A Design for a Security-typed Language with Certificate-based Declassification

Stephen Tse Steve Zdancewic

University of Pennsylvania

Abstract. This paper presents a calculus that supports information-flow security policies and certificate-based declassification. The decentralized label model and its downgrading mechanisms are concisely expressed in the polymorphic lambda calculus with subtyping (System $\mathtt{F}_{\preceq}$). We prove a conditioned version of the noninterference theorem such that authorization for declassification is justified by digital certificates from public-key infrastructures.

1 Introduction

Information-flow policies constrain the propagation of confidential data and provide an end-to-end guarantee of security. *Security-typed languages* have become a promising approach for specifying and enforcing such policies with static type systems [15]. However, designing a *safe* and *secure* information-flow type system is still a challenging problem: programmers want to express fine-grained security policies with advanced types, but reasoning about security guarantees in such complex systems is non-trivial.

This paper presents a security-typed language with well-studied constructs from the polymorphic lambda calculus with subtyping (System $\mathtt{F}_{\preceq}$) [6]. Language features such as labels and effects are isolated in a monadic style. This design makes typing and evaluation rules easy to understand and the proofs of *type-safety* and *noninterference* modular.

Another challenge of designing a security-typed language is to provide downgrading mechanisms that intentionally break the security guarantees, if such actions can be justified externally. Downgrading mechanisms, such as *delegating* to another principal or *declassifying* secret data, are important in practical programming [21]. The decentralized label model by Myers and Liskov [10] addresses this problem and introduces the notions of principals and reader sets to statically track the authority for downgrading.

One of our design decisions is to treat labels, principals and downgrading privileges uniformly as types so that the decentralized label model can be integrated easily into our language. For example, subtyping naturally models principal delegation, while intersection and union types give rise to principal groups and label refinements. A security language with these encodings allows expressive, decentralized policies, yet the semantics remains easy to understand.

Our previous work [18] connects the static security type system with runtime security mechanisms such as public-key infrastructures. The language there

uses *singleton types* such that a principal is represented as a public key, and the authority of a principal granting a privilege is represented as a digital certificate.

We improve on our previous work with monads and subtyping, allowing us to prove a conditioned version of noninterference even in the presence of declassification (which was neither stated nor proved before). In particular, we formalize downgrading mechanisms such as delegation and declassification as subtyping, and certificate verification as *extending* the subtyping relation. More importantly, the conditioned noninterference now captures the intuition that certificates externally justify the information leaks due to declassification.

The main contributions of our paper are:

1. the design of a safe and secure information-flow type system with bounded quantification and effects in a monadic style;
2. the integration of the decentralized label model with type constructors and the use of subtyping to model delegation, declassification, and endorsement;
3. a conditioned version of the noninterference theorem that justifies certificate-based declassification.

The work here subsumes our previous work [18], adding existential types, run-time labels and privileges, and a conditioned noninterference theorem for the full language. We have also built a prototype interpreter called *Apollo*[1].

For brevity, this paper shows only the interesting cases of rules and proof. Our technical report [17] contains all rules and proofs of type-safety and noninterference for the full language. The proof of type-safety is also mechanically formalized and checked with Twelf (a logical framework).

The rest of the paper is organized as follows. Section 2 starts with a core calculus with labels to study noninterference and extends it with effects. Section 3 introduces the decentralized label model and shows how the core calculus supports the notion of principals, confidentiality and integrity. Furthermore, downgrading mechanisms are studied as subtyping, and certificate-based declassification is justified using constructs from public-key infrastructures. The paper then discusses related work in Section 4 and concludes in Section 5.

2 Core label calculus

Let us start by introducing a core calculus with monadic labels and effects for analyzing program dependency. This section proves two important security theorems, *type-safety* and *noninterference*, for our core calculus.

2.1 Monadic labels

The first part of our label calculus is based on the dependency core calculus (DCC) [1] and the polymorphic lambda calculus with subtyping (System $\mathtt{F}_{\preceq}$) [6]. The motivation behind DCC is to use *monadic labels* as a unifying framework to

[1] The interested reader is invited to visit `http://www.cis.upenn.edu/~stse/apollo`.

study many important program analyses such as binding time, information flow, slicing, and function call tracking. DCC uses a lattice of monads and a special typing rule for their associated bind operations to describe the dependency of computations in a program.

Unlike DCC, which is based on the *call-by-name* simply-typed lambda calculus, our core calculus is based on the *call-by-value* $\mathtt{F}_{\preceq}$. Our work should apply also to call-by-name languages; we pick call-by-value semantics simply because of our familiarity. The features of $\mathtt{F}_{\preceq}$ will become essential in later sections: bounded quantification ($\forall\alpha \preceq \mathtt{t}.\mathtt{t}$ and $\exists\alpha \preceq \mathtt{t}.\mathtt{t}$) are used to connect static security policies and run-time public-key infrastructures (Sect. 3.3), and subtyping is used to model principal delegations and policy refinements (Sect. 3.1).

The following grammar defines the syntax for our basic types and terms:

$$\mathtt{t} ::= \mathtt{<>} \mid \mathtt{<t,t>} \mid \mathtt{t}+\mathtt{t} \mid \mathtt{t}\to\mathtt{t} \mid \top_{\mathtt{k}} \mid \bot_{\mathtt{k}} \mid \alpha \mid \forall\alpha \preceq \mathtt{t}.\mathtt{t} \mid \exists\alpha \preceq \mathtt{t}.\mathtt{t}$$

$$\begin{aligned}\mathtt{m} ::= {} & \mathtt{<>} \mid \mathtt{<m,m>} \mid \mathtt{prj}_1\ \mathtt{m} \mid \mathtt{prj}_2\ \mathtt{m} \mid \mathtt{inj}_1\ \mathtt{m} \mid \mathtt{inj}_2\ \mathtt{m} \mid \mathtt{case\ m\ m\ m} \mid \mathtt{x} \mid \lambda\mathtt{x}{:}\mathtt{t}.\mathtt{m} \mid \mathtt{m\ m} \\ & \mid \Lambda\alpha \preceq \mathtt{t}.\mathtt{m} \mid \mathtt{m\ [t]} \mid \mathtt{pack\ (t,m)\ as\ t} \mid \mathtt{open\ }(\alpha,\mathtt{x}) = \mathtt{m\ in\ m}\end{aligned}$$

The types consists of unit, products, sums, functions, top, bottom, variables, universal and existential quantification, while the terms consists of unit, products, projections, injections, cases, variables, functions, applications, type abstractions and instantiations, and package packings and openings. We also encode Booleans `bool` using unit and sums.

The types top $\top_{\mathtt{k}}$ and bottom $\bot_{\mathtt{k}}$ are annotated by a kind $\mathtt{k}$: types $\mathcal{T}$, labels $\mathcal{L}$, principals $\mathcal{P}$, and privileges $\mathcal{J}$. Principals and privileges will be explained in Sect. 3 with the decentralized label model. One of our design choices is to identify these syntactic classes (types $\mathtt{t}$, labels ℓ, principals $\mathtt{p}$, and privileges $\mathtt{j}$):

$$\mathtt{t} \equiv \ell \equiv \mathtt{p} \equiv \mathtt{j} \qquad\qquad \mathtt{k} ::= \mathcal{T} \mid \mathcal{L} \mid \mathcal{P} \mid \mathcal{J}$$

This design allows the reuse of type machinery, such as polymorphism and subtyping, uniformly for these concepts. We will see this benefit again for intersection and union types in Sect 3.1, and singleton types in Sect. 3.3.

We use the semantics of Kernel $\mathtt{F}_{\preceq}$ [6]. The evaluation judgment is denoted by $\mathtt{m} \longrightarrow \mathtt{m}$, the typing judgments by $\Delta;\ \Gamma \vdash \mathtt{m} : \mathtt{t}$, and the subtyping judgment by $\Delta \vdash \mathtt{t} \preceq \mathtt{t}$, where Δ is a type context and Γ is a term context. We follow Pottier's notation [13] for specifying the subtyping polarities Ⓢ of type constructors: $\oplus$ for covariant, $\ominus$ for contravariant, and $\odot$ for invariant:

$$\begin{aligned}&\Delta ::= \cdot \mid \Delta, \alpha \preceq \mathtt{t} \qquad \Gamma ::= \cdot \mid \Gamma, \mathtt{x}{:}\mathtt{t} \\ &Ⓢ ::= \mathtt{<}\oplus,\oplus\mathtt{>} \mid \ \oplus + \oplus \ \mid \ominus\to\oplus \mid \forall\alpha \preceq \odot.\ \oplus \ \mid \exists\alpha \preceq \odot.\oplus\end{aligned}$$

We omit rules for the standard $\mathtt{F}_{\preceq}$ constructs above and focus on the new types and terms for labels: monadic types $\mathtt{t}\{\ell\}$ (indexed by labels ℓ), and their corresponding units $\mathtt{m}\{\ell\}$ and `bind` operator.

$$\mathtt{t} ::= \ldots \mid \mathtt{t}\{\ell\} \qquad \mathtt{m} ::= \ldots \mid \mathtt{m}\{\ell\} \mid \mathtt{bind\ x} = \mathtt{m\ in\ m} \qquad Ⓢ ::= \ldots \mid \ \oplus\{\oplus\}$$

Syntactically, these label constructs have the highest precedence such that $\mathtt{m}_1\ \mathtt{m}_2\{\ell\}$ means $\mathtt{m}_1\ (\mathtt{m}_2\{\ell\})$. We write high and low labels as $\mathtt{H} = \top_{\mathcal{L}}$ and $\mathtt{L} = \bot_{\mathcal{L}}$. The subtyping relation of labels $\Delta \vdash \ell \preceq \ell$ forms a lattice and hence our language

has a lattice of monads $\mathtt{t}\{\ell\}$ and $\mathtt{m}\{\ell\}$. Since labels and types are in the same syntactic class, we use a kinding judgment $\Delta \vdash \mathtt{t} :: \mathtt{k}$ to rule out ill-formed types such as $\mathtt{bool} \rightarrow \mathtt{H}$ or $\mathtt{bool}\{\mathtt{bool}\}$. We omit the straight-forward kind system here; our technical report [17] contains the full details.

