

[10] C. Fournet and A. Gordon. Stack inspection: Theory and variants. In *Proc. 29th ACM Symp. on Principles of Programming Languages (POPL)*, pages 307–318, 2002.

[11] M. Gasser and E. McDermott. An architecture for practical delegation in a distributed system. In *Proc. IEEE Symposium on Security and Privacy*, pages 20–30. IEEE Computer Society Press, 1990.

[12] J. A. Goguen and J. Meseguer. Security policies and security models. In *Proc. IEEE Symposium on Security and Privacy*, pages 11–20. IEEE Computer Society Press, Apr. 1982.

[13] C. A. Gunter and T. Jim. Generalized certificate revocation. In *Proc. 27th ACM Symp. on Principles of Programming Languages (POPL)*, pages 316–329, Boston, Massachusetts, Jan. 2000. ACM Press.

[14] N. Heintze and J. G. Riecke. The SLam calculus: Programming with secrecy and integrity. In *Proc. 25th ACM Symp. on Principles of Programming Languages (POPL)*, pages 365–377, San Diego, California, Jan. 1998.

[15] J. Howell and D. Kotz. End-to-end authorization. In *Proc. USENIX Symp. on Operating Systems Design and Implementation (OSDI)*, pages 151–164, 2000.

[16] T. Jim. SD3: a trust management system with certificate revocation. In *IEEE Symposium on Security and Privacy*, pages 106–115, 2001.

[17] P. Jouvelot and D. K. Gifford. Algebraic reconstruction of types and effects. In *ACM Symposium on Principles of Programming Languages*, pages 303–310, Jan. 1991.

[18] P. Li, Y. Mao, and S. Zdancewic. Information integrity policies. In *Proceedings of the Workshop on Formal Aspects in Security & Trust (FAST)*, Sept. 2003.

[19] J. C. Mitchell. *Foundations for Programming Languages.* Foundations of Computing Series. The MIT Press, 1996.

[20] A. C. Myers, S. Chong, N. Nystrom, L. Zheng, and S. Zdancewic. Jif: Java information flow. Software release. Located at http://www.cs.cornell.edu/jif.

[21] A. C. Myers and B. Liskov. Complete, safe information flow with decentralized labels. In *Proc. IEEE Symposium on Security and Privacy*, pages 186–197, Oakland, CA, USA, May 1998.

[22] A. C. Myers and B. Liskov. Protecting privacy using the decentralized label model. *ACM Transactions on Software Engineering and Methodology*, 9(4):410–442, 2000.

[23] B. C. Pierce. *Types and Programming Languages.* MIT Press, 2002.

[24] F. Pottier and S. Conchon. Information flow inference for free. In *Proc. 5th ACM SIGPLAN International Conference on Functional Programming (ICFP)*, pages 46–57, Sept. 2000.

[25] F. Pottier and V. Simonet. Information flow inference for ML. In *Proc. 29th ACM Symp. on Principles of Programming Languages (POPL)*, Portland, Oregon, Jan. 2002.

[26] F. Pottier, C. Skalka, and S. F. Smith. A Systematic Approach to Static Access Control. In *European Symposium on Programming*, 2001.

[27] A. Sabelfeld and A. C. Myers. Language-based information-flow security. *IEEE Journal on Selected Areas in Communications*, 21(1):5–19, Jan. 2003.

[28] A. Sabelfeld and D. Sands. A PER model of secure information flow in sequential programs. *Higher-Order and Symbolic Computation*, 14(1):59–91, Mar. 2001.

[29] V. Simonet. Flow caml in a nutshell. In G. Hutton, editor, *Proceedings of the first APPSEM-II workshop*, pages 152–165, Mar. 2003.

[30] S. Tse and S. Zdancewic. Run-time principals in information-flow type systems. Technical Report MS-CIS-03-39, University of Pennsylvania, 2004.

[31] D. Volpano, G. Smith, and C. Irvine. A sound type system for secure flow analysis. *Journal of Computer Security*, 4(3):167–187, 1996.

[32] D. S. Wallach, A. W. Appel, and E. W. Felten. 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. 2000.

[33] D. S. Wallach and E. W. Felten. Understanding Java stack inspection. In *Proc. IEEE Symposium on Security and Privacy*, Oakland, California, USA, May 1998.

[34] S. Zdancewic. 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, Mar. 2003.

[35] S. Zdancewic and A. C. Myers. Secure information flow and CPS. In *Proc. of the 10th European Symposium on Programming*, volume 2028 of *Lecture Notes in Computer Science*, pages 46–61, Apr. 2001.

[36] S. Zdancewic and A. C. Myers. Secure information flow via linear continuations. *Higher Order and Symbolic Computation*, 15(2/3), 2002.

A Syntax

A secure type consists of a base type and a label. A function type $[\pi]\ t \rightarrow t$ is annotated with authority π as a program counter that keeps track of a privileged operations i (such as declassify and delegate) granted by principal p. $\mathtt{P}_p$ is a singleton type for a dynamic principal p. A capability $X\mathtt{\{}i\mathtt{\}}$ of type $\mathtt{C}$ represents a digitally signed certificate of a principal name X granting a privileged operation i.

t	$::=$	u_l	Secure types
u	$::=$		Base types
		$\mathtt{1}$	unit
		$t + t$	sum
		$[\pi]\ t \to t$	function
		$\mathtt{P}_p$	principal
		$\mathtt{C}$	capability
		$\forall \alpha \preceq p.\ t$	universal
v	$::=$		Values
		$\mathtt{*}$	unit
		$\mathtt{inl}\ v$	left injection
		$\mathtt{inr}\ v$	right injection
		$\lambda x{:}t.\ e$	function
		X	principal constant
		$X\mathtt{\{}i\mathtt{\}}$	capability
		$\Lambda \alpha \preceq p.\ e$	polymorphism
e	$::=$		Terms
		v	value
		x	variable
		$\mathtt{inl}\ e$	left injection
		$\mathtt{inr}\ e$	right injection
		$\mathtt{case}\ e\ v\ v$	sum case
		$e\ e$	application
		$\mathtt{if}\ (e \preceq e)\ e\ e$	if delegate
		$\mathtt{let}\ (e \preceq e)\ \mathtt{in}\ e$	let delegate
		$\mathtt{if}\ (e \Rightarrow e \triangleright i)\ e\ e$	if certify
		$\mathtt{declassify}\ e\ t$	declassify
		$\mathtt{acquire}\ e \triangleright i$	acquire
		$e\ [p]$	instantiation

p	$::=$		Principals
		α	variable
		X	name
s	$::=$	$\cdot \mid p, s$	Principal sets
c	$::=$	$\cdot \mid p :\ s$	Policies
d	$::=$	$\cdot \mid c;\, d$	Policy sets
l	$::=$	$\texttt{\{}d\texttt{!}s\texttt{\}}$	Labels
Δ	$::=$	$\cdot \mid \Delta, p \preceq p$	Principal environments
$\mathcal{A}$	$::=$	$\cdot \mid \mathcal{A}, X \preceq X$	Acts-for hierarchies
Γ	$::=$	$\cdot \mid \Gamma, x : t$	Term environments
π	$::=$	$\cdot \mid \pi, p \triangleright i$	Authority
δ	$::=$	$\cdot \mid \delta, \alpha \mapsto X$	Principal substitutions
γ	$::=$	$\cdot \mid \gamma, x \mapsto v$	Term substitutions
i	$::=$		Privileges
		$\texttt{declassify}$	declassify
		$\texttt{delegate}_{p \preceq p}$	delegate

B Static semantics

B.1 Typing

A label is well-formed $\Delta \vdash l$ if Δ contains all free principal variables of the label. During function applications in T-Case and T-App, we check the precondition of the program counter is satisfied: $\vdash \pi_2 \preceq (\pi_1|l)$. We also restrict the program counter to the integrity label of the branch condition $\Delta; \Gamma; \pi_1|l$ such that privileged operations are robust. $\Delta \vdash t_2 - t_1 = s$ computes the declassification requisite which is the set of principals whose authorities are needed in order to declassify.

$$\boxed{\Delta; \Gamma; \pi \vdash e : t} \qquad \boxed{\text{Typing}}$$

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

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

$$\frac{\Delta; \Gamma; \pi \vdash e : t_1 \quad \Delta \vdash l}{\Delta; \Gamma; \pi \vdash \mathtt{inl}\ e : (t_1 + t_2)_l} \qquad \text{T-Inl}$$

$$\frac{\Delta; \Gamma; \pi \vdash e : t_2 \quad \Delta \vdash l}{\Delta; \Gamma; \pi \vdash \mathtt{inr}\ e : (t_1 + t_2)_l} \qquad \text{T-Inr}$$

$$\frac{\begin{array}{ll} \Delta; \Gamma; \pi_1 \vdash e : (t_1 + t_2)_l & \Delta; \Gamma; \pi_1|l \vdash v_1 : ([\pi_2]\ t_1 \to t)_l \\ \Delta \vdash \pi_2 \preceq (\pi_1|l) & \Delta; \Gamma; \pi_1|l \vdash v_2 : ([\pi_2]\ t_2 \to t)_l \end{array}}{\Delta; \Gamma; \pi_1 \vdash \mathtt{case}\ e\ v_1\ v_2 : t \sqcup l} \qquad \text{T-Case}$$

$$\frac{\Delta; \Gamma, x : t_1; \pi \vdash e : t_2 \quad \Delta \vdash l}{\Delta; \Gamma; \cdot \vdash \lambda x{:}t_1.\ e : ([\pi]\ t_1 \to t_2)_l} \qquad \text{T-Fun}$$

$$\frac{\Delta; \Gamma; \pi_1 \vdash e_1 : ([\pi_2]\ t_1 \to t_2)_l \quad \Delta; \Gamma; \pi_1 \vdash e_2 : t_1 \quad \Delta \vdash \pi_2 \preceq (\pi_1|l)}{\Delta; \Gamma; \pi_1 \vdash e_1\ e_2 : t_2 \sqcup l} \qquad \text{T-App}$$

$$\frac{\Delta \vdash l}{\Delta; \Gamma; \pi \vdash X : (\mathtt{P}_X)_l} \qquad \text{T-PName}$$

$$\frac{\Delta \vdash l}{\Delta; \Gamma; \pi \vdash X\mathtt{\{}i\mathtt{\}} : \mathtt{C}_l} \qquad \text{T-Cap}$$

