FABLE: A Language for Enforcing User-defined Security Policies

Nikhil Swamy Brian J. Corcoran Michael Hicks
{nswamy, bjc, mwh}@cs.umd.edu
Technical Report: CS-TR-4895
Department of Computer Science, University of Maryland, College Park

Abstract

This paper presents FABLE*, a core formalism for a programming language in which programmers may specify security policies and reason that these policies are properly enforced. In* FABLE*, security policies can be expressed by associating* security labels *with the data or actions they protect. Programmers define the semantics of labels in a separate part of the program called the* enforcement policy. FABLE *prevents a policy from being circumvented by allowing labeled terms to be manipulated only within the enforcement policy; application code must treat labeled values abstractly. Together, these features facilitate straightforward proofs that programs implementing a particular policy achieve their high-level security goals.* FABLE *is flexible enough to implement a wide variety of security policies, including access control, information flow, provenance, and security automata. We have implemented* FABLE *as part of the* LINKS *web programming language; we call the resulting language* SELINKS*. We report on our experience using* SELINKS *to build two substantial applications, a wiki and an on-line store, equipped with a combination of access control and provenance policies. To our knowledge, no existing framework enables the enforcement of such a wide variety of security policies with an equally high level of assurance.*

1 Introduction

For 35 years or more, computer security researchers have explored techniques for ensuring that a software system correctly enforces its security policy, and that, as a result, the software exhibits a desirable security property [31, 25]. A notable success toward this goal has been work on defining programming language-based techniques for enforcing *information flow* security policies [36]. A common form of information flow policy defines a set of security levels that can be ordered as a lattice, where sensitive data within a program is assigned a label derived from this lattice [17]. Correct enforcement of this policy implies that a program exhibits some flavor of *noninterference*, which states that no information visible at level h can be leaked onto a channel visible to level $l < h$. By including the notion of security label in a programming language's types, one can show that a correctly-typed program is certain to enforce its security policy [43]. This approach has been implemented successfully in the Jif [13] and FlowCaml [33] languages.

While information flow policies are useful and important, there are many other styles of policy that are in common use, including access control, type enforcement [5] (as in SELinux [27]), tainting [39, 40] (as via Perl's *taint mode* [32]), provenance tracking [9], stack inspection [21, 18], and forms of security automata [21, 45]. One approach to verifying the correct enforcement of these policies is to encode them as information flow policies for programs written in Jif of FlowCaml. While this will work in some cases (e.g., access control, type enforcement, and tainting could be encoded in conjunction with Jif's declassification operators [37]) it is not likely to scale. For example, Jif's use of noninterference as a baseline property and its attendant restriction of *implicit flows* via the program's control flow, can be cumbersome to work with. Moreover, Jif and FlowCaml fix the format of security labels, which complicates the means to interface with external infrastructure, such as policy management systems, databases, etc.

What we want is a programming language that can enforce a wide range of policies—including, but not limited to, information flow—while providing the same assurance as Jif or FlowCaml that programs enforce their policies correctly. As a step toward this goal, this paper presents FABLE, a core language for writing programs that enforce a variety of security policies. A key observation is that many security policies work by associating labels with data,

where the label expresses the security policy for that data. What varies among policies is the *specification* and *interpretation* of labels, in terms of what actions are permitted or denied.

This observation is embodied in FABLE in two respects. First, programmers can define custom security labels and associate them with the data they protect using dependent types. For example, a programmer could define a label *LOW*, and an integer value protected by this label would have type int{*LOW*}. As another example, the programmer could define a label *ACL*(*Alice*, *Bob*) where an integer with type int{*ACL*(*Alice*, *Bob*)} is meant to be accessed by only *Alice* or *Bob*. Second, programmers define the interpretation of labels in special *enforcement policy* functions separated from the rest of the program. For example, the semantics of our access control label could be implemented by the following enforcement policy function:

$$\textsf{policy}\ \textit{access\_simple}\ (\textit{acl}{:}\textsf{lab}, x{:}\textsf{int}\{\textit{acl}\}) = \textsf{if}\ (\textit{member}\ \textit{user}\ \textit{acl})\ \textsf{then}\ \{\circ\}x\ \textsf{else}\ -1$$

This function takes a label like *ACL*(*Alice*, *Bob*) as its first argument, and an integer protected by that ACL as its second argument. If the current user (represented by variable *user*) is a member of x's ACL (according to some function *member*, not shown), then x is returned with its label removed, expressed by the syntax $\{\circ\}x$, so that it can be accessed by the main program. If the membership test fails, it returns -1 and x's value is not released.

FABLE does not, in and of itself, guarantee that a security policy is correctly implemented, but FABLE's design greatly simplifies proof of this fact. In particular, FABLE's type system ensures that labeled data (that is, data with a type $t\{l\}$) is treated *abstractly* by the main program, since terms with a labeled type can only be constructed, examined, or changed within enforcement policy code. Moreover, FABLE's type system ensures that the main program cannot sever or forge the association between a label and the data it protects. In effect, FABLE ensures *complete mediation* of the user's label policy in that no data can be accessed without consulting the correct security policy.

To demonstrate FABLE's flexibility we have used it to encode a range of policies, including access control, static [36] and dynamic information flow [51] with forms of declassification [24], provenance tracking [10] and policies based on security-automata [45]. In our experience, the soundness of FABLE makes proofs of security properties no more difficult—and arguably simpler—than proofs of similar properties in specialized languages [33, 42, 43]. To demonstrate this fact we present proofs of correctness for our access control policies, provenance policies, and static information flow. FABLE opens the possibility of partially automating such proofs, along the lines of user-defined type systems [11], though we leave exploration of this possibility to future work. To our knowledge, no existing framework enables the enforcement of such a wide variety of security policies with an equally high level of assurance.

To evaluate FABLE's practicality we have implemented FABLE as part of the LINKS web-programming language [15]. We call the resulting language SELINKS (for *security-enhanced* LINKS). We have used SELINKS to build two substantial applications: SEWIKI, a 3500-line secure blog/wiki inspired by *Intellipedia* [35] that implements combined access control and provenance policies, and SEWINESTORE, a 1000-line e-commerce application provided with the LINKS distribution that we extended with an access control policy. In general, we have found that FABLE's label-based security policies are neither lacking nor burdensome, and the modular separation of the enforcement policy permitted some reuse of policy code between the two applications.

In the remainder of the paper we present FABLE, our core language for defining and enforcing custom, label-based security policies (Section 2). We show how FABLE can be used to define a range of security policies and that FABLE's design simplifies proofs that these policies are implemented correctly (Section 3). In Section 4 we discuss our SELINKS implementation of FABLE for building web applications and our experience building SEWIKI and SEWINESTORE. Section 5 discusses related work, and Section 6 sketches future work and concludes.

2 FABLE: System F with Labels

This section presents the syntax, static semantics, and operational semantics of FABLE. The next section illustrates FABLE's flexibility by presenting policies that can be expressed along with proofs of attendant security properties.

2.1 Syntax

Figure 1 defines FABLE's syntax. Throughout, we use the notation $\vec{a}$ to stand for a list of elements of the form $a_1, \ldots, a_n$; where the context is clear, we will also treat $\vec{a}$ as the set of elements $\{a_1, \ldots, a_n\}$.

Expressions e extend a standard polymorphic λ-calculus, System F [20]. Standard forms include integer values n, variables x, abstractions $\lambda x{:}t.e$, term application $e_1\, e_2$, the fixpoint combinator $\textsf{fix}\ x{:}t.v$, type abstraction $\Lambda\alpha.e$ and type application $e\,[t]$.

$$
\begin{array}{rrcl}
\text{Expressions} & e & ::= & n \mid x \mid \lambda x{:}t.e \mid e_1\, e_2 \mid \mathsf{fix}\ x{:}t.v \mid \Lambda\,\alpha.e \mid e\,[t] \\
\text{(Fable-specific)} & & \mid & C(\vec{e}) \mid \mathsf{match}\ e\ \mathsf{with}\ p_i \Rightarrow e_i \mid (\!|e|\!) \mid \{\circ\}e \mid \{e\}e' \\
\text{Types} & t & ::= & \mathsf{int} \mid \alpha \mid \forall\,\alpha.t_2 \mid (x{:}t_1) \to t_2 \\
\text{(Fable-specific)} & & \mid & \mathsf{lab} \mid \mathsf{lab} \sim e \mid t\{e\}
\end{array}
\qquad
\begin{array}{rrcl}
\text{Patterns} & p & ::= & x \mid C(\vec{p}) \\
\text{Pre-values} & u & ::= & n \mid C(\vec{u}) \mid \lambda x{:}t.e \mid \Lambda\,\alpha.e \\
\text{App. values} & v_{app} & ::= & u \mid (\!|\{e\}v_{pol}|\!) \\
\text{Policy values} & v_{pol} & ::= & u \mid \{e\}v_{pol}
\end{array}
$$

Figure 1. Syntax of FABLE.

type abbreviation	$\mathsf{typename}\ N\ \alpha = t\ \mathsf{in}\ e_2$	$\equiv$	$(N\,t' \mapsto ((\alpha \mapsto t')t))e_2$
let binding	$\mathsf{let}\ x = e_1\ \mathsf{in}\ e_2$	$\equiv$	$(\lambda x : t.e_2)\ e_1$ for some t
polymorphic function def.	$\mathsf{let}\ f\langle\alpha\rangle(x{:}t) = e_1\ \mathsf{in}\ e_2$	$\equiv$	$\mathsf{let}\ f = \mathsf{fix}\ f{:}t'.\Lambda\,\alpha.\lambda x{:}t.e_1\ \mathsf{in}\ e_2$ for some t'
policy function def.	$\mathsf{policy}\ f\langle\alpha\rangle(x{:}t) = e_1\ \mathsf{in}\ e_2$	$\equiv$	$\mathsf{let}\ f = \mathsf{fix}\ f{:}t'.\Lambda\,\alpha.\lambda x{:}t.(\!|e_1|\!)\ \mathsf{in}\ e_2$ for some t'
dependent tuple type	$x{:}t \times t'$	$\equiv$	$\forall\,\alpha.((x{:}t) \to t' \to \alpha) \to \alpha$
dependent tuple introduction	(e, e')	$\equiv$	$\Lambda\,\alpha.\lambda f{:}((x{:}t) \to t' \to \alpha).f\ e\ e'$ for some t, t'
dependent tuple projection	$\mathsf{let}\ x,y = f\ \mathsf{in}\ e$	$\equiv$	$f\,[t_e](\lambda x{:}t.\lambda y{:}t'.e)$ for some t, t', and t_e

Figure 2. Syntactic shorthands.

The syntactic constructs specific to FABLE are distinguished in Figure 1. The expression $C(\vec{e})$ is a label, where C represents an arbitrary constructor and each $e_i \in \vec{e}$ must itself be a label; e.g., in *ACL*(*Alice*, *Bob*), *ACL* is 2-ary label constructor and *Alice* and *Bob* are 0-ary label constructors. Labels can be examined by pattern matching. For example, the expression $\mathsf{match}\ z\ \mathsf{with}\ ACL(x,y) \Rightarrow x$ would evaluate to *Alice* if z's run-time value were *ACL*(*Alice*, *Bob*).

As explained earlier, FABLE introduces the notion of an *enforcement policy* that is a separate part of the program authorized to manipulate the labels on a type. Following Grossman et al. [22] we use *bracketed expressions* $(\!|e|\!)$ to delimit policy code e from the main program. In practice, one could use code signing as in Java [21] to ensure that untrusted policy code cannot be injected into a program.As illustrated earlier, the expression $\{\circ\}e$ removes a label from e's type, while $\{e'\}e$ adds one; we discuss these operations in detail below. Labeling and unlabeling operations may only occur within policy code.

Standard types t include int, type variables α, and universally-quantified types $\forall\,\alpha.t$. Functions have dependent type $(x{:}t_1) \to t_2$ where x names the argument and may be bound in t_2. We illustrate the usage of these types shortly. Labels can be given either type lab or the *singleton type* $\mathsf{lab} \sim e$, which describes label expressions equivalent to e. For example, the label constructor *High* can be given the type lab and the type $\mathsf{lab} \sim High$. Singleton types are useful for constraining the form of label arguments to enforcement policy functions. For example, we could write a specialized form of our previous *access_simple* function:

$$\mathsf{policy}\ access\_pub\ (acl{:}\mathsf{lab} \sim ACL(World),\ x{:}\mathsf{int}\{acl\}) = \{\circ\}x$$

The FABLE type checker ensures this function is called only with expressions that evaluate to the label *ACL*(*World*)—i.e., the call *access_pub*(*ACL*(*Alice*,*Bob*),*e*) will be rejected. In effect, the type checker is performing access control at compile time according to the constraint embodied in the type. We will show in Section 3.3 that these constraints are powerful enough to encode an information flow policy that can be checked entirely at compile time.

The dependent type $t\{e\}$ describes a term of type t that is associated with a label e. Such an association is made using the syntax $\{e\}e'$. For example, $\{High\}1$ is an expression of type $\mathsf{int}\{High\}$. Conversely, this association can be broken using the syntax $\{\circ\}e$. For example, $\{\circ\}(\{High\}1)$ has type int. Now we can illustrate how dependent function types $(x{:}t_1) \to t_2$ can be used. The function *access_simple* can be given the type $(acl{:}\mathsf{lab}) \to (x{:}\mathsf{int}\{acl\}) \to \mathsf{int}$ which indicates that the first argument *acl* serves as the label for the second argument x. Instead of writing $(x{:}t_1) \to t_2$ when x does not appear in t_2, we simply omit it. Thus *access_simple*'s type could be written $(acl{:}\mathsf{lab}) \to \mathsf{int}\{acl\} \to \mathsf{int}$.

The operational semantics of Section 2.3 must distinguish between application and policy values in order to ensure that policy code does not inadvertently grant undue privilege to application functions. Application values v_{app} consist of either "pre-values" u—integers n, labels containing values, type and term abstractions—or labeled policy values wrapped with $(\!|\cdot|\!)$ brackets. Values within policy code are pre-values preceded by zero or more relabeling operations.

Encodings. To make our examples more readable, we use the syntactic shorthands shown in Figure 2. The first three shorthands are mostly standard. We use the policy keyword to designate policy code instead of using brackets $(\!|\cdot|\!)$. A dependent pair (e, e') of type $x{:}t \times t'$ allows x, the name for the first element, to be bound in t', the type of the second

```
policy login(user:string, pw:string) =
  let token = match checkpw user pw with
    USER(k) ⇒ USER(k)
    _ ⇒ FAILED in
  (token, {token}0)
```

```
policy member(u:lab, a:lab) =
  match a with
    ACL(u, i) ⇒ TRUE
    ACL(j, tl) ⇒ member u tl
    _ ⇒ FALSE
```

```
policy access⟨k,α⟩(u:lab∼USER(k),
                   cap:int{u},
                   acl:lab, data:α{acl}) =
  match member u acl with
    TRUE ⇒ {○}data
    _ ⇒ halt#access denied
```

Figure 3. Enforcing a simple access control policy.

element. For example, the first two arguments to the *access_pub* function above could be packaged into a dependent pair of type $(acl{:}\mathsf{lab}\sim ACL(World) \times \mathsf{int}\{acl\})$ which is inhabited by terms such as $(ACL(World),\{ACL(World)\}1)$. Dependent pairs can be encoded using dependently-typed functions. We extend the shorthand for function application, policy function definitions, type abbreviations, and tuples to multiple type and term arguments in the obvious way. We also write _ as a wildcard ("don't care") pattern variable.

Phantom label variables. We extend the notation for polymorphic functions in a way that permits quantification over the *expressions* that appear in a type. Consider the example below:

$$\mathsf{policy}\ add\langle l\rangle(x{:}\mathsf{int}\{l\}, y{:}\mathsf{int}\{l\}) = \{l\}(\{\circ\}x + \{\circ\}y)$$

This policy function takes two like-labeled integers x and y as arguments; unlabels them and adds them together; and finally relabels the result, having type $\mathsf{int}\{l\}$. The unusual thing about this function is that the label l is not a normal term argument, but is being quantified—any label l would do.

The reason this makes sense is that in FABLE, (un)labeling operations are merely hints to the type checker to (dis)associate a label term and a type. These operations, along with all types, can be erased at runtime without affecting the result of a computation. After erasing types, our example would become $\mathsf{policy}\ add\ (x, y) = x + y$, which is clearly only a function of x and y, with no mention of l. For this reason, we can treat *add* as *polymorphic in the labels* of x and y—it can be called with any pair of integers that have the same label, irrespective of what label that might be. We express this kind of polymorphism by writing the *phantom label variable* l, together with any other normal type variables like $\alpha, \beta, \ldots$, in a list that follows the function name. In the example above, the phantom variable of *add* are listed as $\langle l\rangle$. Of course, not all label arguments are phantom. For instance, in the *access_simple* function of Section 1, the *acl* is a label argument that *is* passed at runtime. For simplicity, we do not formalize phantom variable polymorphism. Our technical report [2] does model phantom variables and contains the associated proof of soundness.

Example: Access control policy. Figure 3 illustrates a simple, but complete, enforcement policy for access control. Protected data is given a label listing those users authorized to access the data. In particular, such data has type $t\{acl\}$ where *acl* encodes the ACL as a label.

The policy's *login* function calls an external function *checkpw* to authenticate a user by checking a password. If authentication succeeds (the first pattern), *checkpw* returns a label $USER(k)$ where k is some unique identifier for the user. The *login* function returns a pair consisting of this label and a integer labeled with it; this pair serves as our runtime representation of a principal. The *access* function takes the two elements of this pair as its first two arguments. Since FABLE enforces that only policies can produce labeled values, we are assured that the term with type $\mathsf{int}\{USER(k)\}$ can only have been produced by *login*. The *access* function's last two arguments consist of the protected data's label, *acl*, and the data itself, *data*. The *access* function calls the *member* function to see whether the user token u is present in the ACL. If successful, the label *TRUE* is returned, in which case *access* returns the data with its *acl* label removed.

2.2 Typing

Figure 4 defines the typing rules for FABLE. The main judgment $\Gamma \vdash_c e : t$ types expressions. The index c indicates whether e is part of the policy or the application. Only policy terms are permitted to use the unlabeling and relabeling operators. Γ records three kinds of information; $x : t$ maps variables to types, α records a bound type variable, and $e \succ p$ records the assumption that e matches pattern p, used when checking the branches of a pattern-match.