Now, let us see how the type system prevents low-level computation from depending on high-level computation:

$$\frac{\Delta;\ \Gamma \vdash \mathtt{m} : \mathtt{t} \quad \Delta \vdash \ell :: \mathcal{L}}{\Delta;\ \Gamma \vdash \mathtt{m}\{\ell\} : \mathtt{t}\{\ell\}} \qquad \frac{\Delta;\ \Gamma \vdash \mathtt{m}_1 : \mathtt{t}_1\{\ell\} \quad \Delta;\ \Gamma, \mathtt{x}{:}\mathtt{t}_1 \vdash \mathtt{m}_2 : \mathtt{t}_2 \quad \Delta \vdash \ell \ll \mathtt{t}_2}{\Delta;\ \Gamma \vdash \mathtt{bind}\ \mathtt{x} = \mathtt{m}_1\ \mathtt{in}\ \mathtt{m}_2 : \mathtt{t}_2}$$

The label monad $\mathtt{m}\{\ell\}$ marks the computation $\mathtt{m}$ with the label ℓ, restricting how it interacts with the rest of the program. The term $\mathtt{bind}\ \mathtt{x} = \mathtt{m}_1\ \mathtt{in}\ \mathtt{m}_2$ exposes the computation $\mathtt{m}_1$ protected inside the label type $\mathtt{t}\{\ell\}$ to the scope of $\mathtt{m}_2$.

Note that these typings are standard for monadic types, except that the return type of `bind` here has type $\mathtt{t}_2$, rather than the expected type $\mathtt{t}_2\{\ell\}$. Instead, by connecting the subtyping of labels ($\Delta \vdash \ell_1 \preceq \ell_2$) with the subtyping of types ($\Delta \vdash \mathtt{t}_1 \preceq \mathtt{t}_2$), the following label protection judgment $\Delta \vdash \ell \ll \mathtt{t}$ ensures that the result of `bind` still protects the data:

$$\Delta \vdash \ell \ll \langle\rangle \qquad \frac{\Delta \vdash \ell_2 \preceq \ell_1}{\Delta \vdash \ell_2 \ll \mathtt{t}\{\ell_1\}} \qquad \frac{\Delta \vdash \ell_2 \ll \mathtt{t}}{\Delta \vdash \ell_2 \ll \mathtt{t}\{\ell_1\}}$$

The unit type protects all labels as there is only one term of such type. Sum types, as information can be leaked by their tags, do not protect any label. The full set of rules also includes cases for products, functions, and universal types.

Example 1 *The term* $\lambda\mathtt{x}{:}\mathtt{bool}\{\mathtt{H}\}.\ \mathtt{bind}\ \mathtt{y} = \mathtt{x}\ \mathtt{in}\ \mathtt{if}\ \mathtt{y}\ \mathtt{then}\ 0\ \mathtt{else}\ 1$ *is* not *well-typed, because* $\Delta \vdash \mathtt{H} \not\ll \mathtt{int}$. *An integer leaks information just like a Boolean or a sum. In contrast,* $\lambda\mathtt{x}{:}\mathtt{bool}\{\mathtt{H}\}.\ \mathtt{bind}\ \mathtt{y} = \mathtt{x}\ \mathtt{in}\ \mathtt{if}\ \mathtt{y}\ \mathtt{then}\ 0\{\mathtt{H}\}\ \mathtt{else}\ 1\{\mathtt{H}\}$ is *well-typed, because* $\Delta \vdash \mathtt{H} \ll \mathtt{int}\{\mathtt{H}\}$. □

Operationally, the label monad $\mathtt{m}\{\ell\}$ evaluates the term inside until it is a value $\mathtt{v}\{\ell\}$, while `bind` evaluates $\mathtt{m}_1$ to a value $\mathtt{v}\{\ell\}$ and substitutes $\mathtt{v}$ for $\mathtt{x}$ in $\mathtt{m}_2$. We specify the dynamic semantics by the following syntactic classes of values $\mathtt{v}$ and evaluation contexts $\mathtt{E}$ [20], and by small-step computation rules. We use $\mathtt{m}\{\mathtt{v}/\mathtt{x}\}$ to denote the capture-free substitution of $\mathtt{v}$ for $\mathtt{x}$ in $\mathtt{m}$.

$$\begin{array}{ll} \mathtt{v} ::= \ldots \mid \mathtt{v}\{\ell\} & \\ \mathtt{E} ::= \ldots \mid \mathtt{E}\{\ell\} \mid \mathtt{bind}\ \mathtt{x} = \mathtt{E}\ \mathtt{in}\ \mathtt{m} & \mathtt{bind}\ \mathtt{x} = \mathtt{v}\{\ell\}\ \mathtt{in}\ \mathtt{m} \longrightarrow \mathtt{m}\{\mathtt{v}/\mathtt{x}\} \end{array}$$

DCC also has fixpoints and pointed types. In the technical report, we add such features to the full language and prove noninterference using a bisimulation-like technique. For the lack of space, however, such development is left out here.

2.2 Security theorems

Before we go on to enrich the language with features such as effects and the decentralized label model, let us state and prove two important theorems that guarantee the security of programs written in our language. These theorems still

hold for our full languages (modulo some condition to account for declassification, to be explained in Sect. 3.3); however, we prove them here for the core calculus first to demonstrate the proof techniques. By presenting and proving for the full language incrementally, we hope to substantiate our claim that monadic types make the design and proofs more modular.

The first theorem is the type-safety of the language, which we have proved using the progress and preservation theorems. Type-safety states that a closed, well-typed program will not get stuck or generate any error. A closed program means that both the type and term contexts are empty, that is, $\Delta = \Gamma = \cdot$.

Theorem 2 (Progress and preservation) *If* $\cdot;\ \cdot \vdash \mathtt{m}_1 : \mathtt{t}$*, then either* $\mathtt{m}_1 = \mathtt{v}$ *or* $\mathtt{m}_1 \longrightarrow \mathtt{m}_2$*. And, if* $\Delta;\ \Gamma \vdash \mathtt{m}_1 : \mathtt{t}$ *and* $\mathtt{m}_1 \longrightarrow \mathtt{m}_2$*, then* $\Delta;\ \Gamma \vdash \mathtt{m}_2 : \mathtt{t}$*.*

Proof. By induction on the typing derivation [6,1]. □

The second theorem is the noninterference property of the language [15], which states that if a program is well-typed, a low-level observer cannot distinguish between different high-level computations. The theorem requires a model of *observers* ζ for specifying what information leaks are possible. Our model here is that, given an equivalence relation over values of the same type, a well-typed observer cannot distinguish *equivalent values*, which are parameterized by the security label of the observer.

For example, we should have these equivalences for Booleans:

$$\mathtt{true} \sim_\zeta \mathtt{true} : \mathtt{bool} \qquad \mathtt{true} \not\sim_\zeta \mathtt{false} : \mathtt{bool} \qquad \mathtt{true\{H\}} \sim_\mathtt{L} \mathtt{false\{H\}} : \mathtt{bool\{H\}}$$

The first two say that no observer ζ cannot distinguish `true` from `true`, but an observer can tell the difference between `true` and `false`. More interestingly, the third says that if values are protected inside the high monad, then different values become indistinguishable to the low-level observer `L`.

Based on the intuition above, we generalize the equivalence relation in the following ways: (1) extend the relation to be higher-order, to account for functions; (2) parameterize the relation with arbitrary labels; (3) cover all types and values in the relation; and, (4) lift the relation from values to terms by evaluation.

Formally, this logical equivalence relation is defined by the following rules. We denote the equivalence relation for closed values at closed type `t` by $\mathtt{v} \sim_\zeta \mathtt{v} : \mathtt{t}$, and that for closed terms by $\mathtt{m} \approx_\zeta \mathtt{m} : \mathtt{t}$:

$$\frac{\mathtt{m}_1 \longrightarrow^* \mathtt{v}_1 \quad \mathtt{m}_2 \longrightarrow^* \mathtt{v}_2 \quad \mathtt{v}_1 \sim_\zeta \mathtt{v}_2 : \mathtt{t}}{\mathtt{m}_1 \approx_\zeta \mathtt{m}_2 : \mathtt{t}} \qquad \frac{\forall(\mathtt{v}_3 \sim_\zeta \mathtt{v}_4 : \mathtt{t}_1).\ \mathtt{v}_1\ \mathtt{v}_3 \approx_\zeta \mathtt{v}_2\ \mathtt{v}_4 : \mathtt{t}_2}{\mathtt{v}_1 \sim_\zeta \mathtt{v}_2 : \mathtt{t}_1 \to \mathtt{t}_2}$$

$$\frac{\mathtt{v}_1 \sim_\zeta \mathtt{v}_2 : \mathtt{t}}{\mathtt{v}_1\{\ell\} \sim_\zeta \mathtt{v}_2\{\ell\} : \mathtt{t}\{\ell\}} \qquad \frac{\forall(\mathtt{t}_2 \preceq \mathtt{t}_1).\ \mathtt{v}_1\ [\mathtt{t}_2] \approx_\zeta \mathtt{v}_2\ [\mathtt{t}_2] : \mathtt{t}\{\mathtt{t}_2/\alpha\}}{\mathtt{v}_1 \sim_\zeta \mathtt{v}_2 : \forall\alpha \preceq \mathtt{t}_1.\ \mathtt{t}}$$

$$\frac{\ell \not\preceq \zeta}{\mathtt{v}_1\{\ell\} \sim_\zeta \mathtt{v}_2\{\ell\} : \mathtt{t}\{\ell\}} \qquad \frac{\mathtt{v}_1 \sim_\zeta \mathtt{v}_2 : \mathtt{t}\{\mathtt{t}_1/\alpha\}}{\mathtt{pack}\ (\mathtt{t}_1, \mathtt{v}_1) \sim_\zeta \mathtt{pack}\ (\mathtt{t}_1, \mathtt{v}_2) : \exists\alpha \preceq \mathtt{t}_2.\mathtt{t}}$$