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

$$\frac{\Delta; \Gamma; \pi \vdash e_1 : (\mathtt{P}_{p_1})_l \quad \Delta; \Gamma; \pi \vdash e_2 : (\mathtt{P}_{p_2})_l \quad \Delta, p_1 \preceq p_2; \Gamma; \pi \vdash e_3 : t \quad \Delta \vdash p_1 \preceq \pi(\mathtt{delegate}_{p_1 \preceq p_2})}{\Delta; \Gamma; \pi \vdash \mathtt{let}\ (e_1 \preceq e_2)\ \mathtt{in}\ e_3 : t \sqcup l} \qquad \text{T-LetDel}$$

$$\frac{\Delta; \Gamma; \pi \vdash e_1 : \mathtt{C}_l \quad \Delta; \Gamma; \pi \vdash e_2 : (\mathtt{P}_p)_l \quad \Delta; \Gamma; (\pi, p \rhd i)|l \vdash e_3 : t \quad \Delta; \Gamma; \pi|l \vdash e_4 : t}{\Delta; \Gamma; \pi \vdash \mathtt{if}\ (e_1 \Rightarrow e_2 \rhd i)\ e_3\ e_4 : t \sqcup l} \qquad \text{T-IfCert}$$

$$\frac{\Delta; \Gamma; \pi \vdash e : t_2 \quad \Delta \vdash t_2 - t_1 = s \quad \Delta \vdash s \preceq \pi(\mathtt{declassify})}{\Delta; \Gamma; \pi \vdash \mathtt{declassify}\ e\ t_1 : t_1} \qquad \text{T-Dcls}$$

$$\frac{\Delta; \Gamma; \pi \vdash e : (\mathtt{P}_p)_l}{\Delta; \Gamma; \pi \vdash \mathtt{acquire}\ e \rhd i : (\mathtt{C}_l + \mathtt{1}_l)_l} \qquad \text{T-Acq}$$

$$\frac{\Delta, \alpha \preceq p; \Gamma; \pi \vdash e : t \quad \alpha \notin \mathrm{dom}(\Delta) \quad \Delta \vdash l}{\Delta; \Gamma; \pi \vdash \Lambda\alpha \preceq p.\, e : (\forall \alpha \preceq p.\, t)_l} \qquad \text{T-All}$$

$$\frac{\Delta; \Gamma; \pi \vdash e : (\forall \alpha \preceq p_2.\, t)_l \quad \Delta \vdash p_1 \preceq p_2}{\Delta; \Gamma; \pi \vdash e\ [p_1] : t \sqcup l} \qquad \text{T-PApp}$$

$$\frac{\Delta; \Gamma; \pi \vdash e : t_1 \quad \Delta \vdash t_1 \leq t_2}{\Delta; \Gamma; \pi \vdash e : t_2} \qquad \text{T-Sub}$$

B.2 Substitution

Convention 6 (Capture-avoiding substitution). *When we write $\gamma(\lambda x{:}t.\, e)$, it is assumed that $x \notin$ dom(γ). When we write $\delta(\forall \alpha \preceq p.\, t)$ or $\delta(\Lambda\alpha \preceq p.\, e)$, it is assumed that $\alpha \notin$ dom(δ).*

We define $e^\bullet \overset{\text{def}}{=} \gamma\delta(e)$. We define $\gamma(x) = v$ if $x \mapsto v \in \gamma$, and $\gamma(x) = x$ otherwise. Similarly, we define $\delta(\alpha) = X$ if $\alpha \mapsto X \in \delta$, and $\delta(\alpha) = \alpha$ otherwise. The $\diamond$ can be instantiated with any of these syntactic categories: t, u, e, v, p, s, c and d, l, Δ and Γ. (It cannot be instantiated with $\mathcal{A}$, δ or γ.)

$\boxed{\diamond^\bullet = \diamond}$	$\boxed{\text{Substitution}}$
$(\delta, \alpha \mapsto X)(\diamond) = \delta(\diamond\{X/\alpha\})$	Su-Delta
$(\gamma, x \mapsto v)(e) = \gamma(e\{v/x\})$	Su-Gamma
$\cdot^\bullet = \cdot$	Su-Empty
$(u_l)^\bullet = u^\bullet{}_{l^\bullet}$	Tsu-UL
$(t_1 + t_2)^\bullet = t_1^\bullet + t_2^\bullet$	Tsu-Sum
$(t_1 \rightarrow t_2)^\bullet = t_1^\bullet \rightarrow t_2^\bullet$	Tsu-Fun
$(\texttt{P}_p)^\bullet = \texttt{P}_{p^\bullet}$	Tsu-PName
$(\texttt{C})^\bullet = \texttt{C}$	Tsu-Cap
$(\forall \alpha \preceq p.\ t)^\bullet = \forall \alpha \preceq p^\bullet.\ t^\bullet$	Tsu-All
$\texttt{1}^\bullet = \texttt{1}$	Tsu-Unit
$\texttt{*}^\bullet = \texttt{*}$	Esu-Unit
$x^\bullet = \gamma(x)$	Esu-Var
$(\texttt{inl}\ e)^\bullet = \texttt{inl}\ e^\bullet$	Esu-Inl
$(\texttt{inr}\ e)^\bullet = \texttt{inr}\ e^\bullet$	Esu-Inr
$(\texttt{case}\ e\ v_1\ v_2)^\bullet = \texttt{case}\ e^\bullet\ v_1^\bullet\ v_2^\bullet$	Esu-Case
$(\lambda x\!:\!t.\ e)^\bullet = \lambda x\!:\!t^\bullet.\ e^\bullet$	Esu-Fun
$(e_1\ e_2)^\bullet = e_1^\bullet\ e_2^\bullet$	Esu-App
$(X)^\bullet = X$	Esu-PName
$(X\texttt{\{}i\texttt{\}})^\bullet = X\texttt{\{}i^\bullet\texttt{\}}$	Esu-Cap

$$(\texttt{if}\ (e_1 \preceq e_2)\ e_3\ e_4)^\bullet = \texttt{if}\ (e_1^\bullet \preceq e_2^\bullet)\ e_3^\bullet\ e_4^\bullet \qquad \text{Esu-IfDel}$$

$$(\texttt{if}\ (e_1 \Rightarrow e_2 \triangleright e_3)\ e_4\ )^\bullet = \texttt{if}\ (e_1^\bullet \Rightarrow e_2^\bullet \triangleright e_3^\bullet)\ e_4^\bullet \qquad \text{Esu-IfDel}$$

$$(\texttt{let}\ (e_1 \preceq e_2)\ \texttt{in}\ e_3)^\bullet = \texttt{let}\ (e_1^\bullet \preceq e_2^\bullet)\ \texttt{in}\ e_3^\bullet \qquad \text{Esu-IfDel}$$

$$(\texttt{declassify}\ e\ t)^\bullet = \texttt{declassify}\ e^\bullet\ t^\bullet \qquad \text{Esu-IfDel}$$

$$(\texttt{acquire}\ e \triangleright i)^\bullet = \texttt{acquire}\ e^\bullet \triangleright i^\bullet \qquad \text{Esu-IfDel}$$

$$(\Lambda\alpha \preceq p.\ e)^\bullet = \Lambda\alpha \preceq p^\bullet.\ e^\bullet \qquad \text{Esu-All}$$

$$(e_1\ [p])^\bullet = e_1^\bullet\ [p^\bullet] \qquad \text{Esu-PApp}$$

$$\alpha^\bullet = \delta(\alpha) \qquad \text{Lsu-Var}$$

$$X^\bullet = X \qquad \text{Lsu-Name}$$

$$(p, s)^\bullet = p^\bullet, s^\bullet \qquad \text{Lsu-PSet}$$

$$(p:\ s)^\bullet = p^\bullet:\ s^\bullet \qquad \text{Lsu-Policy}$$

$$(c;\ d)^\bullet = c^\bullet;\ d^\bullet \qquad \text{Lsu-CSet}$$

$$\texttt{\{}d\,\texttt{!}\,s\texttt{\}}^\bullet = \texttt{\{}d^\bullet\,\texttt{!}\,s^\bullet\texttt{\}} \qquad \text{Lsu-Label}$$

$$(\Delta, p_1 \preceq p_2)^\bullet = \Delta^\bullet, p_1^\bullet \preceq p_2^\bullet \qquad \text{Su-PEnv}$$

$$(\Gamma, x : t)^\bullet = \Gamma^\bullet, x : t^\bullet \qquad \text{Su-EEnv}$$

$$(\pi, p : i)^\bullet = \pi^\bullet, p^\bullet : i^\bullet \qquad \text{Su-EEnv}$$

$$\texttt{declassify}^\bullet = \texttt{declassify} \qquad \text{Su-Dcls}$$

$$(\texttt{delegate}_{p_1 \preceq p_2})^\bullet = \texttt{delegate}_{p_1^\bullet \preceq p_2^\bullet} \qquad \text{Su-Del}$$