The rules (T-INT), (T-VAR), (T-FIX), (T-TAB) and (T-TAP) are standard for polymorphic lambda calculi. (T-ABS) and (T-APP) are standard for a dependently-typed language. (T-ABS) introduces a dependent function type of the form $(x{:}t_1) \to t_2$. (T-APP) types an application of a (dependently-typed) function. As usual, we require the type t_1 of the argument to match the type of the formal parameter to the function. However, since x may occur in the return type

$\Gamma \vdash_c e : t$ — Expression e has type t in environment Γ under color c

$$\begin{array}{rrcl} \text{Environments} & \Gamma & ::= & \cdot \mid x:t \mid \alpha \mid e \succ p \mid \Gamma_1,\Gamma_2 \\ \text{Substitutions} & \sigma & ::= & \cdot \mid (x \mapsto e) \mid (\alpha \mapsto t) \mid \sigma_1,\sigma_2 \\ \text{Colors} & c & ::= & \mathit{pol} \mid \mathit{app} \end{array}$$

$$\Gamma \vdash_c n : \mathsf{int}\ \text{(T-INT)} \qquad \frac{x:t \in \Gamma}{\Gamma \vdash_c x : t}\ \text{(T-VAR)}$$

$$\frac{\Gamma \vdash t \quad \Gamma, f:t \vdash_c v : t}{\Gamma \vdash_c \mathsf{fix}\ f{:}t.v : t}\ \text{(T-FIX)} \qquad \frac{\Gamma, \alpha \vdash_c e : t}{\Gamma \vdash_c \Lambda\alpha.e : \forall\alpha.t}\ \text{(T-TAB)} \qquad \frac{\Gamma \vdash t \quad \Gamma \vdash_c e : \forall\alpha.t'}{\Gamma \vdash_c e\,[t] : (\alpha \mapsto t)t'}\ \text{(T-TAP)}$$

$$\frac{\Gamma \vdash t \quad \Gamma, x:t \vdash_c e : t'}{\Gamma \vdash_c \lambda x{:}t.e : (x{:}t) \to t'}\ \text{(T-ABS)} \qquad \frac{\Gamma \vdash_c e_1 : (x{:}t_1) \to t_2 \quad \Gamma \vdash_c e_2 : t_1}{\Gamma \vdash_c e_1\,e_2 : (x \mapsto e_2)t_2}\ \text{(T-APP)}$$

$$\frac{\Gamma \vdash_c e_i : \mathsf{lab}}{\Gamma \vdash_c C(\vec{e}) : \mathsf{lab} \sim C(\vec{e})}\ \text{(T-LAB)} \qquad \frac{\Gamma \vdash_c e : \mathsf{lab} \sim e'}{\Gamma \vdash_c e : \mathsf{lab}}\ \text{(T-HIDE)} \qquad \frac{\Gamma \vdash_c e : \mathsf{lab}}{\Gamma \vdash_c e : \mathsf{lab} \sim e}\ \text{(T-SHOW)}$$

$$\frac{\begin{array}{c}\Gamma \vdash_c e : \mathsf{lab} \qquad \Gamma \vdash t \qquad p_n = x \quad \text{where } x \notin \mathrm{dom}(\Gamma) \\ \vec{x_i} = FV(p_i) \setminus \mathrm{dom}(\Gamma) \quad \Gamma, \vec{x_i} : \mathsf{lab} \vdash_c p_i : \mathsf{lab} \quad \Gamma, \vec{x_i} : \mathsf{lab}, e \succ p_i \vdash_c e_i : t\end{array}}{\Gamma \vdash_c \mathsf{match}\ e\ \mathsf{with}\ p_1 \Rightarrow e_1 \ldots p_n \Rightarrow e_n : t}\ \text{(T-MATCH)} \qquad \frac{\Gamma \vdash_{pol} e : t\{e'\}}{\Gamma \vdash_{pol} \{\circ\}e : t}\ \text{(T-UNLAB)}$$

$$\frac{\Gamma \vdash_{pol} e : t \quad \Gamma \vdash_{pol} e' : \mathsf{lab}}{\Gamma \vdash_{pol} \{e'\}e : t\{e'\}}\ \text{(T-RELAB)} \qquad \frac{\Gamma \vdash_{pol} e : t}{\Gamma \vdash_c (\!|e|\!) : t}\ \text{(T-POL)} \qquad \frac{\Gamma \vdash_c e : t \quad \Gamma \vdash t \cong t'}{\Gamma \vdash_c e : t'}\ \text{(T-CONV)}$$

$\Gamma \vdash t \cong t'$ — Types t and t' are convertible

$$\begin{array}{rrcl} \text{Type contexts} & T & ::= & \bullet \mid \bullet\{e\} \mid x{:}\bullet \to t \mid x{:}t \to \bullet \mid \forall\alpha.\bullet \\ \text{Term label contexts} & L & ::= & \mathsf{lab} \sim \bullet \mid t\{\bullet\} \end{array}$$

$$\Gamma \vdash t \cong t\ \text{(TE-ID)} \qquad \frac{\Gamma \vdash t \cong t'}{\Gamma \vdash t' \cong t}\ \text{(TE-SYM)} \qquad \frac{\Gamma \vdash t \cong t'}{\Gamma \vdash T \cdot t \cong T \cdot t'}\ \text{(TE-CTX)} \qquad \frac{e \succ p \in \Gamma}{\Gamma \vdash L \cdot e \cong L \cdot p}\ \text{(TE-REFINE)}$$

$$\frac{\forall\sigma.(\mathrm{dom}(\sigma) = FV(e_1) \wedge \Gamma \vdash \sigma(e_1) : \mathsf{lab}) \Rightarrow \sigma(e_1) \overset{c}{\rightsquigarrow} \sigma(e_2)}{\Gamma \vdash L \cdot e_1 \cong L \cdot e_2}\ \text{(TE-REDUCE)}$$

$\Gamma \vdash t$ — Type t is well-formed in environment Γ

$$\Gamma \vdash \mathsf{int}\ \text{(K-INT)} \qquad \frac{\alpha \in \Gamma}{\Gamma \vdash \alpha}\ \text{(K-TVAR)} \qquad \Gamma \vdash \mathsf{lab}\ \text{(K-LAB)} \qquad \frac{\Gamma \vdash_{pol} e : \mathsf{lab}}{\Gamma \vdash \mathsf{lab} \sim e}\ \text{(K-SLAB)}$$

$$\frac{\Gamma \vdash t \quad \Gamma \vdash_{pol} e : \mathsf{lab}}{\Gamma \vdash t\{e\}}\ \text{(K-LABT)} \qquad \frac{\Gamma \vdash t_1 \quad \Gamma, x:t_1 \vdash t_2}{\Gamma \vdash (x{:}t_1) \to t_2}\ \text{(K-FUN)} \qquad \frac{\Gamma, \alpha \vdash t}{\Gamma \vdash \forall\alpha.t}\ \text{(K-ALL)}$$

Figure 4. Static semantics of FABLE.

t_2, the type of the application must substitute the actual argument e_2 for x in t_2. As an example, consider an application of the *access_simple* function, having type $(acl{:}\mathsf{lab}) \to \mathsf{int}\{acl\} \to \mathsf{int}$, to the term $ACL(Alice, Bob)$. According to (T-APP) the resulting expression is a function with type $\mathsf{int}\{ACL(Alice,Bob)\} \to \mathsf{int}$, which indicates that the function can be applied only to an integer labeled with precisely $ACL(Alice,Bob)$. This is the key feature of dependent typing—the type system ensures that associations between labels and the terms they protect cannot be forged or broken.

Rule (T-LAB) gives a label term $C(\vec{e})$ a singleton label type $\mathsf{lab} \sim C(\vec{e})$ as long as each component $e_i \in \vec{e}$ has type lab. According to this rule $ACL(Alice,Bob)$ can be given the type $\mathsf{lab} \sim ACL(Alice,Bob)$. For that matter, the expression $((\lambda x{:}\mathsf{lab}.x)\ High)$ can be given the type $\mathsf{lab} \sim ((\lambda x{:}\mathsf{lab}.x)\ High)$, there is no requirement that e be a value. The rule (T-HIDE) allows a singleton label type like this one to be subsumed to the type of all labels, lab. Rule (T-SHOW) does the converse, allowing the type of a label to be made more precise.

Rule (T-MATCH) checks pattern matching. The first premise confirms that expression e being matched is a label. The second line of premises describes how to check each branch of the match. Our patterns differ from patterns in, say, ML in two respects. First, the second premise on the second line requires $\Gamma, \vec{x_i} : \mathsf{lab} \vdash_c p_i : lab$, indicating that

patterns in FABLE are allowed to contain variables that are defined in the context Γ. Second, pattern variables may occur more than once in a pattern. Both of these features make it convenient to use pattern matching to check for term equality. For example, in the expression $\mathsf{let}\ y = \mathit{Alice}\ \mathsf{in\ match}\ x\ \mathsf{with}\ \mathit{ACL}(y,y) \Rightarrow e$, the branch e is evaluated only if the runtime value for the label variable x is $\mathit{ACL}(\mathit{Alice},\mathit{Alice})$.

A key feature of (T-MATCH) is the final premise on the second line, which states that the body of each branch expression e_i should be checked in a context including the assumption $e \succ p_i$, which states that e matches pattern p_i. This assumption can be used to refine type information during checking (similar to typecase [23]) using the rule (T-CONV), which we illustrate shortly. (T-MATCH) also requires that no branch "leaks" any variables bound by patterns in the final type of the match; this is ensured by the second premise, $\Gamma \vdash t$, which confirms t is well-formed in the top-level environment (i.e., one not including pattern-bound variables). For simplicity we require a default case in pattern matching expressions: the third premise requires the last pattern to be a single variable x that does not occur in Γ. In our SELINKS implementation we check that matching is complete using standard techniques [44].

Rule (T-UNLAB) types an unlabeling operation. Given an expression e with type $t\{e'\}$, the unlabeling of e strips off the label on the type to produce an expression with type t. Conversely, (T-RELAB) adds a label e' to the type of e. The *pol*-index on these rules indicates that both operations are only admissible in policy terms. This index is introduced by (T-POL) when checking the body of a bracketed term $(\!|e|\!)$. For example, given expression $e \equiv \lambda x{:}\mathsf{int}\{\mathit{Public}\}.(\!|\{\circ\}x|\!)$, we have $\cdot \vdash_{\mathit{app}} e : \mathsf{int}\{\mathit{Public}\} \to \mathsf{int}$ since $\{\circ\}x$ will be typed with index *pol* by (T-POL).

Rule (T-CONV) allows e to be given type t' assuming it can given type t where t and t' are *convertible*, written $\Gamma \vdash t \cong t'$. Rules (TE-ID) and (TE-SYM) define convertibility to be reflexive and symmetric. Rule (TE-CTX) structurally extends convertibility using *type contexts* T. The syntax $T \cdot t$ denotes the application of context T to a type t which defines the type that results from replacing the occurrence of the hole $\bullet$ in T with t. For example, if T is the context $\bullet\{C\}$, then $T \cdot \mathsf{int}$ is the type $\mathsf{int}\{C\}$. (Of course, rule (TE-CTX) can be applied several times to relate larger types.)

The most interesting rules are (TE-REFINE) and (TE-REDUCE), which consider types that contain labels (constructed by applying context L to an expression e). Rule (TE-REFINE) allows two structurally similar types to be considered equal if their embedded expressions e and p have been equated by pattern matching, recorded as the constraint $e \succ p$ by (T-MATCH). To see how this would be used, consider the following example:

```
let tok,cap = login "Joe" "xyz" in
  match tok with USER(k) ⇒ access tok cap
                 _ ⇒ halt
```

We give the *login* function the type $\mathsf{string} \to \mathsf{string} \to (l{:}\mathsf{lab} \times \mathsf{int}\{l\})$ The type of *access* (defined in Figure 3) is $(u{:}\mathsf{lab} \sim \mathit{USER}(k)) \to \mathsf{int}\{u\} \to t$. We type check *access tok* using rule (T-APP), which requires that the function's parameter and its formal argument have the same type t. However, here *tok* has type lab while *access* expects type $\mathsf{lab} \sim \mathit{USER}(k)$. Since the call to *access* occurs in the first branch of the match, the context includes the refinement $\mathit{tok} \succ \mathit{USER}(k)$ due to (T-MATCH). From (T-SHOW) we can give *tok* type $\mathsf{lab} \sim \mathit{tok}$, and by applying (TE-REFINE) we have $\mathsf{lab} \sim \mathit{tok} \cong \mathsf{lab} \sim \mathit{USER}(k)$ and so *tok* can be given type $\mathsf{lab} \sim \mathit{USER}(k)$ as required. Similarly, for *access tok cap*, we can check that the type $\mathsf{int}\{\mathit{tok}\}$ of *cap* is convertible with $\mathsf{int}\{\mathit{USER}(k)\}$ in the presence of the same assumption.

Rule (TE-REDUCE) allows FABLE types to be considered convertible if the expression component of one is reducible to the expression component of the other [3]; reduction $e \overset{c}{\rightsquigarrow} e'$ is defined shortly in Figure 4. For example, we have $\cdot \vdash \mathit{int}\{(\lambda x{:}\mathsf{lab}.x)\ \mathit{Low}\} \cong \mathit{int}\{\mathit{Low}\}$ since $(\lambda x{:}\mathsf{lab}.x)\ \mathit{Low} \overset{c}{\rightsquigarrow} \mathit{Low}$. One complication is that type-level expressions may contain free variables. For example, suppose we wanted to show $y : \mathsf{lab} \vdash \mathit{int}\{(\lambda x{:}\mathsf{lab}.x)\ y\} \cong \mathit{int}\{y\}$. It seems intuitive that these types should be considered convertible, but we do not have that $(\lambda x{:}\mathsf{lab}.x)\ y \overset{c}{\rightsquigarrow} y$ because y is not a value. To handle this case, the rule permits two types to be convertible if, for any well-typed substitution σ of the free variables of e_1, $\sigma(e_1) \overset{c}{\rightsquigarrow} \sigma(e_2)$. This captures the idea that the precise value of y is immaterial—all reductions on well-typed substitutions of y would reduce to the value that was substituted for y.

Satisfying this obligation by exhaustively considering all possible substitutions is obviously intractable. Additionally, we have no guarantee that an expression appearing in a type will converge to a value. Thus, type-checking in FABLE, as presented here, is undecidable. This is not uncommon in a dependent type system; e.g., type-checking in Cayenne is undecidable [4]. However, other dependently typed systems impose restrictions on the usage of recursion in type-level expressions to ensure that type-level terms always terminate [6]. We explore one such option in the full system presented in Appendix C. Additionally, there are several possible decision procedures that can be used to partially-decide type convertability. One simplification would be to attempt to show convertability for closed types

$e \overset{c}{\rightsquigarrow} e'$ Small-step chromatic reduction rules

Evaluation contexts $E_c ::= \bullet e \mid v_c \bullet \mid \bullet[t] \mid C(\vec{v_c}, \bullet, \vec{e}) \mid \mathsf{match} \bullet \mathsf{with}\ p_i \Rightarrow e_i \mid \{e\}\bullet \mid \{\circ\}\bullet$

$$\frac{e \overset{c}{\rightsquigarrow} e'}{E_c \cdot e \overset{c}{\rightsquigarrow} E_c \cdot e'}\ \text{(E-CTX)} \qquad \frac{e \overset{pol}{\rightsquigarrow} e'}{(\!|e|\!) \overset{app}{\rightsquigarrow} (\!|e'|\!)}\ \text{(E-POL)}$$

$$(\lambda x{:}t.e)\ v_c \overset{c}{\rightsquigarrow} (x \mapsto v_c)e\ \text{(E-APP)} \quad (\Lambda\alpha.e)\ [t] \overset{c}{\rightsquigarrow} (\alpha \mapsto t)e\ \text{(E-TAP)} \quad \mathsf{fix}\ f{:}t.v \overset{c}{\rightsquigarrow} (f \mapsto \mathsf{fix}\ f{:}t.v)v\ \text{(E-FIX)}$$

$$\frac{\forall i < j.\ v_c \not\succ p_i : \sigma_i \qquad v_c \succ p_j : \sigma_j}{\mathsf{match}\ v_c\ \mathsf{with}\ p_1 \Rightarrow e_1 \ldots p_n \Rightarrow e_n \overset{c}{\rightsquigarrow} \sigma_j(e_j)}\ \text{(E-MATCH)} \quad (\!|C(\vec{u})|\!) \overset{app}{\rightsquigarrow} C(\vec{u})\ \text{(E-BLAB)} \quad (\!|n|\!) \overset{app}{\rightsquigarrow} n\ \text{(E-BINT)}$$

$$(\!|\lambda x{:}t.e|\!) \overset{app}{\rightsquigarrow} \lambda x{:}t.(\!|e|\!)\ \text{(E-BABS)} \quad (\!|\Lambda\alpha.e|\!) \overset{app}{\rightsquigarrow} \Lambda\alpha.(\!|e|\!)\ \text{(E-BTAB)} \quad (\!|e|\!) \overset{pol}{\rightsquigarrow} e\ \text{(E-NEST)} \quad \{\circ\}\{e\}v_{pol} \overset{pol}{\rightsquigarrow} v_{pol}\ \text{(E-UNLAB)}$$

$e \succ p : \sigma$ Expression e matches pattern p under substitution σ

$$p \succ p : \cdot\ \text{(U-PATID)} \qquad v \succ x : x \mapsto v\ \text{(U-VAR)} \qquad \frac{\forall i.\sigma_i^* = (\sigma_0, \ldots, \sigma_{i-1}) \qquad e_i \succ \sigma_i^* p_i : \sigma_i}{C(\vec{e}) \succ C(\vec{p}) : \vec{\sigma}}\ \text{(U-CON)}$$

Figure 5. Dynamic semantics of FABLE.

only—i.e. no free variables. In our implementation of FABLE, SELINKS, we use a combination of three techniques. First, we use type information. If l is free in a type, and the declared type of l is $\mathsf{lab} \sim e$, then we can use this information to substitute e for l. Similarly, if the type context includes an assumption of the form $l \succ e$ (when checking the branch of a pattern), we can substitute l with e. Finally, since type-level expressions typically manipulate labels by pattern matching, we use a simple heuristic to determine which branch to take when pattern matching expressions with free variables. These techniques suffice for all the examples in this paper and both our SEWIKI and SEWINESTORE applications. Our technical report [2] discusses these decision procedures in greater detail and proves them sound.

Finally, the judgment $\Gamma \vdash t$ states that t is well-formed in Γ. Rules (K-INT), (K-TVAR), and (K-LAB) are standard, (K-FUN) defines the standard scoping rules for names in dependent function types, and (K-ALL) defines the standard scoping rule for universally-quantified type variables. (K-SLAB) and (K-LABT) ensure that all expressions e that appear in types can be given lab-type. Notice that type-level expressions are typed in *pol*-context. Because FABLE enjoys a type-erasure property any (un)labeling operations appearing in types pose no security risk. We use this feature to good effect in Section 3.2 to protect sensitive information that may appear in labels.

2.3 Operational semantics

Figure 5 defines FABLE's operational semantics. We define a pair of small-step reduction relations $e \overset{app}{\rightsquigarrow} e'$ and $e \overset{pol}{\rightsquigarrow} e'$ for application and policy expressions, respectively. Rules of the form $e \overset{c}{\rightsquigarrow} e'$ are *polychromatic*—they apply both to policy and application expressions. Since the values for each kind of expression are different, we also parameterize the evaluation contexts E_c by the color of the expression; i.e., the context, either *app* or *pol*, in which the expression is to be reduced. Rule (E-CTX) uses these evaluation contexts E_c, similar to the type contexts used above, to enforce a left-to-right evaluation order for a call-by-value semantics. Policy expression reduction $e \overset{pol}{\rightsquigarrow} e'$ takes place within brackets according to (E-POL). The rules (E-APP), (E-TAP), and (E-FIX) define function application, type application, and fixed-point expansion, respectively, in terms of substitutions; these are all standard. Rule (E-MATCH) relies on a standard pattern matching judgment $v \succ p : \sigma$, also defined in Figure 5, which is true when the label value matches the pattern such that $v = \sigma(p)$. (E-MATCH) determines the first pattern p_j that matches the expression v and reduces the match expression to the matched branch's body after applying the substitution. The (U-CON) rule in the pattern matching judgment $v \succ p : \sigma$ is the only non-trivial rule. As explained in Section 2.2, since pattern variables may occur more than once in a pattern, (U-CON) must propagate the result of matching earlier sub-expressions when matching subsequent sub-expressions. For example, pattern matching should fail when attempting to match $ACL(Alice, Bob)$ with $ACL(x,x)$. This is achieved in (U-CON) because, after matching $(Alice \succ x : x \mapsto Alice)$ using (U-VAR), we must try to match Bob with $(x \mapsto Alice)x$, which is impossible.

An applied policy function will eventually reduce to a bracketed policy value v_{pol}. When v_{pol} has the form $(\!|u|\!)$, the brackets may be removed so that the value u can be used by application code. (E-BLAB) and (E-BINT) handle label

expressions $(\!|C(\vec{u})|\!)$ and integers n, respectively. To maintain the invariant that (un)labeling operators only appear in policy code, rules (E-BABS) and (E-TABS) extrude only the λ and Λ binders, respectively, from bracketed abstractions, allowing them to be reduced according to (E-APP) or (E-TAP). Brackets cannot be removed from labeled values $(\!|\{e\}u|\!)$ by application code, to preserve the labeling invariant. On the other hand, brackets can be removed from any expression by policy code, according to (E-NEST). This is useful when reducing expressions such as $(\!|\lambda x{:}t.x|\!)(\!|v|\!)$, which produces $(\!|(\!|v|\!)|\!)$ after two steps; (E-NEST) (in combination with (E-POL)) can then remove the inner brackets. Finally, (E-UNLAB) allows an unlabeling operation to annihilate the top-most relabeling operation. Notice that the expressions within a relabeling operation are never evaluated at runtime—relabelings only affect the types and are purely compile time entities. The types that appear elsewhere, such as (E-TAP), are also erasable, as is usual for System F.

2.4 Soundness

We state the standard type soundness theorems for FABLE here. In addition to ensuring that well-typed programs never go wrong or get stuck, we have put this soundness result to good use in proving that security policies encoded in FABLE satisfy desirable security properties. We discuss this further in the next section. Appendix C contains proofs of the following theorems for an extension of FABLE that includes references and substructural types (Section 3.4).