For reference in proofs later, we name these rules (from top to bottom, left to right) R-Term, R-Lab1, R-Lab2, R-Fun, R-All, and R-Some (with type annotations inside the `pack` terms elided). We slightly abuse the notation by using

$\forall$ both for the object-level quantification types $\forall\alpha \preceq \mathtt{t}.\mathtt{t}$ and for the meta-level quantification in logical relations. Note that we do not deal with parametricity of polymorphic functions [19] nor the behavioral equivalence of existential packages [12]. That is, our model assumes that an observer can differentiate different representations of polymorphic functions or different implementations of existential packages. This assumption simplifies the equivalence relations, and is the key difference between noninterference and parametricity.

The last step is to model an arbitrary observer as an open term that contains free type variables and term variables, and model observations as type substitutions δ and term substitutions γ, which are defined as:

$$\delta ::= \cdot \mid \delta, \alpha \mapsto \mathtt{t} \qquad\qquad \gamma ::= \cdot \mid \gamma, \mathtt{x} \mapsto \mathtt{v}$$

A judgment $\delta \models \Delta$ says that a type substitution models a type context: for all $\alpha \in \mathtt{dom}(\delta) = \mathtt{dom}(\Delta)$, if $\delta(\alpha) = \mathtt{t}_1$ and $\alpha \preceq \mathtt{t}_2 \in \Delta$, then $\mathtt{t}_1$ is closed, has the same kind as $\mathtt{t}_2$, and $\Delta \vdash \mathtt{t}_1 \preceq \mathtt{t}_2$. Another judgment $\gamma_1 \sim_\zeta \gamma_2 : \delta(\Gamma)$ says that two term substitutions are equivalent under a term context of closed types: for all $\mathtt{x} \in \mathtt{dom}(\gamma_1) = \mathtt{dom}(\gamma_2) = \mathtt{dom}(\delta(\Gamma))$, if $\gamma_1(\mathtt{x}) = \mathtt{v}_1$, $\gamma_2(\mathtt{x}) = \mathtt{v}_2$ and $\mathtt{x}{:}\mathtt{t} \in \delta(\Gamma)$, then $\mathtt{v}_1 \sim_\zeta \mathtt{v}_2 : \mathtt{t}$.

With the logical relations and the substitutions above, we can formally state the main theorem of the core label calculus: related substitutions preserve the logical equivalence. In other words, an arbitrary observer cannot distinguish values higher in the lattice.

Theorem 3 (Noninterference for terms) *If* $\Delta;\ \Gamma \vdash \mathtt{m} : \mathtt{t}$ *and* $\delta \models \Delta$ *and* $\gamma_1 \sim_\zeta \gamma_2 : \delta(\Gamma)$*, then* $\delta\gamma_1(\mathtt{m}) \approx_\zeta \delta\gamma_2(\mathtt{m}) : \delta(\mathtt{t})$.

Proof. By induction on the typing derivation. Case `bind`: We are given $\Delta;\ \Gamma \vdash \mathtt{bind}\ \mathtt{x} = \mathtt{m}_1\ \mathtt{in}\ \mathtt{m}_2 : \mathtt{t}_2$. By inversion, we have $\Delta;\ \Gamma \vdash \mathtt{m}_1 : \mathtt{t}_1\{\ell\}$ (*1) and $\Delta;\ \Gamma, \mathtt{x}:\mathtt{t}_1 \vdash \mathtt{m}_2 : \mathtt{t}_2$ (*2) and $\Delta \vdash \ell \ll \mathtt{t}_2$ (*3). By induction hypothesis with (*1), we have

$$\delta\gamma_1(\mathtt{m}_1) \approx_\zeta \delta\gamma_2(\mathtt{m}_1) : \delta(\mathtt{t}_1\{\ell\})$$

By inversion of R-Term, $\delta\gamma_1(\mathtt{m}_1) \longrightarrow^* \mathtt{v}_1$ (*4) and $\delta\gamma_2(\mathtt{m}_1) \longrightarrow^* \mathtt{v}_2$ (*5) and $\mathtt{v}_1 \sim_\zeta \mathtt{v}_2 : \delta(\mathtt{t}_1\{\ell\})$. Subcase $\delta(\ell) \preceq \zeta$: By the inversion of R-Lab1, $\mathtt{v}_1 = \mathtt{v}_3\{\delta(\ell)\}$ and $\mathtt{v}_2 = \mathtt{v}_4\{\delta(\ell)\}$ and $\mathtt{v}_3 \sim_\zeta \mathtt{v}_4 : \delta(\mathtt{t}_1)$ (*6). We then extend the term substitutions as

$$\gamma_1' = \gamma_1, \mathtt{x} \mapsto \mathtt{v}_3 \qquad \gamma_2' = \gamma_2, \mathtt{x} \mapsto \mathtt{v}_4$$

such that, by (*6), $\gamma_1' \sim_\zeta \gamma_2' : \delta(\Gamma, \mathtt{x}:\mathtt{t}_1)$ (*7). By induction hypothesis with (*2,*7),

$$\delta\gamma_1'(\mathtt{m}_2) \approx_\zeta \delta\gamma_2'(\mathtt{m}_2) : \delta(\mathtt{t}_2)$$

which means that $\delta\gamma_1(\mathtt{m}_2)\{\mathtt{v}_3/\mathtt{x}\} \approx_\zeta \delta\gamma_2(\mathtt{m}_2)\{\mathtt{v}_4/\mathtt{x}\} : \delta(\mathtt{t}_2)$ (*8). By (*4,*5),

$$\begin{array}{rl} & \delta\gamma_1(\mathtt{bind}\ \mathtt{x} = \mathtt{m}_1\ \mathtt{in}\ \mathtt{m}_2) \\ = & \mathtt{bind}\ \mathtt{x} = \delta\gamma_1(\mathtt{m}_1)\ \mathtt{in}\ \delta\gamma_1(\mathtt{m}_2) \\ \longrightarrow^* & \mathtt{bind}\ \mathtt{x} = \mathtt{v}_3\{\delta(\ell)\}\ \mathtt{in}\ \delta\gamma_1(\mathtt{m}_2) \\ \longrightarrow & \delta\gamma_1(\mathtt{m}_2)\{\mathtt{v}_3/\mathtt{x}\} \end{array} \qquad \begin{array}{rl} & \delta\gamma_2(\mathtt{bind}\ \mathtt{x} = \mathtt{m}_1\ \mathtt{in}\ \mathtt{m}_2) \\ = & \mathtt{bind}\ \mathtt{x} = \delta\gamma_2(\mathtt{m}_1)\ \mathtt{in}\ \delta\gamma_2(\mathtt{m}_2) \\ \longrightarrow^* & \mathtt{bind}\ \mathtt{x} = \mathtt{v}_4\{\delta(\ell)\}\ \mathtt{in}\ \delta\gamma_2(\mathtt{m}_2) \\ \longrightarrow & \delta\gamma_2(\mathtt{m}_2)\{\mathtt{v}_4/\mathtt{x}\} \end{array}$$

Therefore, by R-Term and (*8), we conclude that $\delta\gamma_1(\mathtt{bind}\ \mathtt{x} = \mathtt{m}_1\ \mathtt{in}\ \mathtt{m}_2) \approx_\zeta \delta\gamma_2(\mathtt{bind}\ \mathtt{x} = \mathtt{m}_1\ \mathtt{in}\ \mathtt{m}_2) : \delta(\mathtt{t}_2)$. Subcase $\delta(\ell) \not\preceq \zeta$: by Lemma 4 with (*3). $\square$

Lemma 4 (Noninterference for protected terms) *If* $\Delta \vdash \ell \ll \mathtt{t}$, $\delta \models \Delta$ *and* $\delta(\ell) \not\preceq \zeta$, *then* $\mathtt{m}_1 \approx_\zeta \mathtt{m}_2 : \mathtt{t}$. □

2.3 Monadic effects

We now turn to study information flows in the presence of computational effects. Practical programs interact with external systems and produce effects; observers can then learn high-security values from those effects. To prevent information leaks through such channel, we need to refine the type system with effect types.

We again use the monadic style of effect types [9,5]. The benefit of monads is that the new feature can be incrementally added to the language we have shown so far. That is, all the typing and evaluation rules in Section 2.1 remain unchanged. Traditional approaches, in contrast, require tracking of effects in all typing rules, spreading the interaction of labels and effects everywhere. Monads also help in structuring proofs in a modular way, which will be explained for Theorem 7.