B.3 Subtyping

Note that we disallow subtyping with singleton principal–allowing it will break type soundness and noninterference.

$$\boxed{\Delta \vdash p \preceq p} \qquad \boxed{\text{Acts-for}}$$

$$\Delta \vdash p \preceq p \qquad \text{A-Refl}$$

$$\frac{\Delta \vdash p_1 \preceq p_2 \quad \Delta \vdash p_2 \preceq p_3}{\Delta \vdash p_1 \preceq p_3} \qquad \text{A-Trans}$$

$$\Delta \vdash p \preceq \top \qquad \text{A-Top}$$

$$\frac{p_1 \preceq p_2 \in \Delta}{\Delta \vdash p_1 \preceq p_2} \qquad \text{A-Constr}$$

$$\frac{fv(l) \subseteq \mathrm{dom}(\Delta)}{\Delta \vdash l} \qquad \boxed{\text{Ok-Label}}$$

$$u_l \sqcup l' = u_{l \,\sqcup\, l'} \qquad \boxed{\text{J-UL}}$$

$$\frac{\Delta \vdash u \leq u' \quad \Delta \vdash l \sqsubseteq l'}{\Delta \vdash u_l \leq {u'}_{l'}} \qquad \boxed{\text{St-UL}}$$

$$\boxed{\Delta \vdash u \leq u} \qquad \boxed{\text{Base Subtyping}}$$

$$\Delta \vdash u \leq u \qquad \text{St-Refl}$$

$$\frac{\Delta \vdash u \leq u' \quad \Delta \vdash u' \leq u''}{\Delta \vdash u \leq u''} \qquad \text{St-Trans}$$

$$\frac{\Delta \vdash t_1 \leq t_1' \quad \Delta \vdash t_2 \leq t_2'}{\Delta \vdash (t_1 + t_2) \leq (t_1' + t_2')} \qquad \text{St-Sum}$$

$$\frac{\Delta \vdash t_1' \leq t_1 \quad \Delta \vdash t_2 \leq t_2'}{\Delta \vdash (t_1 \to t_2) \leq (t_1' \to t_2')} \qquad \text{St-Fun}$$

$$\frac{\Delta \vdash p' \preceq p \quad \Delta, \alpha \preceq p' \vdash t \leq t'}{\Delta \vdash (\forall \alpha \preceq p.\, t) \leq (\forall \alpha \preceq p'.\, t')} \qquad \text{St-All}$$

B.4 Declassification and models

$\Delta \vdash t_2 - t_1 = s$ computes the declassification requisite which is the set of principals whose authorities are needed in order to declassify.

The notation $\delta \models \Delta$ denotes a substitution δ that assigns each free principal variable α in hierarchy Δ to a principal name X. Similarly, $\mathcal{A} \vdash \gamma \models \delta(\Gamma)$ denotes a term substitution γ that assigns each free term variable x in environment Γ to a value such that the assignment respects the typing $x : t$ in Γ.

$$\boxed{\Delta \vdash t - t = s} \qquad \boxed{\text{Declassification requisite}}$$

$$\frac{\Delta \vdash u - u' = s_1 \quad \Delta \vdash l - l' = s_2}{\Delta \vdash u_l - {u'}_{l'} = s_1 \cup s_2} \qquad \text{D-UL}$$

$$\Delta \vdash \mathtt{1} - \mathtt{1} = \cdot \qquad \text{D-Unit}$$

$$\frac{\Delta \vdash t_1 - t_1' = s_1 \quad \Delta \vdash t_2 - t_2' = s_2}{\Delta \vdash (t_1 + t_2) - (t_1' + t_2') = s_1 \cup s_2} \qquad \text{D-Sum}$$

$$\frac{\Delta \vdash t_1' - t_1 = s_1 \quad \Delta \vdash t_2 - t_2' = s_2}{\Delta \vdash (t_1 \to t_2) - (t_1' \to t_2') = s_1 \cup s_2} \qquad \text{D-Fun}$$

$$\Delta \vdash \mathtt{P}_p - \mathtt{P}_p = \cdot \qquad \text{D-PName}$$

$$\frac{\Delta, \alpha \preceq p \vdash t - t' = s}{\Delta \vdash (\forall \alpha \preceq p.\ t) - (\forall \alpha \preceq p.\ t') = s} \qquad \text{D-All}$$

$$\frac{s' = \{p \;\mid\; \Delta \vdash d_2(p) \preceq d_1(p),\ \Delta \vdash d_1(p) \not\preceq d_2(p)\}}{\Delta \vdash \mathtt{\{}d_1\,\mathtt{!}\,s\mathtt{\}} - \mathtt{\{}d_2\,\mathtt{!}\,s\mathtt{\}} = s'} \qquad \text{D-Label}$$

$$\boxed{\Delta \vdash \gamma \models \Gamma} \qquad \boxed{\text{ESubs model}}$$

$$\Delta \vdash \cdot \models \cdot \qquad \text{Em-Nil}$$

$$\frac{\Delta \vdash \gamma \models \Gamma \quad \Delta; \Gamma; \pi \vdash v : t}{\Delta \vdash \gamma, x \mapsto v \models \Gamma, x : t} \qquad \text{Em-Cons}$$

$$\boxed{\delta \models \Delta} \qquad \boxed{\text{PSubs model}}$$

$$\cdot \models \cdot \qquad \text{Pm-Nil}$$

$$\frac{\delta \models \Delta \quad \Delta \vdash p_1 \preceq p_2}{\delta, \alpha \mapsto p_1 \models \Delta, \alpha \preceq p_2} \qquad \text{Pm-Var}$$

$$\frac{\delta \models \Delta}{\delta \models \Delta, X \preceq p_2} \qquad \text{Pm-Name}$$

$$\frac{fv(\Gamma) \subseteq \mathrm{dom}(\Delta)}{\Delta \vdash \Gamma} \qquad \boxed{\text{EEnv Ok}}$$

C Dynamic semantics

At run-time, E-IfDelYes checks if delegation from X_1 to X_2 is satisfiable in the acts-for hierarchy $\mathcal{A}$. E-LetDel adds delegation to the hierarchy. E-IfCertYes cryptographically verifies if a certificate is valid. E-AcqYes enquires the external environment $\mathcal{E}$ if a principal p is granting the privileged operation i.

$$\boxed{\mathcal{A}, e \longrightarrow \mathcal{A}, e} \qquad \boxed{\text{Evaluation}}$$

$$\frac{\mathcal{A}, e \longrightarrow \mathcal{A}, e'}{\mathcal{A}, \texttt{inl}\ e \longrightarrow \mathcal{A}, \texttt{inl}\ e'} \qquad \text{E-Inl}$$

$$\frac{\mathcal{A}, e \longrightarrow \mathcal{A}, e'}{\mathcal{A}, \texttt{inr}\ e \longrightarrow \mathcal{A}, \texttt{inr}\ e'} \qquad \text{E-Inr}$$

$$\frac{\mathcal{A}, e \longrightarrow \mathcal{A}, e'}{\mathcal{A}, \texttt{case}\ e\ v_1\ v_2 \longrightarrow \mathcal{A}, \texttt{case}\ e'\ v_1\ v_2} \qquad \text{E-Case}$$

$$\mathcal{A}, \texttt{case}\ (\texttt{inl}\ v)\ v_1\ v_2 \longrightarrow \mathcal{A}, v_1\ v \qquad \text{E-CaseInl}$$

$$\mathcal{A}, \texttt{case}\ (\texttt{inr}\ v)\ v_1\ v_2 \longrightarrow \mathcal{A}, v_2\ v \qquad \text{E-CaseInr}$$

$$\frac{\mathcal{A}, e_1 \longrightarrow \mathcal{A}, e_1'}{\mathcal{A}, e_1\ e_2 \longrightarrow \mathcal{A}, e_1'\ e_2} \qquad \text{E-App1}$$

$$\frac{\mathcal{A}, e \longrightarrow \mathcal{A}, e'}{\mathcal{A}, v\ e \longrightarrow \mathcal{A}, v\ e'} \qquad \text{E-App2}$$

$$\mathcal{A}, (\lambda x\!:\!t.\ e)\ v \longrightarrow \mathcal{A}, e\{v/x\} \qquad \text{E-AppFun}$$

$$\frac{\mathcal{A}, e_1 \longrightarrow \mathcal{A}, e_1'}{\mathcal{A}, \texttt{if}\ (e_1 \preceq e_2)\ e_3\ e_4 \longrightarrow \mathcal{A}, \texttt{if}\ (e_1' \preceq e_2)\ e_3\ e_4} \qquad \text{E-IfDel1}$$

$$\frac{\mathcal{A}, e_2 \longrightarrow \mathcal{A}, e_2'}{\mathcal{A}, \texttt{if}\ (v \preceq e_2)\ e_3\ e_4 \longrightarrow \mathcal{A}, \texttt{if}\ (v \preceq e_2')\ e_3\ e_4} \qquad \text{E-IfDel2}$$

$$\frac{\mathcal{A} \vdash X_1 \preceq X_2}{\mathcal{A}, \texttt{if}\ (X_1 \preceq X_2)\ e_3\ e_4 \longrightarrow \mathcal{A}, e_3} \qquad \text{E-IfDelYes}$$

$$\frac{\mathcal{A} \vdash X_1 \not\preceq X_2}{\mathcal{A}, \texttt{if}\ (X_1 \preceq X_2)\ e_3\ e_4 \longrightarrow \mathcal{A}, e_4} \qquad \text{E-IfDelNo}$$