Theorem (Progress)**.** *If* $\cdot \vdash_c e : t$*; then either* $\exists e'.e \overset{c}{\rightsquigarrow} e'$ *or* $\exists v_c.e = v_c$*.*

Theorem (Preservation)**.** *If* $\cdot \vdash_c e : t$ *and* $e \overset{c}{\rightsquigarrow} e'$*; then,* $\cdot \vdash_c e' : t$*.*

3 Example Policies in FABLE

This section demonstrates FABLE's flexibility by using it to encode several security policies. We prove that any well-typed program using one of these policies enjoys relevant security properties—i.e., the program is sure to enforce the policy correctly. Space constraints preclude presentation of all of the encodings we have explored, so we focus on three kinds of policies: access control, provenance, and static information flow. We also sketch extensions to FABLE that support substructural types useful for enforcing state-based policies, like security automata. We conclude by discussing how FABLE's design eases the construction of proofs of policy correctness.

3.1 Access Control Policies

Access control policies govern how programs release information but, once the information is released, do not control how it is used. To prove that an access control policy is implemented correctly, we must show that programs not authorized to access some information cannot learn the information in any way, e.g., by bypassing a policy check (something not uncommon in production systems [38]) or by employing some kind of covert channel. We call this security condition *non-observability*.

Intuitively, we can state non-observability as follows. If some program P is not allowed to access a resource v_1 having a label l, then a program P' that is identical to P except that v_1 has been replaced with some other resource v_2 (having the same type and label as v_1) should evaluate in the same way as P—it should produce the same result and take the same steps along the way toward producing that result. If this were not true then, assuming P's reduction is deterministic, P must be inferring information about the protected resource.

To make this intuition formal, we will show that the evaluations of programs P and P' are *bisimilar*, where the only difference between them is the value of the protected resource. To express this, first we define an equivalence relation called *similarity up to* l (analogous to definitions of low-equivalence [36, 10]) which holds for two terms e and e' if they only differ in sub-terms that are labeled with l, with the intention that l is the label of restricted resources.

Definition 1 (Similarity up to l)**.** *Expressions* e *and* e'*, identified up to* α*-renaming, are similar up to label* l *according to the following relation:*

$$e \sim_l e \qquad \{l\}e \sim_l \{l\}e' \qquad \frac{e \sim_l e' \quad l' \neq l}{\{l'\}e \sim_l \{l'\}e'} \qquad \frac{e \sim_l e'}{\lambda x{:}t.e \sim_l \lambda x{:}t.e'} \qquad \frac{e_1 \sim_l e_1' \quad e_2 \sim_l e_2'}{e_1\ e_2 \sim_l e_1'\ e_2'} \qquad \ldots$$

The second rule is the most important. It states that arbitrary expressions e and e' are considered similar at label l when both are labeled with l. Other parts of the program must be structurally identical, as stated by the remaining congruence rules (not all are shown; the full relation can be found in Appendix D). We extend similarity to a bisimulation as follows: two similar terms are bisimilar if they always reduce to similar subterms, and do so indefinitely or until no further reduction is possible (thus this notion of bisimulation is both *timing and termination sensitive*).

```
typename Prov α= (l:lab{Auditors} × α{{∘}l})
policy apply⟨α,β⟩ (lf:Prov (α → β), mx:Prov α) =
  let l,f = lf in
  let m,x = mx in
  let y = ({∘}f) ({∘}x) in
  let lm = Union({∘}l, {∘}m) in
    ({Auditors}lm, {lm}y)
```

```
policy flatten⟨α⟩ (x:Prov (Prov α)) =
    let l,inner = x in
    let m,a = inner in
    let lm = Union({∘}l, {∘}m) in
      ({Auditors}lm, {lm}a)
```

Figure 6. Enforcing a dynamic provenance tracking policy.

Definition 2 (Bisimulation)**.** *Expressions e_1 and e_2 are* bisimilar at label l*, written $e_1 \approx_l e_2$, if and only if $e_1 \sim_l e_2$ and there exists e_1', e_2' such that $e_1 \overset{c}{\rightsquigarrow} e_1' \Leftrightarrow e_2 \overset{c}{\rightsquigarrow} e_2'$ and $e_1' \approx_l e_2'$.*

The non-observability theorem states that we have some program e that contains no policy-bracketed terms (it is just application code) but, according to the substitution σ may refer to our access control functions *access* and *member* via the free variables a and m and use a capability ($\{\{user\}0\}$) via the free variable *cap*. This capability gives the program the authority of user *user*. The program may also refer to some protected resource x whose label is *acl*, but the authority of *user* is insufficient to access x according to the access control policy because $(member\ user\ acl \overset{c*}{\rightsquigarrow} False)$. Under these conditions, we can show that for any two (well-typed) v_i we substitute for x according to substitution σ_i, the resulting programs are bisimilar—their reduction is independent of the choice of v_i.

Theorem (Non-observability)**.** *Given a ($\{\cdot\}$)-free expression e such that $(a : t_a, m : t_m, cap : \mathsf{int}\{user\}, x : t\{acl\} \vdash_{app} e : t_e)$ where acl and user are label constants, and given a substitution $\sigma = (a \mapsto access, m \mapsto member, cap \mapsto (\{\{user\}0\}))$. Then, for type-respecting substitutions $\sigma_i = \sigma, x \mapsto v_i$ where $\cdot \vdash_{app} v_i : t\{acl\}$ for i=1,2, we have $(member\ user\ acl \overset{c*}{\rightsquigarrow} False) \Rightarrow \sigma_1(e) \approx_{acl} \sigma_2(e)$.*

Notice that this theorem is indifferent to the actual implementation of the *acl* label and the *member* function. Thus, while our example policy is fairly simplistic, a far more sophisticated model could be used. For instance, we could have chosen labels to stand for RBAC- or RT-style roles [34, 26] and *member* could invoke a decision procedure for determining role membership. Likewise, the theorem is not concerned with the origin of the *user* capability—a function more sophisticated than *login* (e.g., that relied on cryptography) could have been used. The important point is that FABLE ensures the second component of the user credential (l:lab$\sim USER(k) \times$ int$\{l\}$) is unforgeable by application code. Finally, it would be straightforward to extend our theorem to speak to policies that provide access to more than one resource with a single membership test, as in the following code

```
policy access_cap⟨k⟩(u:lab∼USER(k), cred:int{u}, acl:lab) =
  match member u acl with True ⇒ Λα.λx:α{acl}.{∘}x
                         _ ⇒ #fail
```

Here the caller presents a user credential and an access control label *acl* (but no resource labeled with that label). If the membership check succeeds, a function with type $\forall\alpha.\alpha\{acl\} \to \alpha$ is returned. This function can be used to immediately unlabel any resource with the authorized label. This is useful when policy queries are expensive. It is also useful for encoding a form of delegation; rather than releasing his user credential, a user could release a function that uses that credential to a limited effect. Of course, this may be undesirable if the policy is known to change frequently, but even this could be accommodated. Variations that combine static and dynamic checks are also possible.

3.2 Dynamic Provenance Tracking

Provenance is "information recording the source, derivation, or history of some information" [10]. Provenance is relevant to computer security for at least two reasons. First, provenance is useful for auditing, e.g., to discover whether some data was inappropriately released or modified. Second, provenance can be used to establish data integrity, e.g., by carefully accounting for a document's sources. This section describes a label-based provenance tracking policy we constructed in FABLE. To prove that this policy is implemented correctly we show that all programs that use it will accurately capture the dependences (in the sense of information flow) on a value produced by a computation.

Figure 6 presents the provenance policy. We define the type *Prov* α to describe a pair in which the first component is a label l that records the provenance of the second component. The policy is agnostic to the actual form of l. Provenance labels could represent things like authorship, ownership, the channel on which information was received,

```
policy lub(x:lab, y:lab) = match x,y with _, HIGH | HIGH, _ ⇒ HIGH | _, _ ⇒ LOW
policy join⟨α,l,m⟩ (x:α{l}{m}) = ({lub l m}x)
policy sub⟨α,l⟩ (x:α{l}, m:lab) = ({lub l m}x)
policy apply⟨α,β,l,m⟩ (f:(α → β){l}, x:α) = {l}(({∘}f) x)
```

Figure 7. Enforcing a static information flow policy.

etc. An interesting aspect of *Prov* α is that the provenance label is itself labeled with the 0-ary label constant *Auditors*. This represents the fact that provenance information is subject to security concerns like confidentiality and integrity. Intuitively, one can think of data labeled with the *Auditors* label as only accessible to members of a group called *Auditors* (of course, a more complex policy could be used). Finally, note that because the provenance label l is itself labeled (having type lab{*Auditors*}) it would be incorrect to write $\alpha\{l\}$ as the second component of the type since this requires that l have type lab. Therefore we unlabel l when it appears in the type of the second component. As explained in Section 2.2, unlabeling operations in types pose no security risk since the types are erased at runtime.

The policy function *apply* is a wrapper for tracking dependences through function applications. In an idealized language like FABLE it is sufficient to limit our attention to function application, but a policy for a full language would define wrappers for other constructs as well. The first argument of *apply* is a provenance-labeled function *lf* to be called on the second argument *mx*. The body of *apply* first decomposes the pair *lf* into its label l and the function *f* itself and does likewise for the argument *mx*. Then it applies the function, stripping the label from both it and its argument first. The provenance of the result is a combination of the provenance of the function and its argument. We write this as the label pair $Union(l,m)$ which is then associated with the final result. Notice that we strip the *Auditors* labels from l and m before combining them, and then add the label to the label of the final result.

The policy also defines a function *flatten* to convert a value of type *Prov* (*Prov* α) to one of type *Prov* α by extracting the nested labels (the first two lines) and then collapsing them into a *Union* (third line) that is associated with the inner pair's labeled component (fourth line).

An example client program that uses this provenance policy is the following:

$$\text{let } client\langle\alpha,\beta,\gamma\rangle\ (f : Prov(\alpha \to \beta \to \gamma), x : Prov\ \alpha, y : Prov\ \beta) = apply\ [\beta][\gamma]\ (apply\ [\alpha][\beta \to \gamma]\ f\ x)\ y$$

This function takes a labeled two-argument function *f* as its argument and the two arguments *x* and *y*. It calls *apply* twice to get a result of type *Prov* γ. This will be a tuple in which the first component is a labeled provenance label of the form $Union(Union(lf,lx), ly)$ and the second component is a value labeled with that provenance label. In the label, we will have that *lf* is the provenance label of the function argument *f* and *lx* and *ly* are the provenance of the arguments *x* and *y*, respectively. Note that a caller of *client* can instantiate the type variable γ to be a type like *Prov* int. In this case, the type of the returned value will be *Prov* (*Prov* int), which can be flattened if necessary.

We can prove that provenance information is tracked correctly following Cheney et al. [10]. The intention is that if a value *x* of type *Prov* α influences the computation of some other value *y*, then *y* must have type *Prov* β (for some β) and its provenance label must mention the provenance label of *x*. If provenance is tracked correctly, a change to *x* will only affect values like *y*; other values in the program will be unchanged. We can express this using a similarity relation $v_1 \sim_l v_2$ like the one defined in Section 3.1 which relates two values if they differ only on sub-terms of type *Prov* α whose provenance label mentions l. Thus, an application program e that is compiled with the policy of Figure 6 and is executed in contexts that differ only in the choice of a tracked value of label l will compute results that differ only in sub-terms that are also colored using l.

Theorem (Dependency correctness)**.** *Given a* $(\![\cdot]\!)$*-free expression e such that* $a : t_a, f : t_f, x : Prov\ t \vdash_{app} e : t'$*, and given a substitution* $\sigma = (a \mapsto apply, f \mapsto flatten)$*. Then, for type-respecting substitutions* $\sigma_i = \sigma, x \mapsto v_i$ *where* $\vdash_{app} v_i : Prov\ t$ *for i=1,2 it is the case that* $v_1 \sim_l v_2$ *implies* $(\sigma_1(e) \overset{app_*}{\rightsquigarrow} v_1' \ \wedge \ \sigma_2(e) \overset{app_*}{\rightsquigarrow} v_2') \Rightarrow v_1' \sim_l v_2'$

3.3 Static Information Flow

Both policies discussed so far rely on runtime checks. This section illustrates how FABLE can be used to encode *static* lattice-based information flow policies that require no runtime checks. In a static information flow type system (as found in FlowCaml [36]) labels l have no run-time witness; they only appear in types $t\{l\}$. Labels are ordered by a relation $\sqsubseteq$ that typically forms a lattice [17]. This ordering is lifted to a subtyping relation on labeled types such that

$l_1 \sqsubseteq l_2 \Rightarrow t\{l_1\} <: t\{l_2\}$. Assuming the lattice ordering is fixed during execution, well-typed programs can be proven to adhere to the policy defined by the initial label assignment appearing in the types.

Figure 7 illustrates the policy functions, along with a small sample program. In our encoding we define a two-point security lattice with atomic labels *HIGH* and *LOW* and protected expressions will have labeled types like $t\{HIGH\}$. The ordering $LOW \sqsubset HIGH$ is exemplified by the *lub* (least-upper-bound) operation for the lattice. The *join* function (similar to the *flatten* function from Figure 6) combines multiple labels on a type into a single label. The interesting thing here is the label attached to x is a label expression *lub l m*, rather than an label value like *HIGH*. The type rule (T-CONV) presented in Figure 4 can be used to show that a term with type $\mathsf{int}\{lub\ HIGH\ LOW\}$ can be given type $\mathsf{int}\{HIGH\}$ (since $lub\ HIGH\ LOW \overset{c}{\rightsquigarrow} HIGH$). This is critical to being able to type programs that use this policy.

The policy includes a subsumption function *sub*, which takes as arguments a term x with type $\alpha\{l\}$ and a label m and allows x to be used at the type $\alpha\{lub\ l\ m\}$. This is a restatement of the subsumption rule above, as $l \sqsubseteq m$ implies $l \sqcup m = m$. (Once types are erased, *sub* is essentially the identity function and could be optimized away.) Finally, the policy function *apply* unlabels the function f in order to call it, and then adds f's label on the computed result.

Consider the following client program as an example usage of the static information flow policy.

```
let client (f:(int{HIGH} →int{HIGH}){LOW}, x:int{LOW}) =
  let x = (sub [int] x HIGH) in
    join [int] (apply [int{HIGH}][int{HIGH}] f x)
```

The function *client* here calls function f with x, where f expects a parameter of type $\mathsf{int}\{HIGH\}$ while x has type $\mathsf{int}\{LOW\}$. For the call to type check, the program uses *sub* to coerce x's type to $\mathsf{int}\{lub\ LOW\ HIGH\}$ which is convertible to $\mathsf{int}\{HIGH\}$. The call to *apply* returns a value of type $\mathsf{int}\{HIGH\}\{LOW\}$. The call to *join* collapses the pair of labels so that *client*'s return type is $\mathsf{int}\{lub\ HIGH\ LOW\}$, which converts to $\mathsf{int}\{HIGH\}$.

We have proved that FABLE programs using this policy enjoy the standard noninterference property (Appendix F). We have also shown that a FABLE static information flow policy is at least as permissive as the information flow policy implemented by the functional subset of Core-ML, the formal language of FlowCaml [33] (Appendix G). Finally, Appendix A shows how the dynamic provenance tracking and static information flow policies can be combined to enforce dynamic information flow.

3.4 Tracking side-effects with FABLE

Thus far, we have presented security policies that apply only to purely functional programs. Enforcing policies that take side-effects into account is challenging because side-effects can be conduits of information. Consider the small program $x := 0$; if $p == 0$ then $x := 1$. In this program, the assignment to x occurs only when p is 0. Thus the contents of x contain information about p, e.g., if at the end x is zero then p is nonzero. The usual strategy in information flow type systems to prevent such leaks is to add a label l to the type of mutable pointers, as in $\alpha\ \mathsf{ref}\{l\}$. Extending our prior information flow policy we might try to write a wrapper for assignments as follows:

```
policy update⟨α,l⟩ (x:α ref{l}, y:α) = let _ = ({∘}x) := y in {Effect(l)}0
let client⟨l⟩(x:int ref{l}) = let _ = update [int] x 0 in 0
```

Here, the *update* function returns a value of type $\mathsf{int}\{Effect(l)\}$ that witnesses the side-effect—the label $Effect(l)$ reveals that a side-effect involving reference l occurred during the computation. By requiring application functions that have side-effects to include effect witnesses in their returned values, we can design a policy that limits information leaks through effects. However, in the FABLE system we have presented, nothing prevents an application program from ignoring the returned effect. Consider the *client* application program. This function updates the location x, obtains an effect witness from the *update* policy, and then discards it. We can give *client* the type x:int ref$\{l\} \rightarrow$int. Even though calling this function clearly produces a side-effect (a write to a location labeled l), the type of this function does not reflect this fact; i.e., we would like the type of this function to be something like x:int ref$\{l\} \rightarrow \mathsf{int}\{Effect(l)\}$. This indicates correctly that calling f has an effect l that may have influenced the returned value.

The full FABLE calculus addresses this problem by embedding the language in Figure 1 in a *substructural* type system [46]. In such a type system, the return type of *update* can be given a *relevant type*, which enforces that the returned value is "used" at least once. In our setting, a use occurs when the value is passed to policy code. Thus the type system would fail to type check *client* because it discards a relevant value. Space constraints preclude a discussion of the full language, which appears in Appendix C along with proofs of Progress and Preservation theorems.

With the addition of relevant types, we show, in Appendix B, how policies encoded in full FABLE can be used to track memory effects. Appendix B.4 shows how *affine* types (another kind of substructural type—values with an affine type may be used *at most once* in the program) can be used to enforce arbitrary safety properties encoded as *security automata*, in a manner similar to Walker [45].

3.5 Proofs of security properties in FABLE

As mentioned in the introduction, FABLE does not, in and of itself, guarantee that well-typed programs implement a particular security policy's semantics correctly. That said, FABLE has been designed to facilitate proof of such theorems. To illustrate how, we chose to use three very different techniques for each of the correctness results reported here. We conclude from our experience that the metatheory of FABLE provides a useful repository of lemmas that can naturally be applied in showing the correctness of various policy encodings. As such, we believe the task of constructing a correctness proof for a FABLE policy to be no more onerous, and possibly considerably simpler, than the corresponding task for a special-purpose calculus that "bakes in" the enforcement of a single security policy. In the remainder of this section, we report on our experience with each of the three proofs and discuss preliminary progress towards reasoning about proofs involving multiple policies.

In all our proofs, two key features of FABLE play a central role. First, dependent-typing in FABLE allows a policy analyst to assume that all policy checks are performed correctly. For instance, when calling the *access* function to access a value v of type $t\{acl\}$, the label expressing v's security policy must be *acl*, and no other. The type system ensures that the application program cannot construct a label, say *ACL*(*Public*), and trick the policy into believing that this label, and not *acl*, protects v; i.e., dependent-typing rules out *confused deputies* [8]. Second, the restriction that application code cannot directly inspect labeled resources ensures that a policy function must mediate every access of a protected resource. Assuring complete mediation is not unique to FABLE— Zhang et al. [50] used CQual to check that SELinux operations on sensitive objects are always preceded by policy checks and Fraser [19] did the same for Minix. However, the analysis in both these instances only ensures that *some* policy check has taken place, not necessarily the correct one. As such, these other techniques are vulnerable to flaws due to confused deputies.

When combined with these two insights, our proof of non-observability for the access control policy (Appendix D) is particularly simple. In essence, the FABLE system ensures that a value with labeled type must be treated abstractly by the application program. With this observation, the proof proceeds in a manner very similar to a proof of value abstraction [22]. This is a general semantic property for languages like FABLE that support parametric polymorphism or abstract types. Indeed, the policy as presented in Figure 3 could have been implemented in a language like ML, which also has these features. For instance, an integer labeled with an access control list could be represented in ML as a user list integer pair with type (string list $\times$ int). A policy module could export this pair as an abstract type, preventing application code from ever inspecting the value directly, and provide a function to expose the concrete type only after a successful policy check. While such an encoding would suffice for the simple policy of Figure 3, it would not work for other idioms like the function *access_cap* of Section 3.1, which reveals some of the structure of a label to avoid the need for additional checks. Abstract types on their own are also insufficient to support static checking of policies, as in the case of information flow or security automata.

To show dependency correctness (Appendix E) we followed a proof technique used by Tse and Zdancewic [42]. This technique involves defining a logical relation [29] that relates terms whose set of provenance labels include the same label l. Recall that our goal in this theorem is to show that given $x : Prov\ t \vdash_c e : t$ that $\sigma_1(e)$ is related to $\sigma_2(e)$, where σ_i substitutes a provenance labeled value v_i for x in e. The crux of this proof involves showing that the logical relation is preserved under substitution—i.e., a form of substitution lemma for the logical relation. While constructing the infrastructure to define the logical relation requires some work, strategic applications of standard substitution lemma for FABLE can be used to discharge the proof without much difficulty.