For lazy languages like Haskell, we can simply add the IO monad. For eager languages like the one here, we need to introduce a new syntactic class $\mathtt{e}$ for *effectful expressions* to distinguish from *pure terms* $\mathtt{m}$ introduced in Sect. 2.1:

$$\begin{array}{ll} \mathtt{t} ::= \ldots \mid \mathtt{t}\,!\,\epsilon & \textcircled{S} ::= \ldots \mid \oplus\,!\ominus \\ \mathtt{e} ::= \mathtt{return}\ \mathtt{m} \mid \mathtt{run}\ \mathtt{x} = \mathtt{m}\ \mathtt{in}\ \mathtt{e} & \mathtt{m} ::= \ldots \mid \mathtt{e}\,!\,\epsilon \end{array}$$

Every top-level program is now an expression, instead of a term. We model effects as outputs at a given label ℓ, which are visible to an observer of level ζ if $\ell \preceq \zeta$. An observer cannot tell the difference between effects of different labels, but can count the number of visible effects happening in the program. This treatment gives us a uniform way of modeling language features with effects and could be extended to effects that carry additional values. Experience with effectful languages suggests that this technique can be extended to memory references with reads and writes [5].

Expressions are $\mathtt{return}\ \mathtt{m}$ and $\mathtt{run}\ \mathtt{x} = \mathtt{m}\ \mathtt{in}\ \mathtt{e}$, which explicitly specify the order of execution. The term $\mathtt{e}\,!\,\epsilon$ delays the effects ϵ in $\mathtt{e}$ and thus can be considered pure, but has the monadic effect type $\mathtt{t}\,!\,\epsilon$ (indexed by effect labels ϵ). Here ϵ is a lower bound on the labels of observable effects happening in $\mathtt{e}$, similar to the concept of *program counter* in the literature [15].

The typing judgment for expressions is $\Delta;\ \Gamma \vdash \mathtt{e} : \mathtt{t}\,!\,\epsilon$, which says that under the type context Δ and term context Γ, the expression $\mathtt{e}$ has the monadic effect type $\mathtt{t}\,!\,\epsilon$. The following are the typing rules for the new constructs:

$$\frac{\Delta;\ \Gamma \vdash \mathtt{m} : \mathtt{t}}{\Delta;\ \Gamma \vdash \mathtt{return}\ \mathtt{m} : \mathtt{t}\,!\,\mathtt{H}} \qquad \frac{\Delta;\ \Gamma \vdash \mathtt{m} : \mathtt{t}_1\,!\,\epsilon \quad \Delta;\ \Gamma, \mathtt{x}{:}\mathtt{t}_1 \vdash \mathtt{e} : \mathtt{t}_2\,!\,\epsilon}{\Delta;\ \Gamma \vdash \mathtt{run}\ \mathtt{x} = \mathtt{m}\ \mathtt{in}\ \mathtt{e} : \mathtt{t}_2\,!\,\epsilon}$$

$$\frac{\Delta;\ \Gamma \vdash \mathtt{e} : \mathtt{t}\,!\,\epsilon \quad \Delta \vdash \epsilon :: \mathcal{L}}{\Delta;\ \Gamma \vdash \mathtt{e}\,!\,\epsilon : \mathtt{t}\,!\,\epsilon} \qquad \frac{\Delta \vdash \ell \ll \mathtt{t} \quad \Delta \vdash \ell \preceq \epsilon}{\Delta \vdash \ell \ll \mathtt{t}\,!\,\epsilon}$$

The expression $\mathtt{return}\ \mathtt{m}$ has no effect and hence its type is given the empty effect $\mathtt{H}$. We interpret the effect at $\mathtt{H}$ to be visible to no-one, while the effect

at L to be visible to everyone. The expression run x = m in e executes the encapsulated effect of m, and then continue with e. Both m and e have the same effect type ϵ; otherwise, the subsumption rule of subtyping can be used.

The bottom left rule simply connects the typing judgments of terms and expressions. The bottom right rule, on the other hand, is an additional label protection judgment (defined in Sect. 2) for effect types. The rule says that the underlying type must protect the label and the computation must generate effects higher than the label. In other words, once the program has bound high-security data, it may not produce low observable effects.

Example 5 *The expression* run z = (bind y = x in if y then c!H else c!L) in z, *where* c ≡ return <> *and* x:bool{H}*, is insecure. This is a typical example of* implicit information flow through program counter *in the literature [15], where a program leaks information about a high-security Boolean through side effects.*

The evaluation judgment for expressions is $\mathtt{e} \xrightarrow{\epsilon} \mathtt{e}$, where ϵ is the side effect during such step. We use u to denote the values for expressions:

$$\mathtt{u} ::= \mathtt{return\ v} \qquad \mathtt{v} ::= \ldots \mid \mathtt{e!}\epsilon$$
$$\mathtt{E} ::= \ldots \mid \mathtt{return\ E} \mid \mathtt{run\ x = E\ in\ e} \mid \mathtt{run\ x = (return\ E)!}\epsilon\ \mathtt{in\ e}$$
$$\mathtt{run\ x = (return\ v)!}\epsilon\ \mathtt{in\ e} \xrightarrow{\epsilon} \mathtt{e\{v/x\}}$$

Since the congruence rules for expressions have no computational effects, we can still use evaluation contexts E to describe the evaluation order of expressions. The term e!ϵ is a value because it is a closure that delays computation.

Example 6 *The following expression of type* bool!L *evaluates as:*

$$\begin{aligned}
&\mathtt{run\ x = (run\ y = (return\ prj_2\ <true, false>)!L\ in\ return\ y)!H\ in\ return\ x}\\
\longrightarrow\ &\mathtt{run\ x = (run\ y = (return\ false)!L\ in\ return\ y)!H\ in\ return\ x}\\
\xrightarrow{L}\ &\mathtt{run\ x = (return\ y)\{false/y\}!H\ in\ return\ x}\\
=\ &\mathtt{run\ x = (return\ false)!H\ in\ return\ x}\\
\xrightarrow{H}\ &\mathtt{(return\ x)\{false/x\}!H}\\
=\ &\mathtt{return\ false}
\end{aligned}$$

□

To model that an observer can now distinguish programs due to computational effects, we need the following new equivalence judgments for effectful expressions and values: $\mathtt{e} \approx_\zeta \mathtt{e} : \mathtt{t!}\epsilon$ and $\mathtt{u} \sim_\zeta \mathtt{u} : \mathtt{t!}\epsilon$. The rules for expressions make use of a new evaluation relation, $\mathtt{e} \stackrel{\zeta}{\Longrightarrow}^{\mathtt{n}} \mathtt{u}$, which is explained below.

$$\frac{\mathtt{e}_1 \approx_\zeta \mathtt{e}_2 : \mathtt{t!}\epsilon}{\mathtt{e}_1\mathtt{!}\epsilon \sim_\zeta \mathtt{e}_2\mathtt{!}\epsilon : \mathtt{t!}\epsilon}\ \text{(R-Eff)} \qquad \frac{\mathtt{v}_1 \sim_\zeta \mathtt{v}_2 : \mathtt{t}}{\mathtt{return\ v}_1 \sim_\zeta \mathtt{return\ v}_2 : \mathtt{t!}\epsilon}\ \text{(R-Ret)}$$

$$\frac{\mathtt{e}_1 \stackrel{\zeta}{\Longrightarrow}^{\mathtt{n}} \mathtt{u}_1 \quad \mathtt{e}_2 \stackrel{\zeta}{\Longrightarrow}^{\mathtt{n}} \mathtt{u}_2 \quad \mathtt{u}_1 \sim_\zeta \mathtt{u}_2 : \mathtt{t!}\epsilon}{\mathtt{e}_1 \approx_\zeta \mathtt{e}_2 : \mathtt{t!}\epsilon}\ \text{(R-Exp)}$$

The rules on the top simply connect the term equivalence and the expression equivalence. Expressions are equivalent, the bottom rule says, if they produce the same number of effects visible to the observer and halt at equivalent values.

To formalize such equivalence, we first classify evaluation steps into those that are *visible* and those that are *invisible* to the observer. Then, a visible evaluation step can be prefixed and suffixed with any number of invisible evaluation steps.

$$\xrightarrow{\preceq\zeta} \equiv \bigcup_{\epsilon\preceq\zeta} \xrightarrow{\epsilon} \qquad \xrightarrow{\not\preceq\zeta} \equiv \bigcup_{\epsilon\not\preceq\zeta} \xrightarrow{\epsilon} \qquad \stackrel{\zeta}{\Longrightarrow} \equiv \xrightarrow{\not\preceq\zeta}{}^{*} \circ \xrightarrow{\preceq\zeta} \circ \xrightarrow{\not\preceq\zeta}{}^{*}$$

The evaluation judgment we want is therefore the $\mathtt{n}$-step closure $\stackrel{\zeta}{\Longrightarrow}{}^{\mathtt{n}}$ of $\stackrel{\zeta}{\Longrightarrow}$. Note that $\xrightarrow{\not\preceq\zeta}{}^{*} \circ \xrightarrow{\preceq\zeta}$ is the composition of the two relations, while $\xrightarrow{\not\preceq\zeta}{}^{*}$ is the reflexive and transitive closure of $\xrightarrow{\not\preceq\zeta}$.

Having refined our observer model as above, we proceed to proving noninterference for our core calculus with expressions. The main idea is to track the number of visible effects produced during the evaluation.

Note that the following proof is complete yet short in length. Since the proof is by induction on the typing derivation, monadic types allow an incremental proof, because the original proof for Theorem 3 remains valid and requires only a simple extension for $\mathtt{e!}\epsilon$. Here we can focus merely on the new typing rules for $\mathtt{return\ e}$ and $\mathtt{run\ x = m\ in\ e}$.