$$\frac{\mathcal{A}, e_2 \longrightarrow \mathcal{A}, e_2'}{\mathcal{A}, \texttt{let}\ (v \preceq e_2)\ \texttt{in}\ e_3 \longrightarrow \mathcal{A}, \texttt{let}\ (v \preceq e_2')\ \texttt{in}\ e_3} \qquad \text{E-LetDel1}$$

$$\frac{\mathcal{A}, e_1 \longrightarrow \mathcal{A}, e_1'}{\mathcal{A}, \texttt{let}\ (e_1 \preceq e_2)\ \texttt{in}\ e_3 \longrightarrow \mathcal{A}, \texttt{let}\ (e_1' \preceq e_2)\ \texttt{in}\ e_3} \qquad \text{E-LetDel2}$$

$$\frac{(\mathcal{A}, X_1 \preceq X_2), e_3 \longrightarrow (\mathcal{A}, X_1 \preceq X_2), e_3'}{\mathcal{A}, \texttt{let}\ (X_1 \preceq X_2)\ \texttt{in}\ e_3 \longrightarrow \mathcal{A}, \texttt{let}\ (X_1 \preceq X_2)\ \texttt{in}\ e_3'} \qquad \text{E-LetDel}$$

$$\mathcal{A}, \texttt{let}\ (X_1 \preceq X_2)\ \texttt{in}\ v \longrightarrow \mathcal{A}, v \qquad \text{E-LetDelV}$$

$$\frac{\mathcal{A}, e_1 \longrightarrow \mathcal{A}, e_1'}{\mathcal{A}, \texttt{if}\ (e_1 \Rightarrow e_2 \triangleright i)\ e_3\ e_4 \longrightarrow \mathcal{A}, \texttt{if}\ (e_1' \Rightarrow e_2 \triangleright i)\ e_3\ e_4} \qquad \text{E-IfCert1}$$

$$\frac{\mathcal{A}, e_2 \longrightarrow \mathcal{A}, e_2'}{\mathcal{A}, \texttt{if}\ (v \Rightarrow e_2 \triangleright i)\ e_3\ e_4 \longrightarrow \mathcal{A}, \texttt{if}\ (v \Rightarrow e_2' \triangleright i)\ e_3\ e_4} \qquad \text{E-IfCert2}$$

$$\frac{\mathcal{A} \vdash X_1\{i\} \Rightarrow X_2 \triangleright i}{\mathcal{A}, \texttt{if}\ (X_1\{i\} \Rightarrow X_2 \triangleright i)\ e_3\ e_4 \longrightarrow \mathcal{A}, e_3} \qquad \text{E-IfCertYes}$$

$$\frac{\mathcal{A} \vdash X_1\{i\} \not\Rightarrow X_2 \triangleright i}{\mathcal{A}, \texttt{if}\ (X_1\{i\} \Rightarrow X_2 \triangleright i)\ e_3\ e_4 \longrightarrow \mathcal{A}, e_4} \qquad \text{E-IfCertNo}$$

$$\frac{\mathcal{A}, e \longrightarrow \mathcal{A}, e'}{\mathcal{A}, \texttt{declassify}\ e\ t \longrightarrow \mathcal{A}, e'} \qquad \text{E-Dcls1}$$

$$\mathcal{A}, \texttt{declassify}\ v\ t \longrightarrow \mathcal{A}, v \qquad \text{E-Dcls2}$$

$$\frac{\mathcal{A}, e \longrightarrow \mathcal{A}, e'}{\mathcal{A}, \texttt{acquire}\ e \triangleright i \longrightarrow \mathcal{A}, \texttt{acquire}\ e' \triangleright i} \qquad \text{E-Acq}$$

$$\frac{\mathcal{E} \vdash X\{i\}}{\mathcal{A}, \texttt{acquire}\ X \triangleright i \longrightarrow \mathcal{A}, \texttt{inl}\ X\{i\}} \qquad \text{E-AcqYes}$$

$$\frac{\mathcal{E} \not\vdash X\{i\}}{\mathcal{A}, \texttt{acquire}\ X \triangleright i \longrightarrow \mathcal{A}, \texttt{inr}\ \texttt{*}} \qquad \text{E-AcqNo}$$

$$\frac{\mathcal{A}, e \longrightarrow \mathcal{A}, e'}{\mathcal{A}, e\ [X] \longrightarrow \mathcal{A}, e'\ [X]} \qquad \text{E-PApp}$$

$$\mathcal{A}, (\Lambda\alpha \preceq p.\ e)\ [X] \longrightarrow \mathcal{A}, e\{X/\alpha\} \qquad \text{E-PAppAll}$$

$$\boxed{\mathcal{A}, e \longrightarrow^* \mathcal{A}, e} \qquad \boxed{\text{Multistep eval}}$$

$$\mathcal{A}, v \longrightarrow^* \mathcal{A}, v \qquad \text{EM-Refl}$$

$$\frac{\mathcal{A}, e \longrightarrow \mathcal{A}, e' \quad \mathcal{A}, e' \longrightarrow^* \mathcal{A}, e''}{\mathcal{A}, e \longrightarrow^* \mathcal{A}, e''} \qquad \text{EM-Trans}$$

D Theorems

This section proves the soundness of the language with respect to the operational semantics: progress and preservation theorems. The main lemma is substitution for typing and subtyping. Canonical forms, inversion and weakening are standard for languages with subtyping.

Lemma 7 (Canonical forms).

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

Theorem 8 (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:

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

  But $\cdot$ cannot contain $x : t$, and hence this case does not apply.

- T-Unit: $\dfrac{\vdash l}{\mathcal{A};;\pi \vdash \texttt{*} : \texttt{1}_l}$

  e is a value.

- T-Inl: $\dfrac{\mathcal{A};;\pi \vdash e_1 : t_1 \quad \vdash l}{\mathcal{A};;\pi \vdash \texttt{inl}\, e_1 : (t_1 + t_2)_l}$

  By IH on e_1,

  1. $e_1 = v$: $e = \texttt{inl}\, v$ is a value.
  2. $\mathcal{A}, e_1 \longrightarrow \mathcal{A}, e_1'$: by E-Inl, $e' = \texttt{inl}\, e_1'$.

- T-Inr: symmetric to T-Inl.

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

  By IH on e_0,

  1. $\mathcal{A}, e_0 \longrightarrow \mathcal{A}, e_0'$: by E-Case, $e' = \texttt{case}\; e_0'\; v_1\; v_2$.
  2. $e_0 = v$: by Lemma 7 (canonical forms),
     (a) $e_0 = \texttt{inl}\, v_0$: by E-CaseInl, $e' = v_1\; v_0$.
     (b) $e_0 = \texttt{inr}\, v_0$: by E-CaseInr, $e' = v_2\; v_0$.

- T-Fun: $\dfrac{\mathcal{A}; x : t_1; \vdash e_0 : t_2 \quad x \notin \mathrm{dom}(\cdot) \quad \vdash l}{\mathcal{A};;\pi \vdash \lambda x{:}t_1.\; e_0 : [\cdot]\; t_1 \to t_2}$

  e is a value.

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

  By IH on e_1 and e_2,

  1. $\mathcal{A}, e_1 \longrightarrow \mathcal{A}, e_1'$: by E-App1, $e' = e_1'\ e_2$.
  2. $e_1 = v$ and $\mathcal{A}, e_2 \longrightarrow \mathcal{A}, e_2'$: by E-App2, $e' = v\ e_2'$.
  3. $e_1 = v_1$ and $e_2 = v_2$: by Lemma 7 (canonical forms), $e_1 = \lambda x : t.\ 'e_0$ and then by E-AppFun, $e' = e_0\{v_2/x\}$.

- T-PName: $$\frac{\vdash l}{\mathcal{A};;\pi \vdash X : (\texttt{P}_X)_l}$$

  e is a value.

- T-Cap: $$\frac{\vdash l}{\mathcal{A};;\pi \vdash X\texttt{\{}i\texttt{\}} : \texttt{C}_l}$$

  e is a value.

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

  By IH on e_1 and e_2,

  1. $\mathcal{A}, e_1 \longrightarrow \mathcal{A}, e_1'$: by E-IfDel1, $e' = \texttt{if}\ (e_1' \preceq e_2)\ e_3\ e_4$.
  2. $e_1 = v$ and $\mathcal{A}, e_2 \longrightarrow \mathcal{A}, e_2'$: by E-IfDel2, $e' = \texttt{if}\ (v \preceq e_2')\ e_3\ e_4$.
  3. $e_1 = v_1$ and $e_2 = v_2$: by Lemma 7 (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 $\mathcal{A} \vdash X_1 \npreceq X_2$: by E-IfDelNo, $e' = e_4$.

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

  By IH on e_1 and e_2,

  1. $\mathcal{A}, e_1 \longrightarrow \mathcal{A}, e_1'$: by E-LetDel1, $e' = \texttt{let}\ (e_1' \preceq e_2)\ \texttt{in}\ e_3$.
  2. $e_1 = v$ and $\mathcal{A}, e_2 \longrightarrow \mathcal{A}, e_2'$: by E-LetDel2, $e' = \texttt{let}\ (v \preceq e_2')\ \texttt{in}\ e_3$.
  3. $e_1 = v_1$ and $e_2 = v_2$: by Lemma 7 (canonical forms), $e_1 = X_1$ and $e_2 = X_2$. Then, by IH on e_3, we have $e' = \texttt{let}\ (X_1 \preceq X_2)\ \texttt{in}\ e_3'$ (by E-LetDel), or we have $e' = v$ (by E-LetDelV).