While it would be possible to reuse our infrastructure for the dependency correctness proof to show the noninterference result for the static information flow policy (as in Tse and Zdancewic), we choose instead to use another technique, due to Pottier and Simonet [33] (Appendix F). This technique involves representing a pair of executions of a FABLE program within the syntax of a single program and showing a subject reduction property holds true. As with the logical relations proof, once we had constructed the infrastructure to use this technique, the proof was an easy consequence of FABLE's preservation theorem.

All our correctness theorems impose the condition that an application program be "()-free". That is, these theorems apply only to situations where a single policy is in effect within a program. However, in practice, multiple policies may

be used in conjunction and we would like to reason that interactions between the policies do not result in violations of the intended security properties. To characterize the conditions under which a policy can definitely be composed with another, we defined a simple type-based criterion, which when satisfied by two (or more) policies π_P and π_Q, implies that neither policy will interfere with the functioning of the other policy when applied in tandem to the same program.

Intuitively, a policy can be made composable by enclosing all its labels within a unique top-level label constructor that can be treated as a *namespace*. A policy that only manipulates labels and labeled terms that belong to its own namespace can be safely composed with another policy. The main benefit of compositionality is modularity; when multiple composable policies are applied to a program, one can reason about the security of the entire system by considering each policy in isolation. Policy designers that are able to encapsulate their policies within a namespace can package their policies as libraries to be reused along with other policy libraries.

Our notion of composition is a noninterference-like property—a policy is deemed composable if it can be shown not to depend on, or influence the functioning of another policy. As with noninterference properties in other contexts, this condition is often too restrictive for many realistic examples in which policies, by design, must interact with each other. We find that policies that do not compose according to this definition perform a kind of declassification (or endorsement) by allowing labeled terms to exit (or unlabeled terms to enter) the policy's namespace. We conjecture that the vast body of research into declassification [37] can be brought to bear here in order to recover a degree of modularity for interacting policies. Appendix H contains the formal statement and proof of the policy noninterference theorem and further discussion of the applicability of this condition.

4 SELINKS: FABLE applied to web programming

We have implemented FABLE as an extension to the LINKS functional web-programming language [15]; we call our extension *Security-Enhanced* LINKS, or SELINKS. This section briefly describes our SELINKS implementation and presents our experience using it to build two applications, a wiki SEWIKI and an on-line store SEWINESTORE.

4.1 SELINKS

LINKS is a new programming language designed to make web programming easier. It allows a programmer to write an entire multi-tier web application as a single LINKS program and the compiler will split that program into components to run on the database (as SQL), server (as a local fragment of LINKS code), and client (as Javascript). By extending LINKS with FABLE's label-based security policies, we can build applications that police data within and across tiers, up to the level of trust we have in those tiers. In our test applications we assume the server and database are trusted but the client is not.

LINKS is a functional programming language equipped with standard features such as recursive variant types ("datatypes"), pattern matching, parametric polymorphism, and higher-order functions. As such, the FABLE policies we have presented so far transliterate naturally into SELINKS. One difference is that rather than define a special type lab as in FABLE, in SELINKS, we allow values of certain recursive variant types to be treated as labels, which can then be pattern matched using standard LINKS syntax. Legal variants can consist only of nullary constructors or constructors whose types refer only to other legal variants. Policy functions are designated by the qualifier policy, as in our examples, and SELINKS has native support for dependent products.

4.2 SEWIKI and SEWINESTORE

SEWIKI consists of about 3500 lines of SELINKS code. It was constructed to mimic Intellipedia, a set of web-based document management systems designed to promote information sharing between the sixteen agencies that comprise the United States intelligence community [35]. A SEWIKI document is represented as a tree according to the following type definition:

typename *Doc* = *Node* of (*Doc* × *Doc*) | *Leaf* of *String* | *Labeled* of (*l*:*DocLabel* × *Doc*{*l*})

The *Labeled* constructor allows nodes to have a security label according to the dependent pair (*l*:*DocLabel* × *Doc*{*l*}). Here, the type *DocLabel* is an alias for *AclLabel*, used to define security labels for documents. It defines an access control policy:

typename *Group* = *Authors* | *Auditors* | *Administrators*
typename *AclLabel* = *Acl* of (*read*:*List*(*Group*) × *write*:*List*(*Group*))

The type *AclLabel* includes a single constructor *Acl* applied to a tuple in which we maintain a list of groups authorized to read a document and a list for those allowed to modify a document. There are three groups: *Authors*, in which all document authors are members; *Auditors*, the group of users that are trusted to audit a document; and *Administrators*, which include only the system administrators. We implement authentication credentials as terms of the type *Credential* (not shown). This type is similar to the type of credentials produced by *login* in the FABLE access control policy (Figure 3) except that *Credential* includes additional useful information such as the user's identifier and name.

Possible document modifications are mediated by the *write_access* policy function, which has the following type:

policy $write\_access : \forall \alpha,\beta.\ Credential \rightarrow (f{:}\alpha \rightarrow \beta) \rightarrow (l{:}AclLabel) \rightarrow \alpha\{l\} \rightarrow \beta\{l\}$

This function allows a caller to pass in a user's credential and function *f* that is intended to modify a resource α labeled with an access control label *l*. If *write_access* determines that the user is in the writer's group of the *AclLabel l*, the function *f* is applied and the policy relabels the modified document with the (access control) label of the original.

We also added a revision history tracking policy to SEWIKI, similar in spirit to the provenance tracking policy of Section 3.2. We track provenance through all operations that alter a document while still enforcing the access control policy. To do this, we redefine *DocLabel* to be a *CompLabel* type that also contains provenance labels:

typename $ProvOp = Create \mid Modify \mid Copy \mid Delete \mid Restore \mid Relabel$
typename $ProvAction = (oper{:}ProvOp \times user{:}User)$
typename $CompLabel = Composite\ \text{of}\ (acl{:}AclLabel \times prov{:}List(ProvAction))$

The *prov* component of the *Composite* constructor records the provenance information as a list of *ProvAction*s. A *ProvAction* records an operation that was performed together with the identity of the authorizing user. Tracked operations are of type *ProvOp*. Operations include document creation, modification, copy-pasting from other documents, deletion and restoration (documents are never completely deleted in SEWIKI), and document relabeling. For the last, authorized users are presented with an interface to alter the access control labels that protect a document. Policy functions that enforce this composite policy are fairly modular—there is one set for access control and one for provenance. Each projects out the appropriate component from the composite label without considering the remaining labels.

In addition to building SEWIKI, we extended the "wine store" e-commerce application that comes with LINKS by creating labels to represent users and associating such labels with orders in a shopping cart and order history. This helps ensure that an order is only accessed by the customer who created it. As in the SEWIKI, the user id is stored as a *Credential*; order information is defined below. The policy functions to view and add items to an order are implemented as simple wrappers around the same read and write access policies used in the SEWIKI.

typename $Order = (l{:}AclLabel \times List(CartItem)\{l\})$

Our experience using SELINKS to write these applications has been quite positive. The access control policies were easy to define and to use, with policy code consisting of roughly 200 lines of code total (including helper functions). The access control and login policy code was modular enough to be shared in its entirety by the two applications. The provenance policy consists of about 100 lines of code, and was also straightforward to use. Unlike the provenance policy from Section 3.2, SEWIKI provenance labels essentially track only direct data flows to and from other documents. This makes them much easier to program with since far fewer program operations need to be mediated by the policy. To support richer policies while easing the programming burden we are investigating an approach based on program rewriting that, given a policy specification, automatically transforms a program to insert the appropriate label manipulations.

5 Related Work

Dependently-typed languages have found use in a wide variety of applications [48, 47, 4, 6]. In the context of security, Zheng and Myers [51] formalize support for *dynamic security labels* that can be associated with data to express information flow policies. The technical machinery for associating labels to terms in their system is similar to ours. There are two main differences. First, the security policy—an information flow policy with a particular label model—is expressed directly in the type system whereas in FABLE both the security policy and the label model are customizable. As discussed in Section 3.3, dynamic labels for information flow policies can be encoded in FABLE as a combination of our dynamic provenance and static information flow policies. Second, FABLE allows non-values to appear in types, e.g., *lub l m* in Figure 7. This permits a combination of static and dynamic policy checking, but

at the cost of potentially-undecidable type checking. Our SELINKS implementation uses heuristics to ensure that type-checking never diverges.

Walker's "type system for expressive security policies" [45] is also dependently-typed. Labels in Walker's language are uninterpreted predicates rather than arbitrary expressions. Walker's system can enforce policies expressed as security automata, which can capture any *safety property*. This kind of policy is also enforceable in FABLE when extended with substructural types. However, in Walker's system, the policy is always enforced by means of a runtime check. In order to recover some amount of static checking, Walker suggests that a user might add additional rules to the type system, though he is not specific about how this would be done. These additional rules would have to be proved correct with respect to a desired security property.

It has been observed that dependent types can be used to express a kind of customized type system [47], and FABLE's policy functions fit this description. For example, the *sub* function in the policy of Figure 7 effectively introduces a subsumption rule into the type system. Researchers have explored how user-defined type systems can be supported directly via customizable *type qualifiers*. Shankar et al. [39] have used lattice-based type qualifiers in CQual [16] to track dataflow properties like taintedness [39], and Zhang et al. [50] and Fraser et al. [19] have used qualifiers to check complete mediation in access control systems. Millstein et al [11, 1] have developed an approach in which programmers can indicate data invariants that custom type qualifiers are intended to signify. In some cases, they are able to automatically verify that these invariants are correctly enforced by the custom type rules. While their invariants are relatively simple, we ultimately would like to develop a framework in a similar vein, in which correctness properties for FABLE's enforcement policies can be at least partially automated. Marino et al. [28] have proposed using proof assistants for this purpose, and we plan to explore this idea in the context of FABLE policies.

Li and Zdancewic show how to encode information flow policies in Haskell [49]. They define a meta-language that makes the control-flow structure of a program available for inspection within the program itself. Their enforcement mechanism relies on the lazy evaluation strategy of Haskell that allows the control flow graph to be inspected for information leaks prior to evaluation. While their encoding permits the use of custom label models, they only show an encoding of an information flow policy. It is not clear their system could be used to encode the range of policies discussed here.

Our technique of separating the enforcement policy from the rest of the program is based on Grossman et al's *colored brackets* [22]. They use these brackets to model type abstraction, whereas we use them to ensure that the privilege of unlabeling and relabeling terms is not mistakenly granted to application code. As a result, we do not need to specially designate application code that may arise within policy terms, keeping things a bit simpler. We plan to investigate the use of different colored brackets to distinguish different enforcement policies, following Grossman et al.'s support for multiple agents.

Finally, inasmuch as we have targeted the LINKS web-programming language [15] as the platform on which to build FABLE, our work is related to Swift [12] and SIF [14], two Jif-based projects that aim to secure web applications. The former is a technique that permits a web application to be split according to a policy into Javascript code that runs on the client and Java code on the server, while the latter is a framework in which to build secure servlets. As discussed in Section 4, LINKS provides similar functionality, except it additionally integrates database access code into the framework. With our new security checking features in SELINKS, as in Swift, practical, verified, end-to-end security for multi-tiered applications is within reach.

6 Conclusions

This paper has presented FABLE, a core formalism for a programming language in which programmers may specify security policies and reason that these policies are properly enforced. We have shown that FABLE is flexible enough to implement a wide variety of security policies, including access control, provenance, and static information flow, among other policies. We have also argued that FABLE's design simplifies proofs that programs using these policies do so correctly. We have implemented FABLE as part of the LINKS web programming language, and we have used the resulting language, which we call SELINKS, to build two substantial applications, a secure wiki and a secure online store. While more work remains to make SELINKS a fully satisfactory platform, to our knowledge, no existing framework enables the enforcement of such a wide variety of security policies with an equally high level of assurance.

References

[1] C. Andreae, J. Noble, S. Markstrum, and T. Millstein. A framework for implementing pluggable type systems. In *OOPSLA '06: Proceedings of the 21st annual ACM SIGPLAN conference on Object-oriented programming systems, languages, and applications*, pages 57–74, New York, NY, USA, 2006. ACM Press.

[2] Anonymous. Fable: A language for enforcing user-defined security policies (technical report). Technical report, July 2007.

[3] D. Aspinall and M. Hoffmann. *Advanced Topics in Types and Programming Languages*, chapter Dependent Types. MIT Press, 2004.

[4] L. Augustsson. Cayenne–a language with dependent types. In *ICFP '98: Proceedings of the third ACM SIGPLAN international conference on Functional programming*, pages 239–250, New York, NY, USA, 1998. ACM Press.

[5] L. Badger, D. F. Sterne, D. L. Sherman, and K. M. Walker. A domain and type enforcement UNIX prototype. *Computing Systems*, 9(1):47–83, 1996.

[6] Y. Bertot and P. Castéran. *Interactive Theorem Proving and Program Development. Coq'Art: The Calculus of Inductive Constructions*. Texts in Theoretical Computer Science. Springer Verlag, 2004.

[7] K. J. Biba. Integrity considerations for secure computer systems. Technical Report ESD-TR-76-372, USAF Electronic Systems Division, 1977.

[8] M. Bishop. *Computer Security: Art and Science*. Addison Wesley, 2003.

[9] P. Buneman, A. Chapman, and J. Cheney. Provenance management in curated databases. In *SIGMOD*, 2006.

[10] J. Cheney, A. Ahmed, and U. Acar. Provenance as dependency analysis. *Database Programming Languages*, pages 138–152, 2007.

[11] B. Chin, S. Markstrum, and T. Millstein. Semantic type qualifiers. In *PLDI '05: Proceedings of the 2005 ACM SIGPLAN conference on Programming language design and implementation*, pages 85–95, New York, NY, USA, 2005. ACM Press.

[12] S. Chong, J. Liu, A. C. Myers, X. Qi, K. Vikram, L. Zheng, and X. Zheng. Secure web application via automatic partitioning. In *SOSP '07: Proceedings of twenty-first ACM SIGOPS symposium on Operating systems principles*, pages 31–44, New York, NY, USA, 2007. ACM.

[13] S. Chong, A. C. Myers, N. Nystrom, L. Zheng, and S. Zdancewic. Jif: Java + information flow. Software release. Located at `http://www.cs.cornell.edu/jif`, July 2006.

[14] S. Chong, K. Vikram, and A. C. Myers. Sif: Enforcing confidentiality and integrity in web applications. In *USENIX Security '07: Proceedings of sixteenth USENIX Security Symposium*.

[15] E. Cooper, S. Lindley, P. Wadler, and J. Yallop. Links: Web programming without tiers. `http://groups.inf.ed.ac.uk/links/papers/links-esop06.pdf`, 2006.

[16] CQual: A tool for adding type qualifiers to C. `http://www.cs.umd.edu/~jfoster/cqual/`.

[17] D. E. Denning. A lattice model of secure information flow. *Communications of the ACM*, 19(5):236–243, May 1976.

[18] C. Fournet and A. D. Gordon. Stack inspection: theory and variants. In *Proceedings of the 29th ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, pages 307–318, 2002.

[19] T. Fraser, J. Nick L. Petroni, and W. A. Arbaugh. Applying flow-sensitive CQUAL to verify MINIX authorization check placement. In *PLAS*. ACM Press, 2006.

[20] J.-Y. Girard. *Interprétation fonctionelle et élimination des coupures de l'arithmétique d'ordre supérieur*. PhD thesis, Université Paris VI I, 1972.

[21] L. Gong. *Inside Java 2 platform security architecture, API design, and implementation*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 1999.

[22] D. Grossman, G. Morrisett, and S. Zdancewic. Syntactic type abstraction. *ACM Trans. Program. Lang. Syst.*, 22(6):1037–1080, 2000.

[23] R. Harper and G. Morrisett. Compiling polymorphism using intensional type analysis. In *POPL '95: Proceedings of the 22nd ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, pages 130–141, New York, NY, USA, 1995. ACM Press.

[24] B. Hicks, D. King, P. McDaniel, and M. Hicks. Trusted declassification:: high-level policy for a security-typed language. In *PLAS '06: Proceedings of the 2006 workshop on Programming languages and analysis for security*, pages 65–74, New York, NY, USA, 2006. ACM Press.

[25] C. E. Landwehr. The best available technologies for computer security. *IEEE Computer*, 16(7):89–100, July 1983.

[26] N. Li, J. C. Mitchell, and W. H. Winsborough. Design of a Role-Based Trust-Management Framework. In *IEEE Symposium on Security and Privacy*, 2002.

[27] P. Loscocco and S. Smalley. Integrating flexible support for security policies into the linux operating system. In *Proceedings of the FREENIX Track: 2001 USENIX Annual Technical Conference*, pages 29–42, Berkeley, CA, USA, 2001. USENIX Association.

[28] D. Marino, B. Chin, T. Millstein, G. Tan, R. J. Simmons, and D. Walker. Mechanized metatheory for user-defined type extensions. In *ACM SIGPLAN Workshop on Mechanizing Metatheory*, 2006.

[29] J. C. Mitchell. *Foundations of Programming Languages*. MIT Press, 1996.

[30] 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.

[31] P. G. Neumann, R. S. Boyer, R. J. Feiertag, K. N. Levitt, and L. Robinson. A provably secure operating system: The system, its applications, and proofs. Technical Report CSL-116, SRI International, Computer Science Laboratory, May 1980.

[32] Perl 5.8.8 documentation - perlsec. `http://perldoc.perl.org/perlsec.html`.

[33] F. Pottier and V. Simonet. Information flow inference for ML. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, 25(1), Jan. 2003.

[34] Role based access control. `http://csrc.nist.gov/rbac/`, 2006.

[35] Reuters, October 2006. U.S. Intelligence Unveils Spy Version of Wikipedia.

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

[37] A. Sabelfeld and D. Sands. Dimensions and principles of declassification. In *IEEE Computer Security Foundations Workshop (CSFW)*, June 2005.

[38] SecurityFocus: Access control bypass vulnerabilities. `http://search.securityfocus.com/swsearch?sbm=%2F&metaname=alldoc&query=access+control+bypass&x=0&y=0`.

[39] U. Shankar, K. Talwar, J. S. Foster, and D. Wagner. Detecting Format String Vulnerabilities with Type Qualifiers. In *Proceedings of the 10th Usenix Security Symposium*, Washington, D.C., Aug. 2001.

[40] Z. Su and G. Wassermann. The essence of command injection attacks in web applications. In *POPL '06: Conference Record of the 33rd ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pages 372–382, New York, NY, USA, 2006. ACM Press.

[41] N. Swamy, M. Hicks, S. Tse, and S. Zdancewic. Managing policy updates in security-typed languages. In *Proc. Computer Security Foundations Workshop (CSFW)*, pages 202–216, July 2006.

[42] S. Tse and S. Zdancewic. Run-time Principals in Information-flow Type Systems. In *IEEE Symposium on Security and Privacy*, 2004.

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

[44] P. Wadler. *The Implementation of Functional Programming Languages*, chapter Efficient Compilation of Pattern Matching. Prentice Hall, 1987.

[45] D. Walker. A type system for expressive security policies. In *POPL '00: Proceedings of the 27th ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, pages 254–267, New York, NY, USA, 2000. ACM Press.

[46] D. Walker. *Advanced Topics in Types and Programming Languages*, chapter Substructural Type Systems. MIT Press, 2004.

[47] H. Xi. Applied Type System (extended abstract). In *post-workshop Proceedings of TYPES 2003*, pages 394–408. Springer-Verlag LNCS 3085, 2004.

[48] H. Xi and F. Pfenning. Dependent types in practical programming. In *Conference Record of POPL 99: The 26th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, San Antonio, Texas*, pages 214–227, New York, NY, 1999.

[49] P. L. S. Zdancewic. Encoding information flow in Haskell. In *CSFW '06: Proceedings of the 19th IEEE workshop on Computer Security Foundations*, page 16, Washington, DC, USA, 2006. IEEE Computer Society.

[50] X. Zhang, A. Edwards, and T. Jaeger. Using CQUAL for static analysis of authorization hook placement. In *USENIX Security*, 2002.