Theorem 7 (Noninterference for expressions) *If* $\Delta;\ \Gamma \vdash \mathtt{e} : \mathtt{t!}\epsilon$ *and* $\delta \models \Delta$ *and* $\gamma_1 \sim_\zeta \gamma_2 : \delta(\Gamma)$*, then* $\delta\gamma_1(\mathtt{e}) \approx_\zeta \delta\gamma_2(\mathtt{e}) : \delta(\mathtt{t})!\delta(\epsilon)$.

Proof. By mutual induction with Theorem 3 (extended with $\mathtt{e!}\epsilon$) on the typing derivation. Case `return`: We are given $\Delta;\ \Gamma \vdash \mathtt{return\ m} : \mathtt{t!H}$. By inversion, we have $\Delta;\ \Gamma \vdash \mathtt{m} : \mathtt{t}$. By Theorem 3, we have $\delta\gamma_1(\mathtt{m}) \approx_\zeta \delta\gamma_2(\mathtt{m}) : \delta(\mathtt{t})$. By inversion of R-Term, $\delta\gamma_1(\mathtt{m}) \longrightarrow^* \mathtt{v}_1$ and $\delta\gamma_2(\mathtt{m}) \longrightarrow^* \mathtt{v}_2$ and $\mathtt{v}_1 \sim_\zeta \mathtt{v}_2 : \delta(\mathtt{t})$. Therefore, by R-Ret, we conclude that $\delta\gamma_1(\mathtt{return\ m}) \approx_\zeta \delta\gamma_2(\mathtt{return\ m}) : \delta(\mathtt{t})!\delta(\epsilon)$.

Case `run`: We are given $\Delta;\ \Gamma \vdash \mathtt{run\ x = m\ in\ e} : \mathtt{t}_2\mathtt{!}\epsilon$. By inversion, we have $\Delta;\ \Gamma \vdash \mathtt{m} : \mathtt{t}_1\mathtt{!}\epsilon$ (*1) and $\Delta;\ \Gamma, \mathtt{x}{:}\mathtt{t}_1 \vdash \mathtt{e} : \mathtt{t}_2\mathtt{!}\epsilon$ (*2). By Theorem 3 with (*1), we have

$$\delta\gamma_1(\mathtt{m}) \approx_\zeta \delta\gamma_2(\mathtt{m}) : \delta(\mathtt{t}_1)!\delta(\epsilon)$$

By inversion of R-Term, $\delta\gamma_1(\mathtt{m}) \longrightarrow^* \mathtt{v}_1$ (*3) and $\delta\gamma_2(\mathtt{m}) \longrightarrow^* \mathtt{v}_2$ (*4) and $\mathtt{v}_1 \sim_\zeta \mathtt{v}_2 : \delta(\mathtt{t}_1)!\delta(\epsilon)$. By inversion of R-Eff, $\mathtt{v}_1 = \mathtt{e}_1!\delta(\epsilon)$ and $\mathtt{v}_2 = \mathtt{e}_2!\epsilon$ and $\mathtt{e}_1 \approx_\zeta \mathtt{e}_2 : \delta(\mathtt{t}_1)!\delta(\epsilon)$. By inversion of R-Exp, $\mathtt{e}_1 \stackrel{\zeta}{\Longrightarrow}{}^{\mathtt{n}} \mathtt{u}_1$ (*5) and $\mathtt{e}_2 \stackrel{\zeta}{\Longrightarrow}{}^{\mathtt{n}} \mathtt{u}_2$ (*6) and $\mathtt{u}_1 \sim_\zeta \mathtt{u}_2 : \delta(\mathtt{t}_1) : \delta(\epsilon)$. By inversion of R-Ret, $\mathtt{u}_1 = \mathtt{return\ v}_3\mathtt{!}\delta(\epsilon)$ and $\mathtt{u}_2 = \mathtt{return\ v}_4\mathtt{!}\delta(\epsilon)$. We then extend the term substitutions as

$$\gamma_1' = \gamma_1, \mathtt{x} \mapsto \mathtt{v}_3 \qquad \gamma_2' = \gamma_2, \mathtt{x} \mapsto \mathtt{v}_4$$

such that $\gamma_1' \sim_\zeta \gamma_2' : \delta(\Gamma, \mathtt{x}{:}\mathtt{t}_1)$ (*7). By induction hypothesis with (*2,*7),

$$\delta\gamma_1'(\mathtt{e}) \approx_\zeta \delta\gamma_2'(\mathtt{e}) : \delta(\mathtt{t}_2)!\delta(\epsilon)$$

which means that $\delta\gamma_1(\mathtt{e})\{\mathtt{v}_3/\mathtt{x}\} \approx_\zeta \delta\gamma_2(\mathtt{e})\{\mathtt{v}_4/\mathtt{x}\} : \delta(\mathtt{t}_2)!\delta(\epsilon)$ (*8). By (*3,*4,*5,*6),

$\delta\gamma_1(\mathtt{run\ x = m\ in\ e})$	$\delta\gamma_2(\mathtt{run\ x = m\ in\ e})$
$= \mathtt{run\ x} = \delta\gamma_1(\mathtt{m})\ \mathtt{in}\ \delta\gamma_1(\mathtt{e})$	$= \mathtt{run\ x} = \delta\gamma_2(\mathtt{m})\ \mathtt{in}\ \delta\gamma_2(\mathtt{e})$
$\longrightarrow^* \mathtt{run\ x} = \mathtt{e}_1!\delta(\epsilon)\ \mathtt{in}\ \delta\gamma_1(\mathtt{e})$	$\longrightarrow^* \mathtt{run\ x} = \mathtt{e}_2!\delta(\epsilon)\ \mathtt{in}\ \delta\gamma_2(\mathtt{e})$
$\stackrel{\zeta}{\Longrightarrow}{}^{\mathtt{n}} \mathtt{run\ x} = (\mathtt{return\ v}_3)!\delta(\epsilon)\ \mathtt{in}\ \delta\gamma_1(\mathtt{e})$	$\stackrel{\zeta}{\Longrightarrow}{}^{\mathtt{n}} \mathtt{run\ x} = (\mathtt{return\ v}_4)!\delta(\epsilon)\ \mathtt{in}\ \delta\gamma_2(\mathtt{e})$
$\xrightarrow{\epsilon} \delta\gamma_1(\mathtt{e})\{\mathtt{v}_3/\mathtt{x}\}$	$\xrightarrow{\epsilon} \delta\gamma_2(\mathtt{e})\{\mathtt{v}_4/\mathtt{x}\}$

That means

$$\delta\gamma_1(\texttt{run x} = \texttt{m in e}) \stackrel{\zeta}{\Longrightarrow}^{\texttt{n}+\texttt{i}} \delta\gamma_1(\texttt{e})\{\texttt{v}_3/\texttt{x}\}$$
$$\delta\gamma_2(\texttt{run x} = \texttt{m in e}) \stackrel{\zeta}{\Longrightarrow}^{\texttt{n}+\texttt{i}} \delta\gamma_2(\texttt{e})\{\texttt{v}_4/\texttt{x}\}$$

where $\texttt{i} = 1$ if $\epsilon \preceq \zeta$, or $\texttt{i} = 0$ otherwise. By R-Exp and (*8), we conclude that $\delta\gamma_1(\texttt{run x} = \texttt{m in e}) \approx_\zeta \delta\gamma_2(\texttt{run x} = \texttt{m in e}) : \delta(\texttt{t}_2)!\delta(\epsilon)$. □

3 Decentralized label calculus

Having established the security property of our core calculus, we now investigate how to make the policy sublanguage more expressive. The key challenge is to extend the policy language in a modular way, reusing the type machinery from the core as much as possible.

This section shows how the decentralized label model by Myers and Liskov [10] can be integrated into our core label calculus. Decentralized labels allow different *principals* to individually specify fine-grained security policies such as *confidentiality* and *integrity*. Combined with *singleton types*, this extended calculus draws a connection between compile-time dependency analyses and the run-time infrastructure. The benefit is twofold: (1) security policies can now be specified in term of information not known until execution, such as run-time user identities or file access permissions; (2) certificates can be used to regulate declassification and to justify a conditioned version of the noninterference theorem.

3.1 Confidentiality and integrity

Confidentiality policies specify which principals allow which other principals to *read* some data, while integrity policies specify which principals *trust* some data [7]. These policy constructors, or *label constructors*, provide a finer-grained control of security specification than the *label constants* introduced in Sect. 2.1.

To model these policies, we treat principals $\texttt{p}$ as abstract types and treat principal delegation $\texttt{p}_1 \preceq \texttt{p}_2$ as subtyping. That is, $\texttt{p}_1$ is a subtype of $\texttt{p}_2$ whenever $\texttt{p}_1$ *delegates to* $\texttt{p}_2$ (or, $\texttt{p}_2$ is *acting for* $\texttt{p}_1$). We also introduce two new label constructors, $\texttt{R}$ (read) and $\texttt{T}$ (trust), for confidentiality and integrity:

$$\ell ::= \ldots \mid \texttt{R p p} \mid \texttt{T p} \mid \ell \wedge \ell \mid \ell \vee \ell \qquad \textcircled{S} ::= \ldots \mid \texttt{R} \oplus \oplus \mid \texttt{T} \ominus \mid \oplus \wedge \oplus \mid \oplus \vee \oplus$$

A label $\texttt{R}\,\texttt{p}_1\,\texttt{p}_2$ specifies the policy that a data is owned by $\texttt{p}_1$ and that $\texttt{p}_1$ allows $\texttt{p}_2$ to read the data, while a label $\texttt{T p}$ specifies that the data is trusted by $\texttt{p}$.