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

  By IH on e_1 and e_2,

  1. $\mathcal{A}, e_1 \longrightarrow \mathcal{A}, e_1'$: by E-IfCert1, $e' = \texttt{if}\ (e_1' \Rightarrow e_2 \triangleright i)\ e_3\ e_4$.
  2. $e_1 = v$ and $\mathcal{A}, e_2 \longrightarrow \mathcal{A}, e_2'$: by E-IfCert2, $e' = \texttt{if}\ (v \Rightarrow e_2' \triangleright i)\ e_3\ e_4$.
  3. $e_1 = v_1$ and $e_2 = v_2$: by Lemma 7 (canonical forms),
     (a) $e_1 = X\texttt{\{}i\texttt{\}}$, $e_2 = X$ and $\mathcal{A} \vdash X_1\texttt{\{}i\texttt{\}} \Rightarrow X_2 \triangleright i$: by E-IfCertYes, $e' = e_3$.
     (b) $e_1 = X$, $e_2 = X$ and $\mathcal{A} \vdash X_1\texttt{\{}i\texttt{\}} \not\Rightarrow X_2 \triangleright i$: by E-IfCertNo, $e' = e_4$.

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

  By E-Dcls1 or E-Dcls2.

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

  By IH on e_0,

  1. $\mathcal{A}, e_0 \longrightarrow \mathcal{A}, e_0'$: by E-Acq, $e' = \mathtt{acquire}\ e_0' \triangleright i$.
  2. $e_0 = v$: by Lemma 7 (canonical forms), $e_0 = X\mathtt{\{}i\mathtt{\}}$. Then, either
     (a) $\mathcal{E} \vdash X\mathtt{\{}i\mathtt{\}}$: by E-AcqYes, $e' = \mathtt{inl}\ X\mathtt{\{}i\mathtt{\}}$.
     (b) $\mathcal{E} \not\vdash X\mathtt{\{}i\mathtt{\}}$: by E-AcqNo, $e' = \mathtt{inr}\ \mathtt{*}$.

- T-All: $\dfrac{\mathcal{A}, \alpha \preceq p \vdash e_0 : t_0 \quad \alpha \notin \mathrm{dom}(\mathcal{A}) \quad \vdash l}{\mathcal{A};;\pi \vdash \Lambda\alpha \preceq p.\ e_0 : (\forall\alpha \preceq p.\ t_0)_l}$

  e is a value.

- T-PApp: $\dfrac{\mathcal{A};;\pi \vdash e_0 : (\forall\alpha \preceq p.\ t_0)_l}{\mathcal{A};;\pi \vdash e_0\ [p] : t_0 \sqcup l}$

  By IH on e_0,

  1. $\mathcal{A}, e_0 \longrightarrow \mathcal{A}, e_0'$: by E-PApp, $e' = e_0'\ [p]$.
  2. $e_0 = v$: by Lemma 7 (canonical forms), $e_0 = \Lambda\alpha \preceq p.\ e_1$ and $p = X$. Then, by E-PAppAll, $e' = e_1\{X/\alpha\}$.

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

  By IH on e_0.

□

Lemma 9 (Inversion).

1. *If* $\Delta;\Gamma;\pi \vdash \mathtt{inl}\ v : (t_1 + t_2)_l$*, then* $\Delta;\Gamma;\pi \vdash v : t_1$.
2. *If* $\Delta;\Gamma;\pi \vdash \mathtt{inr}\ v : (t_1 + t_2)_l$*, then* $\Delta;\Gamma;\pi \vdash v : t_2$.
3. *If* $\Delta;\Gamma;\pi \vdash \lambda x{:}t_1.\ e : (t_1 \to t_2)_l$*, 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* $\Delta, \alpha \preceq p;\Gamma;\pi \vdash e : t$.

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

Lemma 10 (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* $\pi_1 \preceq \pi_2$ *and* $\pi_2 \preceq \pi_3$*, then* $\pi_1 \preceq \pi_3$.

Lemma 11 (Substitution for join). $\delta(t \sqcup l) = \delta(t) \sqcup \delta(l)$

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 18 (substitution for logical relations), while the last four are used in proving Lemma 13 (substitution for typing). Similarly, substitution also respects subtyping for base types, principal sets, and policy sets.

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* $\Delta, \alpha \preceq p; \Gamma; \pi \vdash e : t$*, 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 18 (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* $\mathcal{A}';; \pi' \vdash e' : t$ *such that* $\mathcal{A} \preceq \mathcal{A}'$ *and* $\pi \preceq \pi'$*.*

Proof. By induction on the typing derivations:

- T-Var: $\dfrac{x : t \in \Gamma}{\mathcal{A};; \pi \vdash x : t}$

  e has no evaluation rule, hence this case does not apply.

- T-Unit: $\dfrac{\vdash l}{\mathcal{A};; \pi \vdash \texttt{*} : \texttt{1}_l}$

  e has no evaluation rule, hence this case does not apply.

- T-Inl: $\dfrac{\mathcal{A};; \pi \vdash e_1 : t_1 \quad \vdash l}{\mathcal{A};; \pi \vdash \texttt{inl}\, e_1 : (t_1 + t_2)_l}$

  There is one possible evaluation rule, E-Inl: $\dfrac{\mathcal{A}, e_1 \longrightarrow \mathcal{A}, e_1'}{\mathcal{A}, \texttt{inl}\, e_1 \longrightarrow \mathcal{A}, \texttt{inl}\, e_1'}$

  By IH on e_1, we have $\mathcal{A}';; \pi' \vdash e_1' : t_1$. The result follows by T-Inl.

- T-Inr: symmetric to T-Inl.

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

  There are three possible evaluation rules:

  1. E-Case: $\dfrac{\mathcal{A}, e_0 \longrightarrow \mathcal{A}, e_0'}{\mathcal{A}, \texttt{case}\, e_0\ v_1\ v_2 \longrightarrow \mathcal{A}, \texttt{case}\, e_0'\ v_1\ v_2}$

     By IH on e_0, we have $\mathcal{A}';; \pi' \vdash e_0' : (t_1 + t_2)_l$. The result follows by Lemma 10 (weakening) and T-Case.

  2. E-CaseInl: $\mathcal{A}, \texttt{case}\ (\texttt{inl}\ v)\ v_1\ v_2 \longrightarrow \mathcal{A}, v_1\ v$

     By Lemma 9 (inversion), $\mathcal{A};; \pi \vdash v : t_1$. The result follows by Lemma 10 (weakening) and T-App.

3. E-CaseInr: symmetric to E-CaseInl.

- T-Fun: $\dfrac{\mathcal{A}; x : t_1; \pi \vdash e_0 : t_2 \qquad \vdash l}{\mathcal{A};; \pi \vdash \lambda x{:}t_1.\, e_0 : ([\pi]\ t_1 \to t_2)_l}$

  e has no evaluation rule, hence this case does not apply.

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

  There are three possible evaluation rules:

  1. E-App1: $\dfrac{\mathcal{A}, e_1 \longrightarrow \mathcal{A}, e_1'}{\mathcal{A}, e_1\ e_2 \longrightarrow \mathcal{A}, e_1'\ e_2}$

     By IH on e_1, we have $\mathcal{A}';; \pi' \vdash e_1' : ([\pi_2]\ t_1 \to t_2)_l$. The result follows by Lemma 10 (weakening) and T-App.

  2. E-App2: similar to E-App1.

  3. E-AppFun: $\mathcal{A}, (\lambda x{:}t_1.\, e_0)\ v \longrightarrow \mathcal{A}, e_0\{v/x\}$

     By Lemma 9 (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.

- T-PName: $\dfrac{\vdash l}{\mathcal{A};; \pi \vdash X : (\mathtt{P}_X)_l}$

  e has no evaluation rule, hence this case does not apply.

- T-Cap: $\dfrac{\vdash l}{\mathcal{A};; \pi \vdash X\mathtt{\{}i\mathtt{\}} : \mathtt{C}_l}$

  e has no evaluation rule, hence this case does not apply.

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

  There are four possible evaluation rules:

  1. E-IfDel1: $\dfrac{\mathcal{A}, e_1 \longrightarrow \mathcal{A}, e_1'}{\mathcal{A}, \mathtt{if}\ (e_1 \preceq e_2)\ e_3\ e_4 \longrightarrow \mathcal{A}, \mathtt{if}\ (e_1' \preceq e_2)\ e_3\ e_4}$

     By IH on e_1, we have $\mathcal{A}';; \pi' \vdash e_1' : (\mathtt{P}_{X_1})_l$. The result follows by Lemma 10 (weakening) and T-IfDel.

  2. E-IfDel2: similar to E-IfDel1.

  3. E-IfDelYes: $\dfrac{\mathcal{A} \vdash X_1 \preceq X_2}{\mathcal{A}, \mathtt{if}\ (X_1 \preceq X_2)\ e_3\ e_4 \longrightarrow \mathcal{A}, e_3}$

     Let $A' = \mathcal{A}, p_1 \preceq p_2$. The result follows by Lemma 10 (weakening) and T-Sub.

  4. E-IfDelNo: similar to E-IfDelYes.

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

  There are four possible evaluation rules:

  1. E-LetDel1: $\dfrac{\mathcal{A}, e_2 \longrightarrow \mathcal{A}, e_2'}{\mathcal{A}, \mathtt{let}\ (v \preceq e_2)\ \mathtt{in}\ e_3 \longrightarrow \mathcal{A}, \mathtt{let}\ (v \preceq e_2')\ \mathtt{in}\ e_3}$

     By IH on e_1, we have $\mathcal{A}';; \pi' \vdash e_1' : (\mathtt{P}_{p_1})_l$. The result follows by Lemma 10 (weakening) and T-LetDel.