[51] L. Zheng and A. C. Myers. Dynamic security labels and noninterference. In *Proceedings of the 2nd International Workshop on Formal Aspects in Security and Trust (FAST)*, Toulouse, France, August 2004.

```
policy flow(src:lab, dest:lab) =
        let f = if oracle src dest then
                    FLOW(src,dest)
                else NOFLOW in
        (f, {f}())

policy low⟨α⟩ (x:α) = let l = oracle_low() in (l, {l}x)

policy sub⟨α,src,dest⟩(cap:unit{FLOW(src,dest)}, x:α{src}) = {dest}x

policy app⟨α,β,l,m⟩(f:(α →β){l}, x:α{m}) = {JOIN(l, m)} ({∘}f) ({∘}x)
```

```
let client⟨α⟩(lb:lab, b:bool{lb}, lx:lab, x:α{lx}, ly:lab, y:α{ly}) =
    let lxy = JOIN(lx,ly) in
    let fx,capx = flow lx lxy in
    let fy,capy= flow ly lxy in
      match fx,fy with
        FLOW(lx,lxy), FLOW(ly,lxy) →
          let x' = sub [α] capx x in
          let y' = sub [α] capy y in
          let tmp = app [α] [α →α] (b[α]) x' in
            app [α] [α] tmp y'
        _ , _ →... #flow must be allowed if oracle is a lattice
```

Figure 8. Enforcing a dynamic information flow policy.

```
policy flow(src:lab, dest:lab) =
    let f = if oracle_flow src dest then FLOW(src,dest)
            else if oracle_enc src dest then ENC(src,dest)
            else NOFLOW in
    (f, {f}())
```

```
policy aes_enc⟨src,des⟩ (cap:unit{ENC(src,dest)},
                         data:bytes{src},
                         key:bytes{dest}) =
    {dest}( ... key ... data)
```

Figure 9. A trusted declassifier, explicit in the policy.

A Dynamic Information Flow and Downgrading

Realistic information flow policies are rarely as simple as that of Section 3.3. For example, the security label of some data may not be known until run-time, and the label itself may be more complex than a simple atom—e.g., it might be drawn from the DLM [30] or some other higher-level policy language, such as RT [41]. Figure 8 shows how dynamic security labels can be associated with the data and an information flow policy enforced using a combination of static and dynamic checks [51].

The label lattice defined by the external *oracle* function and the enforcement policy interfaces with this policy through the function *flow*, which expects two labels *src* and *dest* as arguments and determines whether the oracle permits information to flow from *src* to *dest*. The representation of these labels is abstract in the policy and depends on the implementation of the *oracle*.

The *flow* function has the following type:

$(src{:}\mathsf{lab}) \to (dest{:}\mathsf{lab}) \to (l{:}\mathsf{lab} \times \mathsf{unit}\{l\})$.

If the *oracle* permits the flow, the *flow* function returns a capability similar to that provided by the *login* function of Figure 3. The *sub* function takes this capability as its first argument as proof that type $\alpha\{src\}$ may be coerced to type $\alpha\{dest\}$. The *low* function must appeal to the oracle to acquire the bottom label in the lattice.

The main program in this example has the same high-level behavior as in the static case—it branches on a boolean and returns either *x* or *y*—but here the security labels of the arguments are not statically known. Instead, the argument *lb* is a label term that specifies the security level of *b*, and similarly *lx* for *x* and *ly* for *y*. As previously, our encoding of booleans requires each branch to have the same type, including the security label. In this case, the program arranges the branches to have the type *JOIN*(*lx*,*ly*). The first three lines of the main expression use the flow function to attempt to obtain capabilities that witness the flow from *lx* and *ly* to *JOIN*(*lx*,*ly*). The match inspects the labels that are returned by *flow* and in case where they are actually *FLOW*(...) the final premise of (T-MATCH) permits the type of *fx* to be refined from $\mathsf{lab} \sim fx$ to $\mathsf{lab} \sim FLOW(lx,lxy)$ and the type of *capx* to be refined to $\mathsf{unit}\{FLOW(lx,lxy)\}$, and similarly for *capy*. The remainder of the program is similar to the static case, but requires more uses of subsumption since less is known statically about the labels. The type inferred for this program is:

$\forall\alpha.\ (lb{:}\mathsf{lab}) \to \beta\,\mathsf{bool}\{lb\} \to (lx{:}\mathsf{lab}) \to \alpha\{lx\} \to (ly{:}\mathsf{lab}) \to \beta\{ly\} \to \alpha\{JOIN(JOIN(lx,ly), lb)\}$

A.1 Downgrading policies

Pure information flow policies are too strong for most programs. Realistic policies must include a provision for downgrading information from higher to lower security levels. Figure 9 shows how a simple, but practical, downgrading policy in the form of a trusted declassifier [24] can be encoded in FABLECORE. Here we enhance the *flow* function of the previous example so that it may produce more than one kind of capability. Now, *oracle_flow* defines a label lattice and if $src \sqsubseteq dest$ then a *FLOW*(*src*,*dest*) capability is returned, as previously. However, the *oracle_enc* function decides whether or not an *encrypted* flow of data labeled *src* is permissible to a destination labeled *dest*; if so, *flow* returns an *ENC*(*src*,*dest*)

capability.

In this case, the policy defines a particular encryption function, *aes_enc* that requires an *ENC(src,dest)* capability as an argument, and encrypts and downgrades *data* at level *src* to the level *dest*. The type of the function is precise enough to state that an encrypted downgrading from *src* to *dest* is only permissible using a *key* that is known to *dest*. This ensures, first, that the downgraded data can be decrypted by *dest*, and perhaps more importantly, ensures that *dest* never receives data encrypted using a key that is secret to *dest*, thereby ensuring that information about a secret key is not released.

Including the *implementation* of the *aes_enc* function in the *policy* is also beneficial, as it clearly establishes a trusted component of the application.

B Embedding FABLECORE in a Substructural Type System (Summary)

Throughout the appendices we use FABLECORE to refer the the functional core of FABLE presented in Section 2 and FABLE to stand for the full system presented in summary here, and in toto in Appendix C.

Thus far, we have presented security policies that apply to purely functional programs. However, a much wider variety of security properties can be enforced when we add substructural typing, [46] to the framework. For instance, while references and side-effects are straightforward to add to FABLECORE, enforcing policies that take side-effects into account is challenging without additional support from the type system.

In this section, we informally discuss why FABLECORE is inadequate for enforcing policies that track side effects. Then we sketch an extension of the type system of Figure 4 to add *relevant* types (values with a relevant type must be used *at least once* in the program), and show an example information flow policy that correctly tracks effects. Other kinds of substructural types are also useful. Appendix B.4 shows how *affine* types (values with an affine type may be used *at most once* in the program) can be used to enforce arbitrary safety properties encoded as *security automata*, in a manner similar to Walker [45].

B.1 Attempting to track effects using FABLECORE

In a language with references, information flow policies must account for leaks that occur due to side-effects. For instance, if an assignment to a variable x occurs in a context that is control-dependent on a secret value b, then the contents of x contain information about the secret. The usual strategy to prevent such leaks is to label locations with a secrecy level and to prevent assignments to all locations with a security level that is lower than the label of b. Using FABLECOREwe might try to write a policy function that permits assignment through a labeled reference. It might look something like the following:

$$\mathsf{policy}\ \mathit{update}\langle\alpha,l\rangle\ (x{:}\alpha\ \mathsf{ref}\{l\}, y{:}\alpha) = \langle\mathsf{unit}\{C(\mathit{LOW},\mathit{EFFECT}(l))\}\rangle(\langle\alpha\ \mathsf{ref}\rangle x) := y$$

This function returns a labeled unit value as a witness of a side-effect—the label indicates that a side-effect occurred in the computation that produced $()$ as a result. However, nothing in FABLECORE prevents the program from ignoring the captured effect, so the following application program will type check: $\lambda x{:}\mathsf{int}\ \mathsf{ref}\{l\}.\ \mathsf{let}\ \_ = \mathit{update}\ [\mathsf{int}]\ x\ 0\ \mathsf{in}\ 0$

This function has the type $l.x{:}\mathsf{int}\ \mathsf{ref}\{l\} \to \mathsf{int}$. Even though calling this function clearly produces a side-effect (a write to a location labeled l), the return type of this function does not reflect this fact. We can use the fact that relevant types must be used at least once to reject the above program as incorrect, since it it discards the effect witness without using it.

B.2 Adding Relevant Types to FABLECORE.

FABLE augments FABLECORE by adding references and supporting *relevant* types. The syntactic extensions are summarized below, as an extension of FABLECORE syntax.

$$\begin{array}{rrcl} \text{Expressions} & e & ::= & \ldots \mid x_\ell \mid\ !e \mid e_1 := e_2 \mid \{\mathsf{rel}\}e \\ \text{Pre-values} & u & ::= & \ldots \mid x_\ell \end{array}$$

$$\begin{array}{rrcl} \text{Effects} & \varepsilon & ::= & 0 \mid 1 \\ \text{Types} & t & ::= & \ldots \mid (x{:}t_1) \xrightarrow{\varepsilon} t_2 \mid t\ \mathsf{ref} \mid \mathsf{rel}t \\ \text{Kinds} & \kappa & ::= & \ldots \mid \mathrm{R} \end{array}$$

Expressions now include memory locations x_ℓ, dereferencing and assignment, while pre-values are augmented to include locations. In addition, we provide policy code with the ability to designate certain values as being *relevant* by extending the form of the relabeling operator to optionally include a rel qualifier. Since expressions can now have side-effects, we must be careful to ensure that expressions that appear in types are pure. The static semantics of FABLE is now a very coarse grained type and effect system that assigns the effect 1 to any expression that may access memory. The type language is extended to include effect annotations that appear above function arrows. We also include t ref,

$\Gamma;A \vdash_c e : t;\varepsilon$ — Expression e has type t and effect ε in environment Γ with relevant assumptions A

List of relevant assumptions $A ::= \cdot \mid x \mid A_1, A_2$

$$\frac{x : t \in \Gamma \quad A \subseteq \{x\}}{\Gamma;A \vdash_c x : t;0}\ \text{(T-VAR)} \qquad \frac{x : t \in \Gamma}{\Gamma;A \vdash_{pol} x : t;0}\ \text{(T-POLVAR)} \qquad \frac{\Gamma;A \vdash_{pol} e : t;\varepsilon}{\Gamma;A \vdash_{pol} \{\mathsf{rel}\}e : \mathsf{rel}\ t;\varepsilon}\ \text{(T-RLAB)}$$

$$\frac{t \neq \mathsf{rel}\ t' \quad \Gamma;A \vdash_c e : t;\varepsilon}{\Gamma;A \vdash_c e : \mathsf{rel}\ t;\varepsilon}\ \text{(T-SUBR)} \qquad \frac{\Gamma;A \vdash t_1 :: \kappa \quad t' = (x{:}t_1) \xrightarrow{\varepsilon} t_2 \quad (\kappa = \mathrm{R}) \Rightarrow (A' = A, x)\mathbf{else}\ (A' = A) \quad \Gamma, x : t_1; A' \vdash_c e : t_2;\varepsilon \quad (A = \emptyset) \Rightarrow (t = t')\mathbf{else}\ (t = \mathsf{rel}\ t')}{\Gamma;A \vdash_c \lambda x{:}t_1.e : t;0}\ \text{(T-ABS)}$$

$$\frac{\Gamma;A \vdash_c e : t\ \mathsf{ref};\varepsilon}{\Gamma;A \vdash_c !e : t;1}\ \text{(T-DRF)} \qquad \frac{\Gamma;A \vdash t :: \mathrm{M} \quad \Gamma;A_1 \vdash_c e_1 : t\ \mathsf{ref};\varepsilon \quad \Gamma;A_2 \vdash_c e_2 : t;\varepsilon}{\Gamma;A_1 \oplus A_2 \vdash_c e_1 := e_2 : \mathsf{unit};1}\ \text{(T-ASN)} \qquad \frac{A \subseteq FV(e) \quad \Gamma;A \vdash_{pol} e : t;\varepsilon}{\Gamma;A \vdash_c (\!|e|\!) : t;\varepsilon}\ \text{(T-POL)}$$

$\Gamma;A \vdash t :: \kappa$ — Type t has kind κ in environment $\Gamma;A$

$$\frac{\Gamma;A \vdash t :: \kappa}{\Gamma;A \vdash \mathsf{rel}\ t :: \mathrm{R}}\ \text{(K-1)} \qquad \frac{\Gamma;A \vdash t :: \mathrm{M}}{\Gamma;A \vdash t :: \mathrm{R}}\ \text{(K-2)} \qquad \frac{\Gamma;A \vdash_{pol} e : \mathsf{lab};0}{\Gamma;A \vdash \mathsf{lab} \sim e :: \mathrm{U}}\ \text{(K-3)} \qquad \frac{\Gamma;A \vdash t :: \kappa \quad \Gamma;A \vdash_{pol} e : \mathsf{lab};0}{\Gamma;A \vdash t\{e\} :: \mathrm{M} \sqcup \kappa}\ \text{(K-4)}$$

Figure 10. Static semantics of FABLE. (Selected judgments)

the type of references to t; and most importantly, we include a type $\mathsf{rel}t$, which qualifies the type t as being *relevant*. Finally, we extend the language of type-kinds to include the R-kind, which is the classification given to relevant types.

Figure 10 shows the main elements of the FABLE language; the full semantics can be found in Appendix C. In addition to the properties of FABLECORE, the FABLE type system enforces two main invariants. First, it tracks effects on expressions to ensure that all type-level expressions are pure. Second, it ensures that any value given a relevant type is always passed, at least once, to a policy function, where it can be *consumed* in a manner chosen by the policy function. With the machinery of relevant types in hand, the policy can ensure that a term that is generated as a witness to a side-effect (with a type like $\mathsf{rel}\ \mathsf{unit}\{C(LOW, EFFECT(l)\}$) cannot be hidden away by the program—if such a value is produced in a function, it must be part of the returned value (say a component of a tuple); or, it must be passed to the policy which can decide how to propagate the effect information.

The main typing judgment in Figure 10 is of the form $\Gamma;A \vdash_c e : t;\varepsilon$. Here, Γ has the same form as in FABLECORE. The context A component records the set of relevantly typed variables that are used in e. Finally, the effect ε records whether or not e has an effect.

Given a list of assumptions A, we use $A = A_1 \oplus A_2$ to denote splitting A into two disjoint lists A_1 and A_2. Given a judgment $\Gamma;A \vdash_{app} e : t;\varepsilon$, the system must ensure that on all possible control paths through e, every assumption in A is used *at least once*. However, we also want to exempt a policy expression e from such restrictions since the policy is trusted to manipulate relevant values, like effect witnesses, in a manner of its choosing. The first two type rules, (T-VAR) and (T-POLVAR), enforce this requirement. In (T-VAR), since a variable x uses at most a single relevant assumption, we require A to be no larger than $\{x\}$. This restriction does not apply when typing policy code; a relevant assumption can be treated intuitionistically within policy code. (T-RLAB) provides the policy with a way of introducing relevantly typed values and is similar to (T-RELAB) from FABLECORE.

According to (T-SUBR), any type can be treated as a relevant type. When typing an abstraction, we must ensure that any relevant assumptions used in the body of the abstraction are reflected in the type of the function. Similarly, effects in the body should also be reflected in the type. In the second premise of (T-ABS), if the argument of the function is a given a relevant type ($\kappa = \mathrm{R}$), then we must record the function argument as a relevant assumption. In the third premise, if the body has an effect ε, then we must record this effect on the function arrow. Finally, in the conclusion, if relevant assumptions in A are used in the body of the function e, then, since we must ensure that every relevant value is used at least once during runtime, we must give the function itself a relevant type ($\mathsf{rel}(x : t_1 \xrightarrow{\varepsilon} t_2)$) in the conclusion. This ensures that the function will be applied at least once.

(T-DRF) records that $!e$ has a side-effect by recording an $\varepsilon = 1$ in the conclusion. The rule (T-ASN) first records that an assignments has a side-effect ($\varepsilon = 1$). Then, as is standard in a substructural system, we are permitted to split the

```
policy update⟨α,l⟩ (x:(α ref){l}, y:α) = {rel}{C(LOW, EFFECT(l))} ({∘}x) := y
policy combine (l:lab, m:lab) = match l m with C(l1,fx1),C(l2,fx2) =⟩ C(lub l1 l2, glb fx1 fx2)
policy seq⟨α::U, β::U, l,m⟩(e1:(unit→rel α{l}), e2:(unit→rel β{m})) = let x = e1 () in {rel}{combine l m}(e2 ())
```

Figure 11. Tracking effects using relevant types in FABLE.

set of relevant assumptions A into A_1 and A_2 and ensure that e_i discharges all assumption in A_i. Finally, for simplicity, we do not allow relevant values to escape into the heap ($\Gamma; A \vdash t :: \mathrm{M}$). In a system system that attempts to track, say, resource usage, it might be useful to lift this restriction. However, for our purposes, the additional complication does not provide any obvious benefit. We leave investigation of this issue to future work.

Finally, (T-POL) is as before; we can type a policy term in a *pol*-colored context. However, we have the additional restriction that every relevant assumption A should occur somewhere in the body of e. This ensures that an application function cannot mistakenly hide a relevant value from the policy. Consider, as an example, the following application program as an example: $\lambda effect{:}\ \mathsf{rel}\ \mathsf{unit}\{l\}.(\!|e|\!)0$ Here, we must ensure that the effect witness argument *effect* is destructed by the policy. By requiring that *effect* appear free in the policy expression e, we ensure when this function is applied to some witness that the witness is passed to the policy expression e. Without this restriction, it would be possible for the witness to be "forgotten" when this function is applied.

Finally, Figure 10 shows some of the kinding judgments. (K-1) gives R-kind to a relevant type; (K-2) encodes the sub-kinding relation; (K-3) and (K-4) ensure that all type-level expressions are effect free; (K-4) additionally ensures that kind of $t\{e\}$ is no less that M-kind, given the kind hierarchy $\mathrm{U} \sqsubset \mathrm{M} \sqsubset \mathrm{R}$.

Appendix C contains proofs of the standard progress and preservation theorems for FABLE (Theorems 5 and Theorem 13) stated below. In addition, we have proved that, for programs which terminate, every relevantly-typed value in an application program is eventually passed to policy code where it can be destructed (Theorem 15). Finally, it is also straightforward to prove a type erasure property as none of the reduction rules ever inspect the types.

Theorem (Progress)**.** *Given* $\Gamma;\cdot$ *well-formed;* $\Gamma;\cdot \vdash_c e : t;\varepsilon$*; and memory* M *such that* $\Gamma \models M$*. Then either* $\exists e'.(M,e) \overset{c}{\rightsquigarrow} (M',e')$ *or* $\exists v_c.e = v_c$*.*

Theorem (Preservation)**.** *Given* $\Gamma;\cdot$ *well-formed;* $\Gamma;\cdot \vdash_c e : t;\varepsilon$*; and memory* M *such that* $\Gamma \models M$*, and* e *such that* $\Gamma;\cdot \vdash_c e : t;\varepsilon$*. Then* $(M,e) \overset{c}{\rightsquigarrow} (M',e') \;\Rightarrow\; \Gamma;\cdot \vdash_c e' : t;\varepsilon' \;\wedge\; \Gamma \models M' \;\wedge\; \varepsilon' \leq \varepsilon$

B.3 Tracking effects using FABLE

Equipped with relevant types, we can write a policy that accurately tracks effects through a program. Our strategy is to label values with a composite label $C(l1, l2)$, where $l1$ describes the standard (confidentiality) label of the value, and $l2$ represents the *effect* of the computation that produced the value as a result. The policy, shown in Figure 11, includes a function *update* to assign to a labeled reference; it returns a relevantly typed labeled unit as a witness to the side-effect.

To allow imperative style programming with side-effects, the policy provides the *seq* function to sequence effectful computations. This function takes two arguments *e1* and *e2*, which produce values of type α and β respectively, where each of the values are protected with a label recording both their confidentiality and the effect of the computation. The body of *seq* executes each computation in sequence and returns the result of the second computation. However, *seq* is careful to record the effects of both *e1* and *e2* in the label of the result. This it does by using the label *combine l m* to assert that the confidentiality of the result is no less than either component, while the effect is no greater than the effect of each component. Notice that even though x is given a relevant type in the body of *seq*, since *seq* is a policy function, it is free to ignore x. Finally, both type variables α and β in *seq* are given U-kind; this ensures these variables cannot be instantiated with a type such as $t\{C(HIGH, EFFECT(LOW))\}$, thereby hiding the effect from the policy.

B.4 Encoding Security Automata Using Affine Types

While relevant types are useful in tracking effects, other substructural types are useful too. We briefly sketch how affine types can be used to encode policies that are expressible using security automata, following Walker [45].