Moreover, we add intersection $\ell \wedge \ell$ and union types $\ell \vee \ell$ [3] to precisely model policy sets. Since labels ℓ and principals $\texttt{p}$ are in the same syntactic class, these two constructors can also model principal groups as $\texttt{p} \wedge \texttt{p}$ and $\texttt{p} \vee \texttt{p}$.

Intersection and union types in this paper are used only for labels, principals, and privileges, but not for ordinary types; hence, our language does not have introduction or elimination terms for intersections and unions. This decision helps keeping the static and the dynamic semantics of our language simple.

We need both intersection and union types because the two label constructors have different subtyping polarities: R is covariant, while T is contravariant. Having both intersections and unions gives a natural interpretation of principal sets:

$$\begin{aligned}\mathtt{R}\ [\mathtt{p}_1,\mathtt{p}_2,\ldots,\mathtt{p}_n]\ \mathtt{p} &= \mathtt{R}\ (\mathtt{p}_1 \wedge \mathtt{p}_2 \wedge \ldots \wedge \mathtt{p}_n)\ \mathtt{p}\\ \mathtt{R}\ \mathtt{p}\ [\mathtt{p}_1,\mathtt{p}_2,\ldots,\mathtt{p}_n] &= \mathtt{R}\ \mathtt{p}\ (\mathtt{p}_1 \wedge \mathtt{p}_2 \wedge \ldots \wedge \mathtt{p}_n)\\ \mathtt{T}\ [\mathtt{p}_1,\mathtt{p}_2,\ldots,\mathtt{p}_n] &= \mathtt{T}\ (\mathtt{p}_1 \vee \mathtt{p}_2 \vee \ldots \vee \mathtt{p}_n)\end{aligned}$$

Example 8 *The data* $\mathtt{true}\{\mathtt{R}\ \mathtt{p}_1\ [\mathtt{p}_2,\mathtt{p}_3]\}\{\mathtt{T}\ [\mathtt{p}_1,\mathtt{p}_2]\}$ *has two security policies. The first one is a confidentiality policy saying that the data is owned by* $\mathtt{p}_1$*, and that* $\mathtt{p}_1$ *allows* $\mathtt{p}_2$ *and* $\mathtt{p}_3$ *to read the data. The second one is an integrity policy saying that both* $\mathtt{p}_1$ *and* $\mathtt{p}_2$ *trust the data.* □

A decentralized label looks like $\{\mathtt{p}_1 : \mathtt{p}_2,\mathtt{p}_3;\ \mathtt{p}_2 : \mathtt{p}_3\ !\ \mathtt{p}_1,\mathtt{p}_2\}$ traditionally [10,18], compared to our notation $\{\mathtt{R}\ \mathtt{p}_1\ [\mathtt{p}_2,\mathtt{p}_3]\}\{\mathtt{R}\ \mathtt{p}_2\ \mathtt{p}_3\}\{\mathtt{T}\ [\mathtt{p}_1\ \mathtt{p}_2]\}$ here. Ours is slightly more verbose but its semantics can be specified more easily in terms of subtyping. In addition, new policy constructors can be added in a uniform way by simply specifying their subtyping polarities.

3.2 Downgrading as subtyping

The rest of the section discusses various downgrading mechanisms that intentionally leak information [21]. These mechanisms include:

1. *declassifying* some data to a lower label,
2. a principal *delegating* to other principals,
3. a principal *declassifying* some data to other principals for reading, and
4. a principal *endorsing* the integrity of some data.

The decentralized label model is essential in the last three mechanisms because each concerns a particular *principal*. In Sect. 3.3 we will see how a public key, which represents the concerned principal, can be used to verify a digital certificate, which represents the authority for downgrading.

The innovation here is to model downgrading as subtyping. The motivation is that downgrading can be made *implicit* through the subsumption rule of subtyping, if the concerned principal *explicitly* introduces the authority into the context. This contrasts with the usual approach [10] that uses coercion constructs like $\mathtt{declassify}_{\mathtt{p}}\ \mathtt{m}$ and $\mathtt{endorse}_{\mathtt{p}}\ \mathtt{m}$ for declassification and endorsement. Both approaches ensure that the authority of the concerned principal is granted before declassification. Our implicit approach, however, allows a simple formulation of certificate-based declassification (to be shown in Sect. 3.3).

Foremost, we extend the type context Δ to maintain authority, which is a set of authorizations of the form $\mathtt{p} \lhd \mathtt{j}$ (a principal p *granting* some privilege j):

$$\begin{array}{ll}\Delta ::= \ldots \mid \Delta, \mathtt{p} \lhd \mathtt{j} & \\ \mathtt{j} ::= \ldots \mid \mathtt{del\ p} \mid \mathtt{dcls\ p} \mid \mathtt{endr} & \textcircled{S} ::= \ldots \mid \mathtt{del} \oplus \mid \mathtt{dcls} \oplus \mid \oplus\ \mathtt{\%} \ominus \lhd \oplus \\ \mathtt{t} ::= \ldots \mid \mathtt{t\ \%\ p} \lhd \mathtt{j} & \mathtt{m} ::= \ldots \mid \mathtt{grant\ p} \lhd \mathtt{j\ in\ m} \mid \mathtt{pass\ x} = \mathtt{m\ in\ m}\end{array}$$

The three predefined privileges are delegation (del p), declassification (dcls p), and endorsement (endr), corresponding to downgrading for principal subtyping,

confidentiality and integrity, respectively. Now, the downgrading mechanisms can be concisely expressed using these additional subtyping rules:

$$\frac{\Delta \vdash \mathtt{p}_1 \lhd \mathtt{del}\ \mathtt{p}_2}{\Delta \vdash \mathtt{p}_1 \preceq \mathtt{p}_2} \qquad \frac{\Delta \vdash \mathtt{p}_1 \lhd \mathtt{dcls}\ \mathtt{p}}{\Delta \vdash \mathtt{R}\ \mathtt{p}_1\ \mathtt{p}_2 \preceq \mathtt{R}\ \mathtt{p}_1\ [\mathtt{p}_2, \mathtt{p}]} \qquad \frac{\Delta \vdash \mathtt{p} \lhd \mathtt{endr}}{\Delta \vdash \mathtt{T}\ \mathtt{p}_1 \preceq \mathtt{T}\ [\mathtt{p}_1, \mathtt{p}]}$$

Authority types $\mathtt{t}\ \%\ \mathtt{p} \lhd \mathtt{j}$ track the *effects* of declassification on the lattice so that later theorems can be stated in terms of the authority:

$$\frac{\Delta, \mathtt{p} \lhd \mathtt{j};\ \Gamma \vdash \mathtt{m} : \mathtt{t} \quad \Delta \vdash \mathtt{p} :: \mathcal{P} \quad \Delta \vdash \mathtt{j} :: \mathcal{J}}{\Delta;\ \Gamma \vdash \mathtt{grant}\ \mathtt{p} \lhd \mathtt{j}\ \mathtt{in}\ \mathtt{m} : \mathtt{t}\ \%\ \mathtt{p} \lhd \mathtt{j}} \qquad \frac{\begin{array}{c}\Delta;\ \Gamma \vdash \mathtt{m}_1 : \mathtt{t}_1\ \%\ \mathtt{p} \lhd \mathtt{j} \\ \Delta, \mathtt{p} \lhd \mathtt{j};\ \Gamma, \mathtt{x}{:}\mathtt{t}_1 \vdash \mathtt{m}_2 : \mathtt{t}_2\end{array}}{\Delta;\ \Gamma \vdash \mathtt{pass}\ \mathtt{x} = \mathtt{m}_1\ \mathtt{in}\ \mathtt{m}_2 : \mathtt{t}_2\ \%\ \mathtt{p} \lhd \mathtt{j}}$$

$$\mathtt{v} ::= \ldots \mid \mathtt{grant}\ \mathtt{p} \lhd \mathtt{j}\ \mathtt{in}\ \mathtt{v} \qquad \mathtt{E} ::= \ldots \mid \mathtt{grant}\ \mathtt{p} \lhd \mathtt{j}\ \mathtt{in}\ \mathtt{E} \mid \mathtt{pass}\ \mathtt{x} = \mathtt{E}\ \mathtt{in}\ \mathtt{e}$$

$$\mathtt{pass}\ \mathtt{x} = (\mathtt{grant}\ \mathtt{p} \lhd \mathtt{j}\ \mathtt{in}\ \mathtt{v})\ \mathtt{in}\ \mathtt{m} \longrightarrow \mathtt{grant}\ \mathtt{p} \lhd \mathtt{j}\ \mathtt{in}\ \mathtt{m}\{\mathtt{v}/\mathtt{x}\}$$

These rules are very close to the typing and evaluation rules for standard monadic types, except that the type context Δ is now extended with $\mathtt{p} \lhd \mathtt{j}$. The value $\mathtt{v}$ in the term $\mathtt{grant}\ \mathtt{p} \lhd \mathtt{j}\ \mathtt{in}\ \mathtt{v}$ may capture the constraint $\mathtt{p} \lhd \mathtt{j}$, and hence, to ensure type preservation, $\mathtt{grant}\ \mathtt{p} \lhd \mathtt{j}\ \mathtt{in}\ \mathtt{v}$ is regarded together as a value.