  2. E-LetDel2: similar to E-LetDel1.

3. E-LetDel: similar to E-LetDel1.
4. E-LetDel: by Lemma 10 (weakening) and T-Sub.

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

  There are four possible evaluation rules:

  1. E-IfCert1: $\dfrac{\mathcal{A}, e_1 \longrightarrow \mathcal{A}, e_1'}{\mathcal{A}, \mathtt{if}\ (e_1 \Rightarrow e_2 \triangleright i)\ e_3\ e_4 \longrightarrow \mathcal{A}, \mathtt{if}\ (e_1' \Rightarrow e_2 \triangleright i)\ e_3\ e_4}$

     By IH on e_1, we have $\mathcal{A}';;\pi' \vdash e_1' : \mathtt{C}_l$. The result follows by Lemma 10 (weakening) and T-IfCert.
  2. E-IfCert2: similar to E-IfCert1.
  3. E-IfCertYes: $\dfrac{\mathcal{A} \vdash X_1\{i\} \Rightarrow X_2 \triangleright i}{\mathcal{A}, \mathtt{if}\ (X_1\{i\} \Rightarrow X_2 \triangleright i)\ e_3\ e_4 \longrightarrow \mathcal{A}, e_3}$

     Let $\pi' = \pi, p \triangleright i$. The result follows by T-Sub.
  4. E-IfCertNo: similar to E-IfCertYes.

- T-Dcls: $\dfrac{\mathcal{A};;\pi \vdash e_0 : t_2 \quad \mathcal{A} \vdash t_2 - t_1 = s \quad \mathcal{A} \vdash s \preceq \pi(\mathtt{declassify})}{\mathcal{A};;\pi \vdash \mathtt{declassify}\, e_0\ t_1 : t_1}$

  There is two possible evaluation rule

  1. E-Dcls1: $\dfrac{\mathcal{A}, e_0' \longrightarrow \mathcal{A}, e'}{\mathcal{A}, \mathtt{declassify}\, e_0\ t \longrightarrow \mathcal{A}, e_0}$

     By IH on e_0, we have $\mathcal{A}';;\pi' \vdash e_0' : t_2$. The result follows by Lemma 10 (weakening) and T-Dcls.
  2. E-Dcls: $\mathcal{A}, \mathtt{declassify}\, v\ t_0 \longrightarrow \mathcal{A}, v$. Assume we only declassify a label at the top-level type, that is $\mathcal{A} \vdash t_{l'} - t_l = s$, where $t_2 = t_{l'}$ and $t_1 = t_l$. 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-PName, T-Cap and T-All), we can change the type of v from $\mathcal{A};;\pi \vdash v : t_{l'}$ to $\mathcal{A};;\pi \vdash v : t_l$.

     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 $\mathcal{A}, \lfloor e \rfloor \longrightarrow \mathcal{A}, \lfloor e' \rfloor$, then $\mathcal{A};;\pi \vdash \lfloor e' \rfloor : t$, where $\lfloor\cdot\rfloor$ is the type-erasure function. We omit the proof for this general case here.

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

  There are three possible evaluation rules:

  1. E-Acq: $\dfrac{\mathcal{A}, e_0 \longrightarrow \mathcal{A}, e_0'}{\mathcal{A}, \mathtt{acquire}\ e_0 \triangleright i \longrightarrow \mathcal{A}, \mathtt{acquire}\ e_0' \triangleright i}$

     By IH on e_0, we have $\mathcal{A}';;\pi' \vdash e_0' : \mathtt{C}_l$. The result follows by Lemma 10 (weakening) and T-Acq.
  2. E-AcqYes: $\dfrac{\mathcal{E} \vdash X\{i\}}{\mathcal{A}, \mathtt{acquire}\ X \triangleright i \longrightarrow \mathcal{A}, \mathtt{inl}\ X\{i\}}$

     By T-Cap and T-Inl.
  3. E-AcqNo: similar to E-AcqYes.

- T-All: $\dfrac{\Delta, \alpha \preceq p; \Gamma; \pi \vdash e_0 : t_0 \quad \alpha \notin \mathrm{dom}(\Delta) \quad \vdash l}{\mathcal{A};;\pi \vdash \Lambda\alpha \preceq p.\ e_0 : (\forall \alpha \preceq p.\ t_0)_l}$

  e has no evaluation rule, hence this case does not apply.

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

  There are two possible evaluation rules:

  1. E-PApp: $\dfrac{\mathcal{A}, e_0 \longrightarrow \mathcal{A}, e_0'}{\mathcal{A}, e_0\ [X] \longrightarrow \mathcal{A}, e_0'\ [X]}$

     By IH on e_0, we have $\mathcal{A}';;\pi' \vdash e_0' : (\forall\alpha \preceq p.\ t_0)_l$. The result follows by Lemma 10 (weakening) and T-PApp.

  2. E-PAppAll: $\mathcal{A}, (\Lambda\alpha \preceq p.\ e_0)\ [X] \longrightarrow \mathcal{A}, e_0\{X/\alpha\}$

     By Lemma 9 (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 10 (weakening) and T-Sub.

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

  By IH on e_0.

□

Proposition 15 (Join order). $\Delta \vdash l_1 \sqsubseteq l_1 \sqcup l_2$.

Proposition 16 (Join commutativity). $l_1 \sqcup l_2 = l_2 \sqcup l_1$.

E Noninterference

This section proves the main result of the paper: noninterference theorem. The main lemmas are substitution for logical relations and subtyping for logical relations. The intuition is that in secure programs, high-security inputs do not interfere with low-security outputs.

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 [19]. However, we parameterize the relations with an upper-bound ζ of the observer's security label, capturing the dependence of the terms' equivalence on the observer's label.

The definitions and the theorems for this section are only for the core calculus–no authority, capability, declassification or delegation. Extending the result to the full calculus is left for future work.

$$\frac{\begin{array}{c}\mathcal{A} \vdash \Gamma \quad \mathcal{A} \vdash \gamma \models \Gamma \quad \mathcal{A} \vdash \gamma' \models \Gamma \\ \forall(x : t \in \Gamma).\ \mathcal{A} \vdash \gamma(x) \sim_\zeta \gamma'(x) : t\end{array}}{\mathcal{A} \vdash \gamma \approx_\zeta \gamma' : \Gamma} \qquad \boxed{\text{R-Subs}}$$

$$\frac{\begin{array}{c}\mathcal{A}, e \longrightarrow^* \mathcal{A}, v \quad \mathcal{A}, e' \longrightarrow^* \mathcal{A}, v' \\ \mathcal{A} \vdash e : t \quad \mathcal{A} \vdash e' : t \quad \mathcal{A} \vdash v \sim_\zeta v' : t\end{array}}{\mathcal{A} \vdash e \approx_\zeta e' : t} \qquad \boxed{\text{R-Term}}$$

$$\boxed{\mathcal{A} \vdash v \sim_\zeta v : t} \qquad \boxed{\text{Related values}}$$

$$\frac{\mathcal{A} \vdash l \not\sqsubseteq \zeta}{\mathcal{A} \vdash v \sim_\zeta v' : u_l} \qquad \text{R-Label}$$

$$\mathcal{A} \vdash \mathtt{*} \sim_\zeta \mathtt{*} : \mathtt{1}_l \qquad \text{R-Unit}$$

$$\frac{\mathcal{A} \vdash v \sim_\zeta v' : t_1}{\mathcal{A} \vdash \mathtt{inl}\ v \sim_\zeta \mathtt{inl}\ v' : (t_1 + t_2)_l} \qquad \text{R-Inl}$$

$$\frac{\mathcal{A} \vdash v \sim_\zeta v' : t_2}{\mathcal{A} \vdash \mathtt{inr}\ v \sim_\zeta \mathtt{inr}\ v' : (t_1 + t_2)_l} \qquad \text{R-Inr}$$

$$\frac{\forall(\mathcal{A} \vdash v_2 \sim_\zeta v_2' : t_1).\ \mathcal{A} \vdash (v\ v_2) \approx_\zeta (v'\ v_2') : t_2 \sqcup l}{\mathcal{A} \vdash v \sim_\zeta v' : (t_1 \to t_2)_l} \qquad \text{R-Fun}$$

$$\mathcal{A} \vdash X \sim_\zeta X : (\mathtt{P}_X)_l \qquad \text{R-PName}$$

$$\frac{\forall(\mathcal{A} \vdash X \preceq p).\ \mathcal{A} \vdash (v\ [X]) \approx_\zeta (v'\ [X]) : t \sqcup l}{\mathcal{A} \vdash v \sim_\zeta v' : (\forall\alpha \preceq p.\ t)_l} \qquad \text{R-All}$$

Lemma 17 (Subtyping for logical relations).

1. *If $\mathcal{A} \vdash e \approx_\zeta e' : t$ and $\mathcal{A} \vdash t \leq t'$, then $\mathcal{A} \vdash e \approx_\zeta e' : t'$.*

2. *If $\mathcal{A} \vdash v \sim_\zeta v' : t$ and $\mathcal{A} \vdash t \leq t'$, then $\mathcal{A} \vdash v \sim_\zeta v' : t'$.*

3. *If $\mathcal{A} \vdash v \sim_\zeta v' : u_l$ and $\mathcal{A} \vdash l \sqsubseteq l'$, then $\mathcal{A} \vdash v \sim_\zeta v' : u_{l'}$.*

4. *If $\mathcal{A} \vdash v \sim_\zeta v' : u_l$ and $\mathcal{A} \vdash u \leq u'$, then $\mathcal{A} \vdash v \sim_\zeta v' : {u'}_l$.*

Proof. Part (1): By T-Sub, the result terms are well-typed. By IH of Part (2), their evaluated values are related. The result then follows by R-Term.