Suppose we wish to enforce the following policy: a program can read data from files on the file system where each file is tagged with a label indicating its security level. As long as the program has not read data from a high-security file, it is free to send data across the network. However, once the program reads from a high-security file, it must never subsequently send data across the (low-security) network [7]. Figure 12 shows an encoding of this policy in FABLE.

```
abbrv state∼e = afn (l:lab∼e × unit{l})

policy delta φi.λs:state∼i.λf:lab.λx:lab.
    let t = match (fst s) f x with
      State(Start,_,_), Read, High ⟩⇒ State(Has_read, f, x)
      State(Start,_,_), _, _ ⟩⇒ State(Start, f, x)
      State(Has_read,_,_), Send, _ ⟩⇒ BAD
      State(Has_read,_,_), _, _ ⟩⇒ State(Has_read, f, x)
      _ ⟩⇒ BAD in
        t, {afn} (t, {t}())
```

```
policy app Λα.Λβ.φi.λs:state∼i.λlf:lab.
          λf:(φj,k.state∼State(j,lf,k) →α{k} →β){lf}.
          λlx:lab.λx:α{lx}.
  let lt,t = delta s lf lx in
    match lt with
      BAD ⟩⇒ halt
      State(_,lf,lx) ⟩⇒
        (⟨φj,k.state∼State(j,lf,k)→α{k}→β⟩ f) t x
```

Figure 12. A security automaton in FABLE using affine types.

The notation afn t represents an *affine* type t. In contrast to relevant assumptions that must be discharged at least once, affine assumptions may be discharged *at most* once. Such types are easily added to the semantics of Figure 10—we simply permit affine assumptions to be dropped at will, instead of forcing them to be used at least once. The rest of the system is unchanged.

Our strategy to enforce this kind of policy is as follows. We model the current state of the security automaton using an affine dependent product, the type of which we abbreviate as $state{\sim}e$. Since $state{\sim}e$ is affine, it cannot be duplicated; this ensures that there is a single object in the program that represents the current state. Here, the expression e is a label expression describing the current state of the automaton. Functions can be given types such as $state{\sim}e \to \alpha \to \beta$; such a function requires the current state as its first argument; furthermore, the type expresses a precondition that the current state be described by the expression e. The FABLE type checker can ensure that a function with such a type is only called in states that satisfy the state precondition.

Figure 12 shows the whole policy. It begins by giving a representation for the state object as an affine pair. The function *delta* encodes the transition relation of the automaton. Its first argument is an object representing the current state—any state is acceptable. The second argument is a label f that is intended to be a label on a function. For instance, the function that reads from the file system will have the label *Read*, while the network send function will have the label *Send*. The third argument is the label x of, say, the message that it is to be sent over the network using the *Send* function. The body of the function encodes the automata: if the state is the *Start* state, the function is *Read* and the file being read has the label *High*, then the next state is *State*(*Has_read*, f, x). This label denotes that from the current state s, a transition to the *Has_read* state occurs if the function f is called with argument x. Importantly, the *delta* function includes a clause stating that from the *Has_read* state, executing the *Send* function with any argument results in the *BAD* state. All legal program states are represented by labels where the top-level constructor is *State*.

We now turn to the *app* function, which ensures that a function's preconditions (expressed in the type of its $state{\sim}e$ argument) are met before it is called. This function, takes the current state s as its first argument. The second argument of *app* is a function label *lf* (such as *Read* or *Send*) followed by the function f itself. The last two arguments are lx and x, the labeled argument of f. The type given to f is the key. The state precondition is that f must be called in some state such that calling f with argument x results in a legal next state, i.e., the top-level constructor of the state label is *State* and not *BAD*.

To apply f, *app* must provide evidence to the FABLE type checker that f's precondition can be met. This it does by invoking *delta* with the current state and labels of f and x and inspecting what the next state of the automaton will be. If the state is *BAD*, then evidence for the precondition cannot be provided and the program halts. If, however, a legal state results, then the type of the next state can be refined (by adding an assumption to $\Omega.A$ in the last premise of (T-MATCH)) and the evidence for the call to f is provided.

C Soundness of FABLE

Definition 3 (Well-formed environment)**.** $\Omega = \Gamma; R; \varepsilon$ *is well-formed if and only if*

$$
\begin{array}{lll}
(i.) & \textit{All names bound in } \Gamma \textit{ are distinct} & \\
(ii.) & \Gamma = \Gamma_1, x:t, \Gamma_2 & \Rightarrow \quad \exists \kappa.\Omega[\Gamma = \Gamma_1, \Gamma_2] \vdash t :: \kappa \\
(iii.) & e_1 \succ e_2 \in \Gamma & \Rightarrow \quad \Omega \vdash_c e_1 : \text{rel lab} \ \wedge \ \Omega \vdash_c e_2 : \text{rel lab} \\
(iv.) & R = R_1, x:t, R_2 & \Rightarrow \quad \Gamma = \Gamma_1, x:t, \Gamma_2 \ \ / \ \ \Omega[R = \emptyset] \vdash t :: R
\end{array}
$$

e	::=	$() \mid x \mid C(\vec{e}) \mid \lambda x{:}t.e \mid e_1\, e_2 \mid \Lambda\alpha{::}\kappa.e \mid e\,[t]$	unit, variable, label, abstraction, application, type abstraction, type application
	|	$\mathsf{fix}\ f{:}t'.\lambda x{:}t.e \mid \mathsf{match}\ e\ \mathsf{with}\ p_i \Rightarrow e_i \mid \{\circ\}e \mid \{e\}e' \mid \{\mathsf{rel}\}e$	fixed point, matching, unlabeling, relabeling, relevant ascription
	|	$(\!(e)\!) \mid x_\ell \mid e_1 := e_2 \mid !e$	policy brackets, locations, assignment, deref
p	::=	$x \mid C(\vec{p})$	patterns
t	::=	$\mathsf{unit} \mid \alpha \mid \forall\alpha{::}\kappa.t \mid (x{:}t_1) \xrightarrow{\varepsilon} t_2 \mid \mathsf{lab} \mid \mathsf{lab} \sim e \mid t\{e\} \mid \mathsf{rel}\ t \mid t\ \mathsf{ref}$	unit; var; universal; function, label; labeled; relevant type; reference
α	::=	$\alpha \mid \beta \mid \ldots$	type variables
ε	::=	$0 \mid 1$	effects
κ	::=	$\mathrm{U} \mid \mathrm{M} \mid \mathrm{R}$	unlabeled, maybe-labeled and relevant kinds

Figure 13. Full syntax of FABLE.

Definition 4 (Store typing)**.** *A store M is modeled by the environment Ω ($\Omega \models M$) if all of the following are true.*

$$
\begin{array}{lllll}
(i) & dom(M) \subseteq dom(\Omega.\Gamma) & & & \\
(ii) & \Omega.\Gamma = \Gamma_1, x:t, \Gamma_2 & \Rightarrow & \exists t', e.t \in \{t'\,\mathsf{ref}, t'\,\mathsf{ref}\{e\}\} \wedge \Omega \vdash t :: M \wedge \Omega \vdash t' :: M \\
(iii) & range(M) = \{v_1, \ldots, v_n\} & & & \\
(iv) & \Gamma(x) \in \{t'\,\mathsf{ref}, t'\,\mathsf{ref}\{e\}\} & \Rightarrow & \Omega \vdash_{app} M(x) : t'
\end{array}
$$

Theorem 5 (Progress)**.** *Given $\Omega = \Gamma;\cdot;\varepsilon$, Ω well-formed, (A1) $\Omega \vdash_c e : t$, and memory M such that $\Omega \models M$ and $dom(M) = dom(\Gamma)$; then either $\exists e'.(M,e) \overset{c}{\rightsquigarrow} (M',e')$ or $\exists v.e = v$.*

Proof. By induction on the structure of (A1).

Case (T-VAR): By assumption, $\mathrm{dom}(\Gamma) = \mathrm{dom}(M)$. Thus, if (T-VAR) is applicable, then $\exists x_\ell.x = x_\ell$ then x_ℓ is a value and is thus irreducible.

Case (T-SUBR): Trivial use of the induction hypothesis.

Case (T-UNIT): $()$ is a value.

Case (T-FIX): e takes a step via (E-FIX).

Case (T-L1), (T-L2): If e is $C(\vec{v_c}, e, \vec{e})$ is an evaluation context of the form $E_c \cdot e$. The induction hypothesis is applicable since each R_i is empty. So, $e \overset{c}{\rightsquigarrow} e'$. Thus, by (E-CTX) the goal is established.

Case (T-L3), (T-L4), (T-L5), (T-L6): Straightforward by application of the induction hypothesis the first premise.

Case (T-ABS), (T-TAB): e is a value.

Case (T-TAP): We either have $e = e\,[t]$ or $v_c[t]$. In the first case, we use the induction hypothesis and apply (E-CTX) to show that $(M, e\,[t]) \overset{c}{\rightsquigarrow} (M', \cdot e'\,[t])$. In the second case, by second premise of (T-TAP) we have (A1.2) $\Omega \vdash_c v_c : \forall\alpha{::}\kappa.t$. By inversion on (A1.2), we get that v_c must be of the form $(v = \Lambda\alpha{::}k.e)$ and (E-TAP), since the types of both $\langle t \rangle u$ and $(\!(\langle t \rangle u)\!)$ are of the form $t'\{e\}$.

Case (T-APP): If e is either $\cdot e_1 e_2$ or $v_c \cdot e_2$, then, by applying the induction hypothesis to the third and fourth premises respectively, we get our result using (E-CTX). If, e is $v_c v_c'$ then, if $c = pol$, by inversion, third premise of (T-APP) (A1.3) must be an application of (T-ABS) using the same inversion principle as for (T-TAP). Thus, $v_c = \lambda x : t.e$ and (E-APP) is applicable. If $c = app$, v_c could also be $(\!(\lambda x : t.e)\!)$ since it is a legal value. In this case, reduction proceeds by (E-BAPP).

Case (T-ASN): If we have $e_1 := e_2$ or $v_1 := \cdot e_2$, then by the induction hypothesis on the first and second premise respectively, we have our result via (E-CTX). If, however, $x_\ell := v_c$, then by assumption, $\mathrm{dom}(\Omega.\Gamma) = \mathrm{dom}(M)$ and by $\Omega \models M$, $M = M_1, (\ell, v), M_2$ and from (T-ASN) we have $\Omega \vdash_c v_c : t$ with $\Omega \vdash t :: \mathrm{M}$. Now, if $c = app$ we can take a step by (E-ASNAPP); otherwise, if $c = pol$, depending on what kind of value v_c is, we take a step by (E-ASNPOL1) or (E-ASNPOL2), which are two mutually exclusive and complete rules.

Case (T-DRF): If $e = :e'$, then, by the induction hypothesis on the premise e can take a step via (E-CTX). If $e = :\ell$, then by an argument identical to (T-ASN), a step by (E-DREF) is possible.

Case (T-MATCH): If e is $\mathsf{match}\ e'\ \mathsf{with} \ldots$ then we just use the induction hypothesis on the fifth premise of (T-MATCH) ($R_i = \cdot$) and we have our result via (E-CTX). If, however, e is $\mathsf{match}\ v_c\ \mathsf{with} x_i.p_i \Rightarrow e_i$, then we must show that reduction via (E-MATCH) is applicable. To establish this, note that the third premise of (T-MATCH) requires $p_n = x_{def}$. Thus,

Additional syntactic forms used in judgments

$\Gamma ::= \cdot \mid x : t \mid \alpha :: \kappa \mid e_1 \succ e_2 \mid \Gamma_1, \Gamma_2$ — variable bindings, type variables, refinement assumptions

$R ::= \cdot \mid x : t \mid R_1, R_2$ — Context to track relevant assumptions

$\Omega ::= \Gamma; R; \varepsilon$ — Typing context triple of bindings, relevant assumptions and effect

$c ::= pol \mid app$ — Color index

$\Omega \vdash_c e : t$ — Expression e has type t in environment Ω

$$\frac{x : t \in \Omega.\Gamma \quad \Omega.R \subseteq \{x : t\}}{\Omega \vdash_c x : t}\ \text{(T-VAR)} \qquad \frac{\Omega \vdash_c e : t}{\Omega \vdash_c e : \mathsf{rel}\ t}\ \text{(T-SUBR)} \qquad \frac{\Omega.R = \emptyset}{\Omega \vdash_c () : \mathsf{unit}}\ \text{(T-UNIT)} \qquad \frac{\Omega \vdash t :: \mathrm{M} \quad t = x : t_1 \xrightarrow{1} t_2 \quad \Omega[f : t] \vdash_c \lambda x{:}t_1.e : t}{\Omega \vdash_c \mathsf{fix}\ f{:}t.\lambda x{:}t_1.e : t}\ \text{(T-FIX)}$$

$$\frac{\Omega \vdash_c R_1 \oplus \ldots \oplus R_n \quad \forall i.\Omega[R = R_i][\varepsilon = 0] \vdash_c e_i : \mathsf{rel\ lab}}{\Omega \vdash_c C(\vec{e}) : \mathsf{rel\ lab} \sim C(\vec{e})}\ \text{(T-L1)} \qquad \frac{\Omega \vdash_c R_1 \oplus \ldots \oplus R_n \quad \forall i.\Omega[R = R_i][\varepsilon = 0] \vdash_c e_i : \mathsf{lab}}{\Omega \vdash_c C(\vec{e}) : \mathsf{lab} \sim C(\vec{e})}\ \text{(T-L2)}$$

$$\frac{\Omega \vdash_c e : \mathsf{rel\ lab} \sim e'}{\Omega \vdash_c e : \mathsf{rel\ lab}}\ \text{(T-L3)} \qquad \frac{\Omega[\varepsilon = 0] \vdash_c e : \mathsf{rel\ lab}}{\Omega \vdash_c e : \mathsf{rel\ lab} \sim e}\ \text{(T-L4)} \qquad \frac{\Omega \vdash_c e : \mathsf{lab} \sim e'}{\Omega \vdash_c e : \mathsf{lab}}\ \text{(T-L5)} \qquad \frac{\Omega[\varepsilon = 0] \vdash_c e : \mathsf{lab}}{\Omega \vdash_c e : \mathsf{lab} \sim e}\ \text{(T-L6)}$$

$$\frac{\Omega[\alpha :: \kappa] \vdash_c e : t \quad t_u = \forall\, \alpha :: \kappa.t}{\Omega \vdash_c \Lambda\, \alpha :: \kappa.e : t\,(\Omega.R = \cdot) \Rightarrow (t = t_u)\mathbf{else}\ (t = \mathsf{rel}\ t_u)}\ \text{(T-TAB)} \qquad \frac{\Omega \vdash t :: \kappa \quad \Omega \vdash_c e : \mathsf{rel}\ \forall\, \alpha :: \kappa.t'}{\Omega \vdash_c e\,[t] : (\alpha \mapsto t)t'}\ \text{(T-TAP)}$$

$$\frac{\begin{array}{c}\Omega \vdash t_1 :: \kappa \qquad \Omega[R = R'][x : t_1] \vdash_c e : t_2 \\ (\kappa = \mathrm{R}) \Rightarrow (R' = \Omega.R, x : t_1)\mathbf{else}\ (R' = \Omega.R) \qquad t' = x : t_1 \xrightarrow{\Omega.\varepsilon} t_2\end{array}}{\Omega \vdash_c \lambda x : t_1.e : t\,(\Omega.R = \emptyset) \Rightarrow (t = t')\mathbf{else}\ (t = \mathsf{rel}\ t')}\ \text{(T-ABS)} \qquad \frac{\begin{array}{c}\Omega \vdash_c R_1 \oplus R_2 \qquad \sigma = (x \mapsto e_2) \qquad \varepsilon_1 \leq \Omega.\varepsilon \\ \Omega[R = R_1] \vdash_c e_1 : \mathsf{rel}\ (x : t_1 \xrightarrow{\varepsilon_1} t_2) \qquad \Omega[R = R_2] \vdash_c e_2 : t_1\end{array}}{\Omega \vdash_c e_1 e_2 : \sigma(t_2)}\ \text{(T-APP)}$$

$$\frac{\Omega \vdash_c R_1 \oplus R_2 \qquad \Omega[R = R_1] \vdash_c e_1 : t\,\mathsf{ref} \qquad \Omega[R = R_2] \vdash_c e_2 : t \qquad \Omega \vdash t :: \mathrm{M}}{\Omega[\varepsilon = 1] \vdash_c e_1 := e_2 : \mathsf{unit}}\ \text{(T-ASN)} \qquad \frac{\Omega \vdash_c e : t\,\mathsf{ref}}{\Omega[\varepsilon = 1] \vdash_c\, : e : t}\ \text{(T-DRF)}$$

$$\frac{\begin{array}{c}\Omega \vdash_c R_1 \oplus R_2 \qquad \vec{x}_i = FV(p_i) \setminus \mathrm{dom}(\Omega.\Gamma) \qquad (\vec{x}_n, p_n) = (x_{def}, x_{def}) \qquad (t_e \in \{\mathsf{rel\ lab}, \mathsf{lab}\}) \\ \Omega[R = R_1] \vdash_c e : t_e \qquad \Omega[\vec{x}_i : t_e] \vdash_c p_i : \mathsf{rel\ lab} \qquad \Omega[R = R_2][\vec{x}_i : t_e][e \succ p_i] \vdash_c e_i : t \qquad \Omega \vdash t :: \kappa\end{array}}{\Omega \vdash_c \mathsf{match}\ e\ \mathsf{with}\ p_1 \Rightarrow e_1 \ldots p_n \Rightarrow e_n : t}\ \text{(T-MATCH)} \qquad \frac{\Omega \vdash_c e : t \qquad \Omega \vdash t \cong t'}{\Omega \vdash_c e : t'}\ \text{(T-CONV)}$$

$$\frac{\Omega.R \subseteq FV(e) \quad \Omega \vdash_{pol} e : t}{\Omega \vdash_c (\!|e|\!) : t}\ \text{(T-POL)} \qquad \frac{\Omega \vdash_{pol} e : t\{e'\}}{\Omega \vdash_{pol} \{\circ\}e : t}\ \text{(T-UNLAB)} \qquad \frac{\Omega \vdash_{pol} e : t \qquad \Gamma \vdash t\{e'\}}{\Omega \vdash_{pol} \{e'\}e : t\{e'\}}\ \text{(T-RELAB)} \qquad \frac{\Omega \vdash_{pol} e : t}{\Omega \vdash_{pol} \{\mathsf{rel}\}e : \mathsf{rel}\ t}\ \text{(T-RLAB)}$$

$R = R_1 \oplus R_2 \quad ; \quad \Omega \vdash_c R_1 \oplus R_2$ — Splitting of relevant assumptions

$$\cdot = \cdot \oplus \cdot\ \text{(X-E)} \qquad \frac{R = R_1 \oplus R_2}{R, x : t = R_1, x : t \oplus R_2}\ \text{(X-L)} \qquad \frac{R = R_1 \oplus R_2}{R, x : t = R_1 \oplus R_2, x : t}\ \text{(X-R)} \qquad \frac{R = R_1 \oplus R_2}{R, x : t = R_1, x : t \oplus R_2, x : t}\ \text{(X-LR)}$$

$$\frac{\Omega.R = R_1 \oplus \ldots \oplus R_n}{\Omega \vdash_{app} R_1 \oplus \ldots \oplus R_n}\ \text{(X-IP)} \qquad \frac{relevant(\Omega.\Gamma) = R_1 \oplus \ldots \oplus R_n \qquad \{R'_1, \ldots, R'_n\} \subseteq \{R_1, \ldots, R_n\}}{\Omega \vdash_{pol} R'_1 \oplus \ldots \oplus R'_n}\ \text{(X-ESC)}$$