As a pleasant bonus of monadic analysis, checking the *robustness condition* of downgrading reduces to adding one condition in the label protection rule $\Delta \vdash \ell \ll \mathtt{t}$ in Sect. 2.1. In particular, robust declassification says that the program context of a declassification operation should be trusted by the owner of the data [21]. The following rule generalizes the robustness condition to any downgrading mechanism. The intuition is that, for robust downgrading, when $\mathtt{p}_2$ authorizes some privilege $\mathtt{j}$, the program context should have trust $(\mathtt{T}\ \mathtt{p}_1)$ higher than $\mathtt{p}_2$'s trust $(\mathtt{T}\ \mathtt{p}_2)$. That is $\Delta \vdash \mathtt{T}\ \mathtt{p}_2 \preceq \mathtt{T}\ \mathtt{p}_1$, or equivalently, $\Delta \vdash \mathtt{p}_1 \preceq \mathtt{p}_2$.

$$\frac{\Delta \vdash \mathtt{T}\ \mathtt{p}_1 \ll \mathtt{t} \quad \Delta \vdash \mathtt{p}_1 \preceq \mathtt{p}_2}{\Delta \vdash \mathtt{T}\ \mathtt{p}_1 \ll (\mathtt{t}\ \%\ \mathtt{p}_2 \lhd \mathtt{j})}$$

It is known that noninterference does not hold in the presence of downgrading [15]. Yet, it is intuitive that if the program does not use any downgrading, the program should still be secure. In fact, a slightly stronger statement holds: if no one transitively downgrades to the observer, the program is still secure.

The following modified theorem of noninterference formally captures such intuition. We write $\Delta = \Delta_\alpha, \Delta_\lhd$ to separate the bindings and the authority, and we write $\mathtt{t} \Rightarrow \mathtt{t}_0\ \%\ \Delta$ to collect all required authority in the value positions of the type. For example, $\mathtt{p}_1, \mathtt{j}, \mathtt{p}_1 \lhd \mathtt{j}, \mathtt{p}_2 \vdash \mathtt{m} : \mathtt{bool} \to (\mathtt{bool}\ \%\ \mathtt{p}_1 \lhd \mathtt{j})$ has $\Delta_\alpha = \mathtt{p}_1, \mathtt{j}, \mathtt{p}_2$ and $\Delta_\lhd = \mathtt{p}_1 \lhd \mathtt{j}$ and $\mathtt{bool} \to (\mathtt{bool}\ \%\ \mathtt{p}_1 \lhd \mathtt{j}) \Rightarrow (\mathtt{bool} \to \mathtt{bool})\ \%\ \mathtt{p}_1 \lhd \mathtt{j}$. These straight-forward rules are defined in our technical report [17].

Theorem 9 (Conditioned noninterference) *Suppose* $\Delta;\ \Gamma \vdash \mathtt{m} : \mathtt{t}$*, where* $\Delta = \Delta_\alpha, \Delta_\lhd$ *and* $\mathtt{t} \Rightarrow \mathtt{t}_0\ \%\ \Delta_0$*, and* $\delta \models \Delta_\alpha$ *and* $\gamma_1 \sim_\zeta \gamma_2 : \delta(\Gamma)$*. If* $\Delta, \Delta_0 \not\vdash \mathtt{p} \preceq \zeta$ *for all* $\mathtt{p} \in \mathtt{dom}(\Delta_\alpha)$ *such that* $\Delta \not\vdash \mathtt{p} \preceq \zeta$*, then* $\delta\gamma_1(\mathtt{m}) \approx_\zeta \delta\gamma_2(\mathtt{m}) : \delta(\mathtt{t})$*.*

Proof. By induction on the typing derivation. Case $\lambda\mathtt{x}{:}\mathtt{t}_1.\mathtt{m}$: We are given $\Delta;\ \Gamma \vdash \lambda\mathtt{x} : \mathtt{t}_1.\mathtt{m} : \mathtt{t}_1 \to \mathtt{t}_2$. By inversion, we have $\Delta;\ \Gamma, \mathtt{x} : \mathtt{t}_1 \vdash \mathtt{m} : \mathtt{t}_2$ (*1). Since

downgrading in the input propagates to the output in a function, we have $\mathtt{t}_2 \Rightarrow \mathtt{t}_3 \,\%\, \Delta_0$ (*2) for the same Δ_0 as in $\mathtt{t}_1 \to \mathtt{t}_2 \Rightarrow \mathtt{t}_0 \,\%\, \Delta_0$. Assume $\mathtt{v}_3 \sim_\zeta \mathtt{v}_4 : \delta(\mathtt{t}_1)$. We then extend the term substitutions as

$$\gamma_1' = \gamma_1, \mathtt{x} \mapsto \mathtt{v}_3 \qquad\qquad \gamma_2' = \gamma_2, \mathtt{x} \mapsto \mathtt{v}_4$$

such that $\gamma_1' \sim_\zeta \gamma_2' : \delta(\Gamma, \mathtt{x}{:}\mathtt{t}_1)$ (*3). By induction hypothesis with (*1,*2,*3),

$$\delta\gamma_1'(\mathtt{m}) \approx_\zeta \delta\gamma_2'(\mathtt{m}) : \delta(\mathtt{t}_2)$$

which, by R-Term, $\delta\gamma_1(\mathtt{m})\{\mathtt{v}_3/\mathtt{x}\} \approx_\zeta \delta\gamma_2(\mathtt{m})\{\mathtt{v}_4/\mathtt{x}\} : \delta(\mathtt{t}_2)$. By R-Term again,

$$\delta\gamma_1(\lambda \mathtt{x}{:}\mathtt{t}.\mathtt{m})\ \mathtt{v}_3 \approx_\zeta \delta\gamma_2(\lambda \mathtt{x}{:}\mathtt{t}.\mathtt{m})\ \mathtt{v}_4 : \delta(\mathtt{t}_2)$$

Hence, by R-Fun, $\delta\gamma_1(\lambda \mathtt{x}{:}\mathtt{t}.\mathtt{m}) \approx_\zeta \delta\gamma_2(\lambda \mathtt{x}{:}\mathtt{t}.\mathtt{m}) : \delta(\mathtt{t}_1 \to \mathtt{t}_2)$. □

The proof is surprisingly similar to that for standard noninterference, showing that this conditioned version is only a slight generalization. In the next subsection, however, we will show how combining this theorem with ideas from public-key infrastructures justifies certificate-based declassification.

3.3 Public keys and certificates

Public-key infrastructures provide public keys and digital certificates for distributed access control. Our motivation here is to connect the type system with the security infrastructure such that a certificate of authority, when verified with a principal's public key, can justify the information leaks due to downgrading. Certificates are also important for auditing purpose.

In our previous work [18], we presented the language λ_{RP} for specifying security policies with run-time principals. The type system uses *singleton types* to represent run-time principals and an abstract type to represent certificates. Effectively, λ_{RP} models public keys and certificate verifications of public-key infrastructures in a sound type system.

Allowing such run-time principals gives programmers more flexibility in specifying security policies. Together with universal and existential quantification, programs can determine the run-time user identity of the system (`getuid`) and write functions polymorphic in principals (`getenv`). Here the type $\top_{\mathcal{P}}$ represents the top principal, and $'\alpha$ is a singleton type to be explained below.

$$\begin{array}{lcl} \mathtt{getuid} & : & () \to \exists \alpha \preceq \top_{\mathcal{P}}.\ '\alpha \\ \mathtt{getenv} & : & \forall \alpha \preceq \top_{\mathcal{P}}.\ '\alpha \to \mathtt{string}\{\mathtt{R}\ \alpha\ \alpha\} \end{array}$$

Our language readily generalizes the idea of *run-time types* to run-time labels and run-time privileges as well. For example, access permissions from the file system (`fstat`) can be used as run-time labels to constrain the information flow of data read from a file.

Let us recap our previous work on run-time principals [18]:

$$\mathtt{t} ::= \ldots \mid \mathtt{'p} \mid \mathtt{'j} \mid \mathtt{cert} \qquad \mathtt{m} ::= \ldots \mid \mathtt{'p} \mid \mathtt{'j} \mid \mathtt{'p} \lhd \mathtt{'j} \mid \mathtt{if\ (m} \Rightarrow \mathtt{m} \lhd \mathtt{m)\ m\ m}$$

The term $\texttt{'p}$ is the run-time representation of principal $\texttt{p}$ and has the singleton type $\texttt{'p}$, carrying the most precise information about the term in the type system. Similarly, $\texttt{'j}$ is the run-time representation of privilege $\texttt{j}$.

The term $\texttt{'p} \triangleleft \texttt{'j}$ represents the authority of principal $\texttt{p}$ granting privilege $\texttt{j}$. Such term has the abstract type $\texttt{cert}$ that does not reveal any information at all at the type level. The reason is that we do not trust the validity of the certificate unless verified with the term $\texttt{if (}\mathtt{m}_1 \Rightarrow \mathtt{m}_2 \triangleleft \mathtt{m}_3\texttt{)}\ \mathtt{m}_4\ \mathtt{m}_5$. More formally:

$$\Delta;\ \Gamma \vdash \texttt{'p} : \texttt{'p} \qquad \Delta;\ \Gamma \vdash \texttt{'j} : \texttt{'j} \qquad \Delta;\ \Gamma \vdash \texttt{'p} \triangleleft \texttt{'j} : \texttt{cert}$$

$$\frac{\Delta;\ \Gamma \vdash \mathtt{m}_1 : \texttt{cert} \quad \Delta;\ \Gamma \vdash \mathtt{m}_2 : \texttt{'p} \quad \Delta;\ \Gamma \vdash \mathtt{m}_3 : \texttt{'j} \quad \Delta, \mathtt{p} \triangleleft \mathtt{j};\ \Gamma \vdash \mathtt{m}_4 : \mathtt{t} \quad \Delta;\ \Gamma \vdash \mathtt{m}_5 : \mathtt{t}}{\Delta;\ \Gamma \vdash \texttt{if (}\mathtt{m}_1 \Rightarrow \mathtt{m}_2 \triangleleft \mathtt{m}_3\texttt{)}\ \mathtt{m}_4\ \mathtt{m}_5 : \mathtt{t}\ \texttt{\%}\ \mathtt{p} \triangleleft \mathtt{j}}$$

$$\frac{\vdash \texttt{'}\mathtt{p}_1 \triangleleft \texttt{'}\mathtt{j}_1 \Rightarrow \texttt{'}\mathtt{p}_2 \triangleleft \texttt{'}\mathtt{j}_2}{\texttt{if ('}\mathtt{p}_1 \triangleleft \texttt{'}\mathtt{j}_1 \Rightarrow \texttt{'}\mathtt{p}_2 \triangleleft \texttt{'}\mathtt{j}_2\texttt{)}\ \mathtt{m}_1\ \mathtt{m}_2 \longrightarrow \texttt{grant}\ \mathtt{p}_2 \triangleleft \mathtt{j}_2\ \texttt{in}\ \mathtt{m}_1}$$