Part (2): by St-UL, Part (3) and Part (4).

Part (3): by Lst-trans and R-Label.

Part (4): by induction on the subtyping derivations:

- St-Refl: $\mathcal{A} \vdash u \leq u$

  The result trivially follows because $u = u'$.

- St-Trans: $\dfrac{\mathcal{A} \vdash u \leq u' \quad \mathcal{A} \vdash u' \leq u''}{\mathcal{A} \vdash u \leq u''}$

  The result follows by IH.

- St-Sum: $\dfrac{\mathcal{A} \vdash t_1 \leq t_1' \quad \mathcal{A} \vdash t_2 \leq t_2'}{\mathcal{A} \vdash (t_1 + t_2) \leq (t_1' + t_2')}$

  By the inversion of R-Sum with $\mathcal{A} \vdash v \sim_\zeta v' : (t_1 + t_2)_l$, either

  1. R-Label with $\mathcal{A} \vdash l \not\sqsubseteq \zeta$: the result follows by R-Label.
  2. R-Inl with $v = \texttt{inl}\, v_0$, $v' = \texttt{inl}\, v_0'$ and $\mathcal{A} \vdash v_0 \sim_\zeta v_0' : t_1$: by IH of Part (2) with $\mathcal{A} \vdash t_1 \leq t_1'$, we have $\mathcal{A} \vdash v_0 \sim_\zeta v_0' : t_1'$. The result follows by R-Inl.
  3. R-Inr: symmetric to the previous case.

- St-Fun: $\dfrac{\mathcal{A} \vdash t_1' \leq t_1 \quad \mathcal{A} \vdash t_2 \leq t_2'}{\mathcal{A} \vdash (t_1 \to t_2) \leq (t_1' \to t_2')}$

  By R-Fun, we need to show that $\forall \mathcal{A} \vdash v_2 \sim_\zeta v_2' : t_1'$,

  $$\mathcal{A} \vdash (v\ v_2) \approx_\zeta (v'\ v_2') : t_2' \sqcup l$$

  By IH of Part (2) with $\mathcal{A} \vdash t_1' \leq t_1$, we have $\mathcal{A} \vdash v_2 \sim_\zeta v_2' : t_1$. By the inversion of R-Fun with $\mathcal{A} \vdash v \sim_\zeta v' : (t_1 \to t_2)_l$, we have $\mathcal{A} \vdash (v\ v_2) \approx_\zeta (v'\ v_2') : t_2 \sqcup l$. By St-UL with $\mathcal{A} \vdash t_2 \leq t_2'$ and $\mathcal{A} \vdash l \sqsubseteq l$, we have $\mathcal{A} \vdash t_2 \sqcup l \leq t_2' \sqcup l$. Then, the results follows by IH of Part (1).

- St-All: $\dfrac{\mathcal{A} \vdash p' \preceq p \quad \Delta, \alpha \preceq p' \vdash t_0 \leq t_0'}{\mathcal{A} \vdash (\forall \alpha \preceq p.\, t_0) \leq (\forall \alpha \preceq p'.\, t_0')}$

  By R-All, we need to show that $\forall \mathcal{A} \vdash X \preceq p'$,

  $$\mathcal{A} \vdash (v\ [X]) \approx_\zeta (v'\ [X]) : t_0' \sqcup l$$

  By the inversion of R-All with $\mathcal{A} \vdash v \sim_\zeta v' : (\forall \alpha \preceq p.\, t_0)_l$, we have $\mathcal{A} \vdash (v\ [X]) \approx_\zeta (v'\ [X]) : t_0 \sqcup l$. By St-UL with $\mathcal{A} \vdash t_0 \leq t_0'$ and $\mathcal{A} \vdash l \sqsubseteq l$, we have $\mathcal{A} \vdash t_0 \sqcup l \leq t_0' \sqcup l$. By IH of Part (2), we have $\mathcal{A} \vdash v \sim_\zeta v' : (\forall \alpha \preceq p.\, t_0')_l$. By J-UL and St-All with $\mathcal{A} \vdash p' \leq p$, we have $\mathcal{A} \vdash (\forall \alpha \preceq p.\, t_0')_l \leq (\forall \alpha \preceq p'.\, t_0')_l$. Then, the results follows by IH of Part (1).

□

Lemma 18 (Substitution for logical relations). *If $\Delta; \Gamma \vdash e : t$, $\delta \models \Delta$, $\mathcal{A} = \delta(\Delta)$ and $\mathcal{A} \vdash \gamma \approx_\zeta \gamma' : \delta(\Gamma)$, then*

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

Proof. Let us name the assumptions:

1. $\Delta; \Gamma \vdash e : t$ (Z-Type)
2. $\delta \models \Delta$ (Z-DModel)
3. $\mathcal{A} = \delta(\Delta)$ (Z-IfDel)
4. $\mathcal{A} \vdash \gamma \approx_\zeta \gamma' : \delta(\Gamma)$ (Z-RSubs)

By Lemma 13 (Part 1), the whole terms are well-typed. It remains to show that $\mathcal{A}, \gamma\delta(e) \longrightarrow^* \mathcal{A}, v$ and $\mathcal{A}, \gamma'\delta(e') \longrightarrow^* \mathcal{A}, v'$ and

$$\mathcal{A} \vdash v \sim_\zeta v' : t$$

which we prove by induction on the typing derivations.

- T-Var: $\dfrac{x : t \in \Gamma}{\Delta; \Gamma \vdash x : t}$

  The result follows by Z-RSubs and R-Subs.

- T-Unit: $\dfrac{\Delta \vdash l}{\Delta; \Gamma \vdash \texttt{*} : \mathtt{1}_l}$

  By Esu-Unit, $\gamma\delta(e) = \gamma'\delta(e) = \texttt{*}$. Their evaluated values are related by EM-Refl and R-Unit. The result then follows by R-Term.

- T-Inl: $\dfrac{\Delta; \Gamma \vdash e_1 : t_1 \quad \Delta \vdash l}{\Delta; \Gamma \vdash \texttt{inl}\, e_1 : (t_1 + t_2)_l}$

  By Esu-Inl, Tsu-Inl and Lemma 11 (substitution for join),

$$\begin{aligned} \gamma\delta(\texttt{inl}\, e_1) &= \texttt{inl}\, \gamma\delta(e_1) && (1)\\ \gamma'\delta(\texttt{inl}\, e_1) &= \texttt{inl}\, \gamma'\delta(e_1) && (2)\\ \delta((t_1 + t_2)_l) &= (\delta(t_1) + \delta(t_2))_{\delta(l)} && (3) \end{aligned}$$

  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 EM-Trans and E-Inl, we have $\mathcal{A}, \texttt{inl}\, \gamma\delta(e_1) \longrightarrow^* \mathcal{A}, \texttt{inl}\, v$ and $\mathcal{A}, \texttt{inl}\, \gamma'\delta(e_1) \longrightarrow^* \mathcal{A}, \texttt{inl}\, v'$. The result follows by (1)-(3), R-Inl and R-Term.

- T-Inr: symmetric to T-Inl.

- T-Case: $\dfrac{\Delta; \Gamma \vdash e_0 : (t_1 + t_2)_l \quad \Delta; \Gamma \vdash v_1 : (t_1 \to t_0)_l \quad \Delta; \Gamma \vdash v_2 : (t_2 \to t_0)_l}{\Delta; \Gamma \vdash \texttt{case}\, e_0\; v_1\; v_2 : t_0 \sqcup l}$

  By IH on e_0, v_1 and v_2,

  1. $\mathcal{A} \vdash \gamma\delta(e_0) \approx_\zeta \gamma'\delta(e_0) : \delta((t_1 + t_2)_l)$ with $\mathcal{A}, \gamma\delta(e_0) \longrightarrow^* \mathcal{A}, v_0$ and $\mathcal{A}, \gamma'\delta(e_0) \longrightarrow^* \mathcal{A}, v_0'$
  2. $\mathcal{A} \vdash \gamma\delta(v_1) \approx_\zeta \gamma'\delta(v_1) : \delta((t_1 \to t_0)_l)$ with $\mathcal{A}, \gamma\delta(v_1) \longrightarrow^* \mathcal{A}, \gamma\delta(v_1)$ and $\mathcal{A}, \gamma'\delta(v_1) \longrightarrow^* \mathcal{A}, \gamma'\delta(v_1)$
  3. $\mathcal{A} \vdash \gamma\delta(v_2) \approx_\zeta \gamma'\delta(v_2) : \delta((t_2 \to t_0)_l)$ with $\mathcal{A}, \gamma\delta(v_2) \longrightarrow^* \mathcal{A}, \gamma\delta(v_2)$ and $\mathcal{A}, \gamma'\delta(v_2) \longrightarrow^* \mathcal{A}, \gamma'\delta(v_2)$

  By Esu-Case and EM-Trans with E-Case,

$$\begin{aligned} \gamma\delta(\texttt{case}\, e_0\; v_1\; v_2) &= \texttt{case}\, \gamma\delta(e_0)\; \gamma\delta(v_1)\; \gamma\delta(v_2)\\ \gamma'\delta(\texttt{case}\, e_0\; v_1\; v_2) &= \texttt{case}\, \gamma'\delta(e_0)\; \gamma'\delta(v_1)\; \gamma'\delta(v_2)\\ \mathcal{A}, \texttt{case}\, \gamma\delta(e_0)\; \gamma\delta(v_1)\; \gamma\delta(v_2) &\longrightarrow^* \mathcal{A}, \texttt{case}\, v_0\; \gamma\delta(v_1)\; \gamma\delta(v_2)\\ \mathcal{A}, \texttt{case}\, \gamma'\delta(e_0)\; \gamma'\delta(v_1)\; \gamma'\delta(v_2) &\longrightarrow^* \mathcal{A}, \texttt{case}\, v_0'\; \gamma'\delta(v_1)\; \gamma'\delta(v_2) \end{aligned}$$

  By Tsu-Sum and the inversion of $\mathcal{A} \vdash v_0 \sim_\zeta v_0' : \delta((t_1 + t_2)_l)$, either

  1. R-Label with $\mathcal{A} \vdash l \not\sqsubseteq \zeta$: by R-Label and R-Term.
  2. R-Inl with $v_0 = \texttt{inl}\, v$ and $v_0' = \texttt{inl}\, v'$ with $\mathcal{A} \vdash v \sim_\zeta v' : \delta(t_1)$: by EM-Trans with E-CaseInl,

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

     By Esu-App, Tsu-Fun, Lemma 11 (substitution for join) 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 EM-Trans, we have related values for the result terms.