$\Omega \vdash t :: \kappa$ — Type t has kind κ in environment Ω

$$\Omega \vdash \mathsf{unit} :: \mathrm{U}\ \text{(K-1)} \qquad \frac{\alpha :: \kappa \in \Omega.\Gamma}{\Omega \vdash \alpha :: \kappa}\ \text{(K-V)} \qquad \frac{\Omega \vdash t_1 :: \kappa' \qquad \Omega[x : t_1] \vdash t_2 :: \kappa}{\Omega \vdash x : t_1 \to t_2 :: \mathrm{U}}\ \text{(K-F)} \qquad \frac{\Omega[\alpha :: \kappa] \vdash t :: \kappa'}{\Omega \vdash \forall\, \alpha :: \kappa.t :: \mathrm{U}}\ \text{(K-U)} \qquad \frac{\Omega \vdash t :: \mathrm{M}}{\Omega \vdash t :: \mathrm{R}}\ \text{(K-MR)}$$

$$\frac{\Omega \vdash t :: \mathrm{U}}{\Omega \vdash t :: \mathrm{M}}\ \text{(K-UM)} \qquad \Omega \vdash \mathsf{rel\ lab} :: \mathrm{R}\ \text{(K-L1)} \qquad \frac{\Omega[\varepsilon = 0] \vdash_{pol} e : \mathsf{rel\ lab}}{\Omega \vdash \mathsf{rel\ lab} \sim e :: \mathrm{R}}\ \text{(K-L2)} \qquad \Omega \vdash \mathsf{lab} :: \mathrm{U}\ \text{(K-L3)} \qquad \frac{\Omega[\varepsilon = 0] \vdash_{pol} e : \mathsf{rel\ lab}}{\Omega \vdash \mathsf{lab} \sim e :: \mathrm{U}}\ \text{(K-L4)}$$

$$\frac{\Omega \vdash t :: \mathrm{M} \qquad \Omega[\varepsilon = 0] \vdash_{pol} e : \mathsf{rel\ lab}}{\Omega \vdash t\{e\} :: \mathrm{M}}\ \text{(K-SEC)} \qquad \frac{\Omega \vdash t :: \kappa \qquad \Omega[\varepsilon = 0] \vdash_{pol} e : \mathsf{rel\ lab}}{\Omega \vdash t\{e\} :: \mathrm{R}}\ \text{(K-RSEC)} \qquad \frac{\Omega \vdash t :: \kappa}{\Omega \vdash \mathsf{rel}\ t :: \mathrm{R}}\ \text{(K-REL)}$$

$\Omega \vdash t \cong t'$ — Types t and t' are convertible

$$T ::= \bullet \mid \mathsf{rel}\bullet \mid \bullet\{e\} \mid x : \bullet \to t \mid x : t \to \bullet \mid \forall\, \alpha :: \kappa.\bullet \mid \mathsf{lab} \sim \bullet \mid t\{\bullet\} \qquad \Omega \vdash t \cong t\ \text{(TE-ID)} \qquad \frac{\Omega \vdash t \cong t'}{\Omega \vdash t' \cong t}\ \text{(TE-SYM)} \qquad \frac{\Omega \vdash t \cong t'}{\Omega \vdash T \cdot t \cong T \cdot t'}\ \text{(TE-CTX)}$$

$$\frac{\{i, j\} = \{1, 2\} \qquad \forall \sigma.\mathrm{dom}(\sigma) = FV(e_i) \qquad \Omega \vdash \sigma(e_i) : t \qquad \sigma(e_i) \overset{pol}{\rightsquigarrow} \sigma(e_j)}{\Omega \vdash T \cdot e_1 \cong T \cdot e_2}\ \text{(TE-REDUCE)} \qquad \frac{e \succ e' \in \Omega}{\Omega \vdash T \cdot e \cong T \cdot e'}\ \text{(TE-REFINE)}$$

Figure 14. Static semantics of FABLE.

Additional syntactic forms used in judgments

M	$::=$	$\cdot \mid (\ell, v) \mid M_1, M_2$	Memory: finite map from locations to values
E_c	$::=$	$\bullet \mid \bullet e \mid v_c \bullet \mid \bullet[t]$	Evaluation contexts
	$\mid$	$C(\vec{u}, \bullet, \vec{e}) \mid \mathsf{match}\ \bullet\ \mathsf{with}\ \vec{x}.p_i \Rightarrow e_i$	
	$\mid$	$\langle t \rangle \bullet \mid\ !\bullet \mid \bullet := e \mid v_c := \bullet$	
u	$::=$	$() \mid C(\vec{u}) \mid x_\ell \mid \lambda x : t.e \mid \Lambda\, \alpha ::\kappa.e$	pre-values
v_{app}	$::=$	$u \mid (\!\{e\} v_{pol}\!)$	application values: pre-values; labeled, bracketed abstractions
v_{pol}	$::=$	$u \mid \{e\} v_{pol}$	policy values: pre-values and labeled values

$(M, e) \overset{c}{\rightsquigarrow} (M', e')$ — Small-step chromatic reduction rules

$$\frac{(M,e) \overset{c}{\rightsquigarrow} (M,e')}{(M, E_c \cdot e) \overset{c}{\rightsquigarrow} (M', E_c \cdot e')}\ \text{(E-CTX)} \qquad \frac{(M,e) \overset{pol}{\rightsquigarrow} (M',e')}{(M, (\!|e|\!)) \overset{app}{\rightsquigarrow} (M', (\!|e'|\!))}\ \text{(E-POL)}$$

$$(M, (\lambda x : t.e) v_c) \overset{c}{\rightsquigarrow} (M, (x \mapsto v_c) e)\ \text{(E-APP)} \qquad (M, \Lambda\, \alpha ::\kappa.e\ [t]) \overset{c}{\rightsquigarrow} (M, [\alpha \mapsto t] e)\ \text{(E-TAP)} \qquad (M, \mathsf{fix}\ f{:}t.v) \overset{c}{\rightsquigarrow} (M, (f \mapsto \mathsf{fix} f{:}t.v) v)\ \text{(E-FIX)}$$

$$\frac{\forall i < j. \cdot \not\vdash v_c \succ p_i : \sigma_i \qquad \cdot \vdash v_c \succ p_j : \sigma_j}{(M, \mathsf{match}\ v_c\ \mathsf{with}\ \vec{b_i}.p_i \Rightarrow e_i) \overset{c}{\rightsquigarrow} (M, \sigma_j(e_j))}\ \text{(E-MATCH)}$$

$$(M, (\!|C(\vec{u})|\!)) \overset{app}{\rightsquigarrow} (M, C(\vec{u}))\ \text{(E-BLAB)} \qquad (M, (\!|()|\!)) \overset{app}{\rightsquigarrow} (M, ())\ \text{(E-BUNIT)} \qquad (M, (\!|x_\ell|\!)) \overset{app}{\rightsquigarrow} (M, x_\ell)\ \text{(E-BLOC)}$$

$$(M, (\!|\lambda x{:}t.e|\!)) \overset{app}{\rightsquigarrow} (M, \lambda x{:}t.(\!|(\lambda x{:}t.e) x|\!))\ \text{(E-BABS)} \qquad (M, (\!|\Lambda\, \alpha ::\kappa.e|\!)) \overset{app}{\rightsquigarrow} (M, \Lambda\, \alpha ::\kappa.(\!|e|\!))\ \text{(E-BTAB)}$$

$$(M, (\!|e|\!)) \overset{pol}{\rightsquigarrow} (M, e)\ \text{(E-NEST)} \qquad (M, \{\circ\}\{e\} v_{pol}) \overset{pol}{\rightsquigarrow} (M, v_{pol})\ \text{(E-UNLAB)} \qquad (M, \{\mathsf{rel}\} v_{pol}) \overset{pol}{\rightsquigarrow} (M, v_{pol})\ \text{(E-RLAB)}$$

$$\frac{M = M_1, (\ell, v_{app}), M_2}{(M, {:}\ell) \overset{c}{\rightsquigarrow} (M, v_{app})}\ \text{(E-DREF)} \qquad \frac{M = M_1, (\ell, \_), M_2}{(M, \ell := v_{app}) \overset{app}{\rightsquigarrow} ((M_1, (\ell, v_{app}), M_2), ())}\ \text{(E-ASNAPP)}$$

$$\frac{M = M_1, (\ell, \_), M_2 \qquad \nexists v_{app}.v_{pol} = v_{app}}{(M, \ell := v_{pol}) \overset{pol}{\rightsquigarrow} ((M_1, (\ell, (\!|v_{pol}|\!)), M_2), ())}\ \text{(E-ASNPOL1)} \qquad \frac{M = M_1, (\ell, \_), M_2 \qquad v_{pol} = v_{app}}{(M, \ell := v_{pol}) \overset{pol}{\rightsquigarrow} ((M_1, (\ell, v_{app}, M_2), ())}\ \text{(E-ASNPOL2)}$$

$e \succ p : \sigma$ — Expression e matches pattern p under substitution σ

$$e \succ e : \cdot\ \text{(U-EXPID)} \qquad v \succ x : x \mapsto v\ \text{(U-VAR)} \qquad \frac{\forall i. \sigma_i^* = (\sigma_0, \ldots, \sigma_{i-1}) \qquad e_i \succ \sigma_i^* e_i' : \sigma_i}{C(\vec{e}) \succ C(\vec{e'}) : \vec{\sigma}}\ \text{(U-CON)}$$

Figure 15. Operational semantics of FABLE.

it suffices to show that $v_c \succ x_{def} : \sigma$ for all label-typed values v_c (since by the fifth premise v must have type rel lab or lab). We now proceed by inversion on the fifth premise to establish that the syntactic form of v_c (and all it's sub-terms) is

Sub-case ($(\!|\langle t \rangle u|\!)$, where $t = \mathsf{lab}, \mathsf{rel\ lab}, \ldots$): Impossible, since v is a not a value and can be reduced using (E-STRIP2).

Sub-case ($(\!|C(\vec{v'})|\!)$): Impossible, since v is a not a value and can be reduced using (E-XL).

Sub-case ($(\!|(\!|v'|\!)|\!)$): Impossible, since reduction by (E-NEST) is possible.

Sub-case ($C(\vec{u})$): (U-VAR) succeeds for any value.

Case (T-POL): If we have $e = (\!|e|\!)$ we can use (E-BRAC) with the induction hypothesis in the premise. If $c = pol$ we can reduce by (E-NEST). Otherwise, if $e = (\!|v_{pol}|\!)$, then v_{pol} must be either

Sub-case $v = ()$: In which case, a reduction via (E-BUNIT) or is possible.

Sub-case $v = C(\vec{u})$: In which case, a reduction via (E-BLAB) is possible.

Sub-case $v = x_\ell$: In which case, a reduction via (E-BLOC) is possible.

Sub-case $v = \lambda x{:}t.e$: In which case, $(\!|v|\!)$ is reducible using (E-BABS) to a value.

Sub-case $v = \Lambda\alpha :: \kappa.e$: In which case, a reduction via (E-BTAB) is possible.

Sub-case $v = \langle t\{e\} \rangle u$: In which case, $(\!|v|\!)$ is an application value.

Case (T-UNLAB): Clearly, in this case, $c = pol$. If $e = \{\circ\}e'$ then by using the induction hypothesis on the second premise we have our result via (E-CTX). If $e = \{\circ\}v_{pol}$, by the premise we have that $\Omega \vdash_{pol} v_{pol} : t\{e\}$. Thus, v_{pol} must be of the form $\{e''\}v_{pol}$ and reduction can proceed using (E-UNLAB).

Case (T-RELAB): Clearly, in this case, $c = pol$. If $e = \{e''\}e'$ then by using the induction hypothesis on the second premise we have our result via (E-CTX). If $e = \{e''\}v_{pol}$, then, by definition, e is a value.

Case (T-RLAB): Clearly, in this case, $c = pol$. If $e = \{e''\}e'$ then by using the induction hypothesis on the second premise we have our result via (E-CTX). If $e = \{\mathsf{rel}\}v_{pol}$, then, by (E-RLAB) a reduction to a value is possible.

Case (T-CONV): Straightforward after the induction hypothesis is used on the first premise.

□

Lemma 6 (Preservation of well-formedness)**.** *If Ω is well-formed, and (A1) $\Omega \vdash_c e : t$ contains a sub-derivation of the form $\Omega' \vdash_c e' : t'$ then Ω' is well-formed and $\Omega' \vdash_c t' :: \kappa$. Similarly, if (A1) contains a sub-derivation of the form $\Omega' \vdash_c t' :: \kappa$, then Ω' is well-formed.*

Proof. Straightforward induction on the structure of the expression-typing and type-kinding derivations. □

Lemma 7 (Sub-coloring of derivations)**.** *If, for well-formed Ω, $\Omega \vdash_{app} e : t$, then $\Omega \vdash_{pol} e : t$.*

Lemma 8 (Substitution)**.** *Given well-formed Ω, Ω' both well-formed, and Ω'' such that $\Omega'[\Omega'']$ is well-formed. If all of the following conditions are true:*

(A1) $\Omega.R = \emptyset$

(A2) $\Omega' = \Omega[x : t_1][R_x]$, *where* $R_x = \{x : t_1\}$ *if* $\Omega \vdash t : R$; $R_x = \emptyset$ *otherwise.*

(A3) $\Omega'[\Omega''] \vdash_c e : t$

(A4) $\Omega[\varepsilon = 0] \vdash_c v : t_1$

Then, for $\sigma = x \mapsto v$,

$$\Omega[\sigma\Omega''] \vdash_c \sigma e : \sigma t_2$$

Proof. By induction on the structure of assumption (iii).

Throughout, we are free to assume $\sigma\Omega = \Omega$, since $x \notin \mathrm{dom}(\Omega)$

Case (T-VAR): We first consider the case where $R = \emptyset$. Here we have two sub-cases, depending on whether or not $x \in \mathrm{dom}(\sigma')$.

Sub-case (a): (A3) is of the form $\Omega[x : t_1][R_x = \emptyset][\Omega''] \vdash y : t_2$; $y \neq x$ and thus, $\sigma(y) = y$.

We have two further sub-cases:

Sub-case (a.i): $y : t_2 \in \Omega''.\Gamma$. In this case, $FV(t_2) \cap \mathrm{dom}(\sigma') \neq \emptyset$; thus our conclusion is of the form $\Omega[\sigma'\Omega''] \vdash y : \sigma'(t_2)$

Sub-case (a.ii): $y : t_2 \in \Omega.\Gamma$. From our initial remark, we know that $\sigma'\Omega = \Omega$; thus, $\sigma'(t_2) = t_2$. Our conclusion is of the form $\Omega[\sigma'\Omega''] \vdash y : \sigma'(t_2)$.

Sub-case (b): (A3) is of the form $\Omega[x : t_1][R_x = \emptyset][\Omega''] \vdash x : t_1$. But, $\sigma(x) = v$ and, so, from (A4), $\Omega \vdash \sigma(x) : t_1$ is trivial. Furthermore, $\sigma(t_1) = t_1$, since $\Omega \vdash t_1 :: \kappa$ and $x \notin \mathrm{dom}(\Omega)$, by well-formedness of $\Omega[\Omega'']$. Finally, we must show that $\Omega[\sigma\Omega''] \vdash v : t_1$; however, from the premise of (T-VAR), we have that $\Omega.R = \Omega''.R = \emptyset$. Thus, we establish the conclusion since weakening of $\Omega.\Gamma$ is permissible.

Now, we consider the case where $R \neq \emptyset$. Again we have two sub-cases, depending on whether or not $x \in \mathrm{dom}(\sigma)$.

Sub-case (a): (A3) is of the form $\Omega[x : t_1][R_x][\Omega''] \vdash_c y : t_2$, where $y \neq x$ and thus $R_x = \emptyset$ (since R_y must be a singleton). Again, we proceed by cases on whether the $y : t$ assumption is present in Ω or in Ω'', in a manner identical to (T-VAR), sub-case (a).

Sub-case (b): (A3) is of the form $\Omega[x : t_1][R_x = x : t_1][\Omega''] \vdash x : t_1$. Again, we proceed similar to (T-VAR) sub-case (b). This time, we can conclude that $\Omega''.R = \emptyset$ since $R_x \neq \emptyset$ and (T-RVAR) requires only a single relevant assumption in the context. Thus, using weakening $\Omega.\Gamma$ by adding $\Omega''.\Gamma$ we reach our conclusion.

Case (T-SUBR): Straightforward use of induction hypothesis on the premise.

Case (T-UNIT): Trivial.

Case (T-L1): We use the induction hypothesis on each of the n-premises, obtaining $\Omega[\sigma\Omega''][R = \sigma R_i] \vdash_c \sigma(e_i) : \mathsf{rel\ lab}$ for the ith premise. For the conclusion, we note that $\sigma(C(\vec{e})) = C(\sigma(\vec{e}))$ and obtain $\Omega[\sigma\Omega''] \vdash_c \sigma(C(\vec{e})) : \sigma(\mathsf{rel\ lab} \sim C(\vec{e}))$, the desired result by noting that, by (A1), all of the R_i's are present in Ω'' and thus $\Omega[\sigma\Omega''] \vdash_c \sigma R_1 \oplus \ldots \oplus \sigma R_n$. The relevant assumptions can therefore be partitioned according in a manner that makes the induction hypothesis applicable.

Case (T-L2): Similar to (T-L1).

Case (T-L3), (T-L4), (T-L5), (T-L6): Straightforward use of induction hypothesis on the premise.

Case (T-FIX): We have $f \notin \mathrm{dom}(\sigma)$. Thus, $\sigma(\mathsf{fix}\ f.v) = \mathsf{fix}\ f.\sigma(v)$. Now, we can use the induction hypothesis on the second premise to establish the conclusion.

Case (T-TAB): Since $\alpha \notin \mathrm{dom}(\sigma)$ we can use the induction hypothesis on the first premise of (A3). However, we must be careful in the conclusion with R. If, $R_x \neq \emptyset$ in (A3); and $\Omega''.R = \emptyset$, then, where initially we have $\Omega'[\Omega''] \vdash_c e : \mathsf{rel} t_2$, in the conclusion we have $\Omega[\sigma\Omega''] \vdash_c \sigma e : \sigma' t_2$. To restore the type in the conclusion to $\mathsf{rel}\sigma'(t_2)$, we can use (T-SUBR), if necessary.

Case (T-ABS): Our goal is to show, via (T-ABS), $\Omega[\sigma\Omega''] \vdash_c \lambda y : \sigma'(s_1).\sigma'(e) : \sigma'(t)$, since $\mathrm{dom}(\sigma)$ cannot mention y. From (A3), we have $\Omega'[\Omega''] \vdash_c \lambda y : s_1.e : t$ with the premise, $\Omega'[\Omega''][y : s_1][R_y] \vdash_c e : s_2$. However, this context is of the form $\Omega'[\Omega'']$, and is well-formed by Lemma 6. Thus, the induction hypothesis is applicable and we obtain

$$(T1) \qquad \Omega[\sigma\Omega''][y : \sigma s_1][\sigma R_y] \vdash_c \sigma e : \sigma s_2$$

Thus, to reach our goal, we use this last judgment (T1) in the premise of (T-ABS). Finally, if $R_y = \Omega''.R = \emptyset$ and $R_x \neq \emptyset$, then we conclude with an application of (T-SUBR) to ensure that our conclusion preserves the relevant qualifier on the function type (similar to the conclusion of (T-TAB).

Case (T-TAP): We use Lemma 9 (proved by simultaneous induction together with this Lemma) to establish that $\Omega[\sigma\Omega''] \vdash \sigma t :: \kappa$. Now, we use the induction hypothesis on the second premise, and the conclusion is straightforward.

Case (T-APP): From the assumption (A1) that $\Omega.R = \emptyset$, we conclude that $R_1 \oplus R_2 = (R_x \cup \Omega''.R)$. We can use the induction hypothesis on the first premise of (A3) to establish

$$\Omega[\sigma\Omega''][R = \sigma(R_1 \setminus R_x)] \vdash_c \sigma e_1 : \mathsf{rel}(y : \sigma t_1) \xrightarrow{\varepsilon} \sigma t_2$$

Similarly, for the second premise we obtain

$$\Omega[\sigma\Omega''][R = \sigma(R_2 \setminus R_x)] \vdash_c \sigma e_2 : \sigma t_1$$

For the conclusion, we get $\Omega[\sigma\Omega''] \vdash_c \sigma(e_1 e_2) : (y \mapsto \sigma e_2)(\sigma t_2)$. However, since $y \notin \mathrm{dom}(\sigma)$ and $FV(\mathrm{range}(\sigma)) = \emptyset$, the substitutions $y \mapsto \sigma e_2$ and σ commute to give us the necessary conclusion of $\sigma(y \mapsto e_2)t_2$.

Case (T-ASN): Straightforward application of induction hypothesis to the first two premises, again noting that by (A1) all the relevant assumptions are from Ω''.

Case (T-DRF): Straightforward application of induction hypothesis.

Case (T-FX): Straightforward application of induction hypothesis.