The judgment $\vdash \texttt{'}\mathtt{p}_1 \triangleleft \texttt{'}\mathtt{j}_1 \Rightarrow \texttt{'}\mathtt{p}_2 \triangleleft \texttt{'}\mathtt{j}_2$ defines an external verification procedure of the authority with respect to the principal and the privilege. If the procedure fails, the term $\texttt{if ('}\mathtt{p}_1 \triangleleft \texttt{'}\mathtt{j}_1 \Rightarrow \texttt{'}\mathtt{p}_2 \triangleleft \texttt{'}\mathtt{j}_2\texttt{)}\ \mathtt{m}_1\ \mathtt{m}_2$ steps to $\mathtt{m}_2$.

There exists a direct mapping from the language constructs ($\texttt{'p}$, $\texttt{'p} \triangleleft \texttt{'j}$, and $\vdash \texttt{'}\mathtt{p}_1 \triangleleft \texttt{'}\mathtt{j}_1 \Rightarrow \texttt{'}\mathtt{p}_2 \triangleleft \texttt{'}\mathtt{j}_2$) to the mechanisms of public-key infrastructures (public keys and digital certificates). In fact, public-key infrastructures are just one possible implementation that supports distributed access control [4]. Our previous work [18] carries out the design and the proof in an abstract setting and provides constructs for testing delegation and acquiring certificates.

We conclude the development of our language by presenting a modified theorem of noninterference. It states that any information leaked by a well-typed program can be justified by certificates in the environment. In fact, the theorem is simply the contrapositive of the conditioned noninterference in Sect. 3.2. Our technical report [17] contains detailed proofs of the type-safety and the following theorem for the full language.

Theorem 10 (Certified noninterference) *Suppose* $\Delta;\ \Gamma \vdash \mathtt{m} : \mathtt{t}$, *where* $\Delta = \Delta_\alpha, \Delta_\triangleleft$ *and* $\mathtt{t} \Rightarrow \mathtt{t}_0\ \texttt{\%}\ \Delta_0$, *and* $\delta \models \Delta_\alpha$ *and* $\gamma_1 \sim_\zeta \gamma_2 : \delta(\Gamma)$. *If* $\delta\gamma_1(\mathtt{m}) \not\approx_\zeta \delta\gamma_2(\mathtt{m}) : \delta(\mathtt{t})$, *then* $\mathtt{t} = \mathtt{t}_0\ \texttt{\%}\ \Delta_0$ *and* $\Delta, \Delta_0 \vdash \mathtt{p} \preceq \zeta$ *for some* $\mathtt{p}$ *that satisfies* $\Delta \not\vdash \mathtt{p} \preceq \zeta$.

Proof. By Theorem 9, extended with singletons and certificates.

4 Related work

The survey by Sabelfeld and Myers [15] on language-based information-flow security is an excellent introduction to the field. In particular, their paper cites a long line of research [10,13,2] that studies the interactions of security policies and language features in Java and ML. This paper instead focuses on a smaller set of interesting features with a modular design and with the goal of justifying declassification with certificates. Compared to our previous work [18], this paper

concisely expresses the decentralized label model in the polymorphic lambda calculus with subtyping ($\mathtt{F}_{\preceq}$). Various downgrading mechanisms are understood as subtyping such that not only type-safety but also a conditioned version of noninterference can be formulated and proved. We also extensively employ monadic constructs [9,1,5] to keep the design and the proofs modular. As a future work, one may check if these constructs satisfy some formal monad laws.

Chothia et al. also use public-key infrastructures to model typed cryptographic operations for distributed access control [4]. Strecker [16] formalizes an analysis of information flow for μ-Java and proves noninterference in Isabelle by shallow embedding, while Naumann [11] similarly formalizes a core subset of Java in PVS by deep embedding. Our ongoing work has the same goal of proving noninterference in a machine-checkable way.

It is known that standard noninterference does not hold in the presence of declassification [15]. Hence, it has been a challenging problem to formulate and prove *any* variant of noninterference with declassification. Various ideas such as *selective declassification* [13], *delimited release* [14], and *relaxed noninterference* [8] are proposed to allow downgrading that can be externally justified.

5 Conclusion

We have presented the design of a safe and secure information-flow type system with bounded quantification and effects in a monadic style. One of our design decisions is to treat labels, principals and privileges uniformly, as they are all abstract types necessary only for compile-time analyses. This treatment allows reuse of type machinery such as polymorphism, subtyping, and singleton types, keeping the calculus consistent yet general.

The integration of the decentralized label model with type constructors allows programmers specify expressive policies, while the use of subtyping to model delegation, declassification, and endorsement simplifies the semantics of downgrading. More importantly, these simplifications lead to a conditioned version of the noninterference theorem that justifies certificate-based downgrading.

Formalizing the full language semantics and security theorems is our long-term goal of building a rigid foundation for security-typed languages. One exciting future work is to use Twelf (a logical framework) to mechanically formalize and check the various noninterference theorems presented in this paper. We are also writing larger examples in our language interpreter to gain more experience of monadic secure programming.

Acknowledgments

The authors thank Peng Li, Nitin Khandelwal, Eijiro Sumii, and the anonymous reviewers for their comments on drafts of this paper. This research was supported in part by NSF grant CCR-0311204 *(Dynamic Security Policies)* and NSF grant CNS-0346939 *(CAREER: Language-based Distributed System Security)*.

References

1. Martin Abadi, Anindya Banerjee, Nevin Heintze, and Jon G. Riecke. A Core Calculus of Dependency. In *ACM Symposium on Principles of Programming Languages*, 1999.
2. Anindya Banerjee and David A. Naumann. Secure Information Flow and Pointer Confinement in a Java-like Language. In *Computer Security Foundations Workshop*, 2002.
3. Franco Barbanera, Mariangiola Dezani-Ciancaglini, and Ugo de'Liguoro. Intersection and Union Types: Syntax and Semantics. *Information and Computation*, 119, 1995.
4. Tom Chothia, Dominic Duggan, and Jan Vitek. Type-Based Distributed Access Control. In *Computer Security Foundations Workshop*, 2003.
5. Karl Crary, Aleksey Kliger, and Frank Pfenning. A monadic analysis of info flow security with mutable state. In *Foundations of Computer Security*, 2004.
6. Pierre-Louis Curien and Giorgio Ghelli. Coherence of subsumption, minimum typing and type-checking in Fsub. *Mathematical Structures in Computer Science*, 1992.
7. Peng Li, Yun Mao, and Steve Zdancewic. Information Integrity Policies. In *Proceedings of the Workshop on Formal Aspects in Security and Trust (FAST)*, 2003.
8. Peng Li and Steve Zdancewic. Downgrading Policies and Relaxed Noninterference. In *ACM Symposium on Principles of Programming Languages*, 2004.
9. Eugenio Moggi. Computational Lambda-Calculus and Monads. In *IEEE Symposium on Logic in Computer Science*, 1989.
10. Andrew C. Myers and Barbara Liskov. A Decentralized Model for Information Flow Control. In *ACM Symposium on Operating Systems Principles*, 1997.
11. David A. Naumann. Machine-checked correctness of a secure information flow analyzer. Technical Report CS-2004-10, Stevens Institute of Technology, 2004.
12. Andrew Pitts. Existential Types: Logical Relations and Operational Equivalence. In *International Colloquium on Automata, Languages and Programming*, 1998.
13. Francois Pottier and Vincent Simonet. Information flow inference for ML. In *ACM Symposium on Principles of Programming Languages*, 2002.
14. Andrei Sabelfeld and Andrew C. Myers. A Model for Delimited Release. In *International Symposium on Software Security*, 2003.
15. Andrei Sabelfeld and Andrew C. Myers. Language-Based Information-Flow Security. *IEEE Journal on Selected Areas in Communications*, 21(1), 2003.
16. Martin Strecker. Formal Analysis of an Information Flow Type System for MicroJava. Technical report, Technische Universitat Munchen, 2003.
17. Stephen Tse and Steve Zdancewic. Certificate-based Declassification. Technical Report MS-CIS-04-16, University of Pennsylvania, 2004.
18. Stephen Tse and Steve Zdancewic. Run-time Principals in Information-flow Type Systems. In *IEEE Symposium on Security and Privacy*, 2004.
19. Philip Wadler. Theorems for Free! In *Functional Programming Languages and Computer Architecture*, 1989.
20. Andrew K. Wright and Matthias Felleisen. A Syntactic Approach to Type Soundness. *Information and Computation*, 115(1), 1994.
21. Steve Zdancewic. *Programming Languages for Information Security*. PhD thesis, Cornell University, 1997.