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

- T-Fun: $\dfrac{\Delta;\Gamma, x : t_1 \vdash e_0 : t_2 \quad x \notin \mathrm{dom}(\Gamma) \quad \Delta \vdash l}{\Delta;\Gamma \vdash \lambda x\!:\!t_1.\ e_0 : (t_1 \to t_2)_l}$

  $x \notin \mathrm{dom}(\gamma)$ because $x \notin \mathrm{dom}(\Gamma)$.

  By Esu-Fun, Tsu-Fun and Lemma 11 (substitution for join),

$$\begin{aligned}
\gamma\delta(\lambda x\!:\!t_1.\ e_0) &= \lambda x\!:\!\delta(t_1).\ \gamma\delta(e_0) && (1)\\
\gamma'\delta(\lambda x\!:\!t_1.\ e_0) &= \lambda x\!:\!\delta(t_1).\ \gamma'\delta(e_0) && (2)\\
\delta((t_1 \to t_2)_l) &= (\delta(t_1) \to \delta(t_2))_{\delta(l)} && (3)\\
\delta(t_2) \sqcup \delta(l) &= \delta(t_2 \sqcup l) && (4)
\end{aligned}$$

  By (1)-(4), EM-Refl, 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 EM-Trans with E-AppFun,

$$\begin{aligned}
\mathcal{A}, (\lambda x\!:\!\delta(t_1).\ \gamma\delta(e_0))\ v &\longrightarrow^* \mathcal{A}, \gamma\delta(e_0)\{v/x\} && (5)\\
\mathcal{A}, (\lambda x\!:\!\delta(t_1).\ \gamma'\delta(e_0))\ v' &\longrightarrow^* \mathcal{A}, \gamma'\delta(e_0)\{v'/x\} && (6)
\end{aligned}$$

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

$$\begin{aligned}
\gamma\delta(e_0)\{v/x\} &= \gamma_0\delta(e_0) && (7)\\
\gamma'\delta(e_0)\{v/x\} &= \gamma'_0\delta(e_0) && (8)\\
\mathcal{A} \vdash \gamma_0 &\approx_\zeta \gamma'_0 : \delta(\Gamma, x : t_1) && (9)
\end{aligned}$$

  By IH with Z-DModel and Z-IfDel and (9), $\mathcal{A} \vdash \gamma_0\delta(e_0) \approx_\zeta \gamma'_0\delta(e_0) : \delta(t_2 \sqcup l)$. Then, the result follows by (5)-(8) and R-Term.

- T-App: $\dfrac{\Delta;\Gamma \vdash e_1 : (t_1 \to t_2)_l \quad \Delta;\Gamma \vdash e_2 : t_1}{\Delta;\Gamma \vdash e_1\ e_2 : t_2 \sqcup l}$

  By IH on e_1 and e_2,

  1. $\mathcal{A} \vdash \gamma\delta(e_1) \approx_\zeta \gamma'\delta(e_1) : \delta((t_1 \to t_2)_l)$ with $\mathcal{A}, \gamma\delta(e_1) \longrightarrow^* \mathcal{A}, v_1$ and $\mathcal{A}, \gamma'\delta(e_1) \longrightarrow^* \mathcal{A}, v'_1$
  2. $\mathcal{A} \vdash \gamma\delta(e_2) \approx_\zeta \gamma'\delta(e_2) : \delta(t_1)$ with $\mathcal{A}, \gamma\delta(e_2) \longrightarrow^* \mathcal{A}, v_2$ and $\mathcal{A}, \gamma'\delta(e_2) \longrightarrow^* \mathcal{A}, v'_2$

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

- T-PName: $\dfrac{\Delta \vdash l}{\Delta;\Gamma \vdash X : (\mathtt{P}_X)_l}$

  By Esu-PName and Tsu-PName, $\gamma\delta(e) = \gamma'\delta(e) = X$ and $\delta((\mathtt{P}_X)_l) = (\mathtt{P}_{\delta(X)})_l$. The result then follows by EM-Refl, R-PName and R-Term.

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

  By IH on e_1, e_2, e_3 and e_4,

  1. $\mathcal{A} \vdash \gamma\delta(e_1) \approx_\zeta \gamma'\delta(e_1) : \delta((\mathtt{P}_{p_1})_l)$ with $\mathcal{A}, \gamma\delta(e_1) \longrightarrow^* \mathcal{A}, v_1$ and $\mathcal{A}, \gamma'\delta(e_1) \longrightarrow^* \mathcal{A}, v'_1$
  2. $\mathcal{A} \vdash \gamma\delta(e_2) \approx_\zeta \gamma'\delta(e_2) : \delta((\mathtt{P}_{p_2})_l)$ with $\mathcal{A}, \gamma\delta(e_2) \longrightarrow^* \mathcal{A}, v_2$ 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 $\mathcal{A}, \gamma'\delta(e_3) \longrightarrow^* \mathcal{A}, v'_4$

By Esu-Case and EM-Trans with E-Case,

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

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

1. R-Label with $\mathcal{A} \vdash l \not\sqsubseteq \zeta$: by R-Label, we have related values for the result terms.
2. R-PName, R-PName 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 EM-Trans with E-IfDel3,

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

   Otherwise, if $\mathcal{A} \vdash \delta(p_1) \not\preceq \delta(p_2)$, then by EM-Trans with E-IfDel4,

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

   In both cases, by EM-Trans, we have related values for the result terms at type $\delta(t_0)$. By Proposition 15 (join order) and Lemma 12 (substitution for subtyping), they are also related at type $\delta(t_0 \sqcup l)$. The result then follows by R-Term.

- T-All: $\dfrac{\Delta, \alpha \preceq p; \Gamma \vdash e_0 : t_0 \quad \alpha \notin \mathrm{dom}(\Delta) \quad \Delta \vdash l}{\Delta; \Gamma \vdash \Lambda\alpha \preceq p.\ e_0 : (\forall\alpha \preceq p.\ t_0)_l}$

  $\alpha \notin \mathrm{dom}(\gamma)$ because $\alpha \notin \Delta$.

  By Esu-All, Tsu-All and Lemma 11 (substitution for join),

  $$\begin{array}{rcll} \gamma\delta(\Lambda\alpha \preceq p.\ e_0) & = & \Lambda\alpha \preceq \delta(p).\ \gamma\delta(e_0) & (1) \\ \gamma'\delta(\Lambda\alpha \preceq p.\ e_0) & = & \Lambda\alpha \preceq \delta(p).\ \gamma'\delta(e_0) & (2) \\ \delta(\forall\alpha \preceq p.\ t_0) & = & (\forall\alpha \preceq p.\ \delta(t_0))_{\delta(l)} & (3) \\ \delta(t_0) \sqcup \delta(l) & = & \delta(t_0 \sqcup l) & (4) \end{array}$$

  By (1)-(4), EM-Refl, 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 EM-Trans with E-PAppAll,

  $$\begin{array}{rl} \mathcal{A}, (\Lambda\alpha \preceq \delta(p).\ \gamma\delta(e_0))\ [X] \longrightarrow^* \mathcal{A}, \gamma\delta(e_0)\{X/\alpha\} & (5) \\ \mathcal{A}, (\Lambda\alpha \preceq \delta(p).\ \gamma'\delta(e_0))\ [X] \longrightarrow^* \mathcal{A}, \gamma'\delta(e_0)\{X/\alpha\} & (6) \end{array}$$

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

  $$\begin{array}{rl} \gamma\delta(e_0)\{X/\alpha\} = \gamma\delta_0(e_0) & (7) \\ \gamma'\delta(e_0)\{X/\alpha\} = \gamma'\delta_0(e_0) & (8) \\ \delta_0 \models \Delta, \alpha \preceq p & (9) \end{array}$$

  By IH with Z-IfDel, (13) and Z-RSubs, $\mathcal{A} \vdash \gamma\delta(e_0) \approx_\zeta \gamma'\delta(e_0) : \delta(t_0 \sqcup l)$. Then, the result follows by (7)-(12) and R-Term.

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

  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 two principals on the right are both $\delta(p_1)$ and, by $\mathcal{A} \vdash p_1 \preceq p_2$ and Lemma 12, we have $\mathcal{A} \vdash \delta(p_1) \preceq \delta(p_2)$. The result then follows by R-All and R-Term.

- T-Sub: $\dfrac{\Delta; \Gamma \vdash e_0 : t_1 \quad \Delta \vdash t_1 \leq t_2}{\Delta; \Gamma \vdash e_0 : t_2}$

  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 then follows by Lemma 17 (subtyping for logical relations) and R-Term.

□