Case (T-MATCH): As with (T-APP) $R_1 \oplus R_2 = R_x \cup \Omega''.R$. The first four premises are trivial to re-establish since $\mathrm{dom}(\sigma')$ does not include any of $\vec{x_i}$. For the fifth premise, we use the induction hypothesis and restore $\sigma e : t_e$. The sixth and seventh premises is also established by the induction hypothesis. For the final premise, we rely on the companion Lemma 9.

Case (T-CONV): Applying the induction hypothesis to the first premise, we obtain $\Omega[\sigma\Omega''] \vdash_c \sigma e : \sigma t$. We proceed by cases on the structure of (A3.2) $\Omega \vdash t \cong t'$ from the second premise of (A3), to show that $\sigma\Omega \vdash \sigma t \cong \sigma t'$.

Sub-case (TE-ID): Trivial.

Sub-case (TE-CTX): Induction.

Sub-case (TE-REFINE): $e \succ e' \in \Omega \Rightarrow \sigma e \succ \sigma e' \in \sigma\Omega$.

Sub-case (TE-REDUCE): By construction, we have that $\forall \sigma'.\sigma' e \overset{c}{\rightsquigarrow} \sigma' e'$.

Case (T-POL): We have $\Omega'[\Omega''] \vdash_c (\!|e|\!) : t$ with $\Omega'[\Omega''] \vdash_{pol} e : t$ in the premise. Now, if (A4) is $\Omega \vdash_{pol} v : t_1$, then we can use the induction hypothesis to establish $\Omega[\sigma\Omega''] \vdash_{pol} \sigma(e) : \sigma(t)$ and conclude with (T-POL). However, if (A4) is $\Omega \vdash_{app} v : t_1$, then we must first use Lemma 7 before proceeding as before.

Case (T-UNLAB), (T-RELAB), (T-RLAB): Induction hypothesis on the first premise. □

Lemma 9 (Substitution for kinding judgment)**.** *Given well-formed Ω, Ω' both well-formed, and Ω'' such that $\Omega'[\Omega'']$ is well-formed. If all of the following conditions are true:*

(A1) $\Omega.R = \emptyset$

(A2) $\Omega' = \Omega[x : t_1][R_x]$, *where* $R_x = \{x : t_1\}$ *if* $\Omega \vdash t : R$; $R_x = \emptyset$ *otherwise.*

(A3) $\Omega'[\Omega''] \vdash_c t :: \kappa$

(A4) $\Omega[\varepsilon = 0] \vdash_c v : t_1$

Then, for $\sigma = x \mapsto v$,

$$\Omega[\sigma\Omega''] \vdash_c \sigma(t) :: \kappa$$

Proof. By simultaneous induction with cases of Lemma 8 on the structure of assumption (iii).

Case (K-1), (K-L1), (K-L3): Trivial.

Case (K-V): $\sigma = (x \mapsto v)$; Thus, $\alpha \in \Omega[\sigma\Omega'']$.

Case (K-F): Induction hypothesis on first premise; then, since $\Omega'[\Omega''][y : t_1]$ is well-formed, we must have $y \neq x$.

Case (K-F): Induction hypothesis on each premise.

Case (K-UM), (K-MR): Induction hypothesis.

Case (K-L2): We use the IH of Lemma 8 (proved simultaneously) to show $\Omega[\sigma\Omega''][\varepsilon = 0] \vdash_c \sigma e : \mathsf{rel\ lab}$. If (A4) is of the form $\Omega \vdash_{app} v : t_1$, then we first use Lemma 7 before proceeding.

Case (K-L4): Similar to (K-L2).

Case (K-LSEC), (K-RSEC): Induction hypothesis on the first premise, and then similar to (K-L2) on the second premise.

□

Corollary 10 (Contraction of assumptions)**.** *For well-formed Ω and $\Omega[e_1 \succ e_2]$, if both of the following are true*

i. $\Omega[\vec{x} : \mathsf{lab}][e_1 \succ e_2][\Omega''] \vdash_c e : t$

ii. $e_1 \succ e_2 : \sigma$

iii. $dom(\sigma) = \vec{x}$

Then, $\Omega[\sigma\Omega''] \vdash_c \sigma e : \sigma' t$, where $\sigma' \in \{\emptyset, \sigma\}$.

Proof. If (i) does not contain a sub-derivation with an application of (T-CONV) using (TE-REFINE), then the assumption $e_1 \succ e_2$ is redundant and we conclude with $\sigma' = \emptyset$.

If (i) does contain an application of (TE-REFINE) then, we can use the substitution lemma to establish our result. By Lemma 8, we have that (T1) $\Omega[\sigma e_1 \succ \sigma e_2][\sigma\Omega''] \vdash_c \sigma e : \sigma t$. However, by construction of pattern matching and by assumption (ii), we have $\sigma e_1 = \sigma e_2$. Thus, every relevant application of (TE-REFINE) in (T1) that concludes with $\sigma T \cdot e_1 \cong \sigma T \cdot e_2$ can be replaced by an application of (TE-ID) in the result $\Omega[\sigma\Omega''] \vdash_c \sigma e : \sigma t$. □

Lemma 11 (Type substitution)**.** *Given well-formed Ω, Ω' both well-formed, and Ω'' such that $\Omega'[\Omega'']$ is well-formed. If all of the following conditions are true:*

(A1) $\Omega.R = \Omega.A = \emptyset$

(A2) $\Omega' = \Omega[\alpha :: \kappa]$

(A3) $\Omega'[\Omega''] \vdash_c e : t$ *or* $\Omega'[\Omega''] \vdash t :: \kappa$

(A4) $\Omega \vdash t' :: \kappa$

(A5) $\sigma = \alpha \mapsto t'$

Then,

$$\Omega[\sigma\Omega''] \vdash_c \sigma e : \sigma t$$

or

$$\Omega[\sigma\Omega''] \vdash_c \sigma t :: \kappa$$

Proof. By induction on the structure of (A3).

Case (T-VAR): If $x : t \in \Omega$, then by well-formedness, α is not free in t and $\sigma(t) = t$. If $x : t \in \Omega'$, then $x : \sigma(t) \in \sigma(\Omega')$ and we have the conclusion using (T-VAR).

Case (T-SUBR): Induction hypothesis.

Case (T-UNIT): Trivial.

Case (T-FIX): The second premise is of the form

$$\Omega[\alpha :: \kappa][\Omega''][f : t] \vdash_c v : t$$

Applying the induction hypothesis to this premise, we have

$$\Omega[\sigma(\Omega'')][f : \sigma(t)] \vdash_c \sigma(v) : \sigma(t)$$

This is sufficient to use (T-FIX) to establish the goal:

$$\Omega[\sigma(\Omega'')] \vdash_c \sigma \mathsf{fix}\ f.v : \sigma(t)$$

using Lemma 6 for the first premise.

Case (T-L1), (T-L2), (T-L3), (T-L4), (T-L5), (T-L6), (T-FX): Induction hypothesis.

Case (T-TAB): We have

$$\frac{\Omega[\alpha][\Omega''][\beta :: \kappa] \vdash_c e : t \qquad t_u = \forall \beta :: \kappa.t}{\Omega[\alpha][\Omega''] \vdash_c \Lambda \beta :: \kappa.e : t(\Omega.R = \cdot) \Rightarrow (t = t_u)\textbf{else}\ (t = \mathsf{rel} t_u)} \text{ (T-TAB)}$$

with $\alpha \neq \beta$ by Lemma 6. Thus, by the induction hypothesis we have

$$\Omega[\sigma(\Omega'')][\beta :: \kappa] \vdash_c \sigma(e)$$

which is sufficient to establish the goal:

$$\Omega[\sigma(\Omega'')][\beta :: \kappa] \vdash_c \Lambda \beta :: \kappa.\sigma(e) : \sigma(t)$$

Case (T-TAP):

$$\frac{\Omega[\alpha][\Omega''] \vdash t :: \kappa \qquad \Omega[\alpha][\Omega''] \vdash_c e : \mathsf{rel} \forall \beta :: \kappa.t'}{\Omega[\alpha][\Omega''] \vdash_c e\,[t] : (\beta \mapsto t)t'} \text{ (T-TAP)}$$

By the induction hypothesis (using the right side of the dijunct in (A3)), we have

$$\Omega[\sigma(\Omega'')] \vdash \sigma(t) :: \kappa$$

and, using the left side of the dijunct in (A3), we have

$$\Omega[\sigma(\Omega'')] \vdash_c \sigma(e) : \mathsf{rel} \forall \beta :: \kappa.\sigma(t')$$

Together, this is sufficient to establish the goal.

Case (T-ABS):

$$\frac{\begin{array}{cc}\Omega[\alpha :: \kappa][\Omega''] \vdash t_1 :: \kappa & \Omega[\alpha :: \kappa][\Omega'']'[R = R'][x : t_1][\varepsilon = \varepsilon'] \vdash_c e : t_2 \\ (\kappa = \mathrm{R}) \Rightarrow (R' = \Omega[\alpha :: \kappa][\Omega''].R, x : t_1)\textbf{else}\ (R' = \Omega[\alpha :: \kappa][\Omega''].R) & t' = x : t_1 \xrightarrow{\varepsilon'} t_2\end{array}}{\Omega[\alpha :: \kappa][\Omega''] \vdash_c \lambda x : t_1.e : t(\Omega[\alpha :: \kappa][\Omega''].R = \emptyset) \Rightarrow (t = t')\textbf{else}\ (t = \mathsf{rel} t')} \text{ (T-ABS)}$$

Using the induction hypothesis on premise 1 we have

$$\Omega[\sigma(\Omega'')] \vdash \sigma(t_1) :: \kappa$$

and using IH on premise 2, we have

$$\Omega[\sigma(\Omega'')]'[R = \sigma(R')][x : \sigma(t_1)][\varepsilon = \varepsilon'] \vdash_c \sigma(e) : \sigma(t_2)$$

This is sufficient to conclude

$$\Omega[\sigma(\Omega'')] \vdash_c \lambda x : \sigma(t_1).\sigma(e) : \sigma(t)$$

Case (T-ASN), (T-DRF), (T-APP), (T-MATCH): Induction hypothesis.

Case (T-CONV): Induction hypothesis on the first premise. We must establish

$$\Omega[\alpha :: \kappa][\Omega''] \vdash t \cong t' \Rightarrow \Omega[\sigma(\Omega'')] \vdash \sigma(t) \cong \sigma(t')$$

For (TE-REDUCE), we can establish $\Omega[\sigma(\Omega'')] \vdash \sigma'(\sigma(e)) : \mathsf{lab}$ by the induction hypothesis and $\sigma'(\sigma(e)) \overset{c}{\rightsquigarrow}^* \sigma'(\sigma(e'))$ since, by erasure, type substitution does not affect reduction.

For (TE-REFINE), we proceed similarly to (T-VAR) by cases on whether $e \succ e'$ appears in Ω or Ω''.

Case (T-POL): Induction hypothesis. (Note, the statement of the Lemma does not impose any restrictions on the color of the derivation.)

Case (T-UNLAB), (T-RELAB), (T-RLAB): Induction hypothesis.

We now proceed to the case of $\Omega \vdash t :: \kappa$

Case (K-1), (K-L1), (K-L3): Trivial.

Case (K-2): By assumption (A4) if $\alpha = \beta$; otherwise $\beta :: \kappa \in \Omega[\sigma(\Omega'')]$.

Case (K-F), (K-U), (K-UM), (K-REL), (K-MR): Induction hypothesis.

Case (K-L2), (K-L4), (K-SEC), (K-RSEC): Induction hypothesis using the left-side of the disjunct (A3). □

Lemma 12 (Strong normalization for type-level expressions)**.** *Given* $\cdot \vdash_c t\{e\} :: \kappa$*; then* $e \overset{c}{\rightsquigarrow}^* v$.

Proof. Type-level expressions are from pure System-F (no fix). The result follows from Girard's proof of strong normalization of System F [20]. □

Theorem 13 (Preservation)**.** *Given* $\Omega \equiv \Gamma; \cdot; \varepsilon$ *and memory* M *such that* $\Omega \models M$*, and* e *such that*

$$(A1)\ \ \Omega \vdash_c e : t \qquad \textit{and} \qquad (A2)\ \ (M, e) \overset{c}{\rightsquigarrow} (M', e')$$

Then, $\Omega \models M'$ *and* $\Omega \vdash_c e' : t$.

Proof. By induction on the structure of the derivation (A2). We examine all the side-effect free cases first, (i.e., where $M = M'$) where the obligation of $\Omega \models M'$ is satisfied by assumption. In each case, upon inversion, one or more of the subtyping judgments is also applicable. We treat all the subtyping cases uniformly and separately.

Case (E-TAP): By inversion, (A1) must conclude with an application of (T-TAP), $\Omega \vdash_c \Lambda\alpha::k.e\ [t_1] : (\alpha \mapsto t_1)t_2$, where $t = (\alpha \mapsto t_1)t_2$. Inverting the second premise of (T-TAP), we find an application (A3) of (T-TAB) of the form $\Omega[\alpha::\kappa] \vdash t_2$. In order to re-establish that the conclusion of (A2) is well-typed, we must show that $\Omega \vdash_c (\alpha \mapsto t_1)e : t$. However, this is a straightforward application of the type-substitution lemma, Lemma 11, to (A3), where Ω'' is empty.

Case (E-FIX): By inversion, (A1) concludes with (T-FIX). By applying the substitution Lemma 8 to the second premise of (T-FIX), taking $\sigma = (f \mapsto \mathsf{fix}\ f.v)$. From the first premise of (T-FIX) we have that $f \notin FV(t)$. Thus, $\sigma(t) = t$ and we have the desired result.

Case (E-APP): : By inversion, we have that (A1) concludes with an application of (T-APP) of the form $\Omega \vdash_c \lambda x{:}t_1.e\ v_2 : t$. From the fourth premise of (T-APP), we have (A1.4) $\Omega \vdash_c \lambda x{:}t_1.e : \mathsf{rel}(x : t_1 \xrightarrow{\varepsilon_1} t_2$, since $\Omega.R = R_1 = R_2 = \cdot$, with $t = (x \mapsto v_2)t_2$. Inverting (A1.4), we find that it is an application of (T-ABS). From the second premise of (T-ABS) in (A1.4), we have (A1.4.2) $\Omega[R = R'][x : t_1] \vdash e : t_2$, and from the fifth premise of (A1) we have (A1.5) $\Omega \vdash_c v_2 : t_1$.

Our goal, then, is to show that $\Omega \vdash_c (x \mapsto v_2)e : (x \mapsto v_2)t_2$. Using (A1.4.2) and (A1.5), the substitution lemma (Lemma 8) is applicable with Ω'' empty.

Case (E-MATCH): : By inversion, (A1) is an application of (T-MATCH), $\Omega \vdash_c \mathsf{match}\ v\ \mathsf{with}\ldots : t$.

The third premise of (A2) gives us (A2.1) $v \succ p_j : \sigma_j$ and from the premise (A1) we get (A1.5) $\Omega \vdash_c v : t_v$; by (T-LAB) all sub-terms of v must also be values of type t_v. Thus, for each $x \in FV(p_j)$, $\Omega \vdash_c \sigma_j(x) : t_v$. Thus, by repeated application of the substitution lemma to the second last premise of (A1), (A1.7) $\Omega[e \succ p_j][\vec{x_j} : \mathsf{rel\ lab}] \vdash_c e_j : t$, we have that (T1) $\Omega[e \succ p_j] \vdash_c \sigma_j(e_j) : \sigma_j(t)$.

Now, we must show that from (T1) and (A2.1), that (T2) $\Omega \vdash_c \sigma_j(e_j) : \sigma'_j(t)$. This we can establish by an application of Corollary 10 taking $\Omega'' = \cdot$.

Finally, by the last premise of (A1), (A1.8) $\Omega \vdash_c t :: \kappa$, and noting that $\vec{x_j} \notin \mathrm{dom}(\Omega.\Gamma)$, and since $\sigma'_j = \emptyset\mathrm{dom}(\sigma_j) = \vec{x_j}$, we have that $\sigma_j(t) = t$. Thus, from (T2), we have $\Omega \vdash_c \sigma_j(e_j) : t$, which is our goal.

Case (E-BUNIT): Use (T-1) for the conclusion.

Case (E-BLAB): By inversion, (A1) is (T-POL) with (T-L1) or (T-L2) in the premise. For the non-trivial case, we must show that $\Omega \vdash_{pol} C(\vec{u}) : \mathsf{rel\ lab} \Rightarrow Omega \vdash_{app} C(\vec{u}) : \mathsf{rel\ lab}$, which is straightforward by induction since each sub-term is pre-value of the form $C'(\vec{u'})$, with the base case being $\Omega \vdash_{app} C : \mathsf{lab}$.

Case (E-BLOC): Use (T-VAR) for the conclusion.

Case (E-BABS): We have

$$(A1)\frac{\Omega.R \subseteq FV(\lambda x{:}t_1.e) \qquad \Omega \vdash_{pol} \lambda x : t_1.e : t_1 \xrightarrow{\varepsilon} t_2}{\Omega \vdash_{c} (\!|\lambda x : t_1.e|\!) : t_1 \xrightarrow{\varepsilon} t_2}$$

To conclude, we must show

$$(G)\frac{\Omega[x : t_1] \vdash_{c} (\!|(\lambda x{:}t_1.e)x|\!) : t_2}{\Omega \vdash_{c} \lambda x{:}t_1.(\!|(\lambda x : t_1.e)x|\!) : t}$$

The interesting case is when $t_1 = \mathsf{rel}\ t_1'$. Then, to satisfy the premise of (G), we must show that $\Omega.R \cup \{x\} \subseteq FV((\lambda x : t_1.e)x)$, which is true by construction, since we have $\Omega.R \subseteq FV(\lambda x : t_1.e)$ from (A1) and x is free in $(\lambda x : t_1.e)x$.

Case (E-BTAB): Trivial.

Case (E-POL): By inversion, (A1) is (T-POL), with $\Omega \vdash_{pol} e : t$ in the premise. Thus, we can use the induction hypothesis to establish $\Omega \vdash_{pol} e' : t$. Which is sufficient, even if (A1) is of the form $\Omega \vdash_{app} (\!|e|\!) : t$.

Case (E-NEST): By inversion, $\Omega \vdash_{pol} (\!|e|\!) : t$ is an instance of (T-POL) with $\Omega \vdash_{pol} e : t$ in the premise, which is sufficient for the conclusion.

Case (E-UNLAB): By inversion, (A1) must be (T-UNLAB), with

$$\frac{\dfrac{\Omega \vdash_{pol} v_{pol} : t}{\Omega \vdash_{pol} \{e'\}v_{pol} : t\{e'\}}}{\Omega \vdash_{pol} \{\circ\}\{e'\}v_{pol} : t}$$

To conclude, we use the nested premise $\Omega \vdash_{pol} v_{pol} : t$.

Case (E-RLAB): By inversion, (A1) must be (T-RLAB), with

$$\frac{\Omega \vdash_{pol} v_{pol} : t}{\Omega \vdash_{pol} \{\mathsf{rel}\}v_{pol} : \mathsf{rel}t}$$

To conclude, we use

$$\frac{\Omega \vdash_{pol} v_{pol} : t}{\Omega \vdash_{pol} v_{pol} : \mathsf{rel}\ t}\ \text{(T-SUBR)}$$

Case (E-DREF): By inversion, (A1) is $\Omega \vdash_{c} :l : t$ with (A1.1) $\Omega \vdash_{c} l : t\,\mathsf{ref}$ in the premise. By well-formedness of Ω and $\Omega \models M$ we have that $\Omega \vdash v : t$.

Case (E-ASNAPP): By inversion, (A1) is (T-ASN). To establish the well-formedness of M' we use (A1.2) and (A1.3) along with clause (iv) of well-formedness of M to show that v_{app} and $v_{app}{}'$ have the same type. Finally, since $\Omega \vdash_{app} v_{app}$.

Case (E-ASNPOL1): Similar to (E-ASNAPP), except given $\Omega \vdash_{pol} v_{pol} : t$, we can construct $\Omega \vdash_{app} (\!|v_{pol}|\!) : t$

Case (E-ASNPOL2): Similar to (E-ASNAPP).

Case (E-CTX): We proceed by cases on the syntactic structure of evaluation contexts. Case $E = e$ is already completed.

Sub-case ($E = C(\vec{v}, \cdot e_j, \vec{e})$): The judgment (A1) has the form $\Omega \vdash_{c} E : t$, using (T-L1) or (T-L2). For (T-L2), we use the induction hypothesis to establish that $\Omega[R = R_j] \vdash_{c} e_j' : \mathsf{lab}$, and replace the appropriate premise in (A1) for the conclusion. For (T-L1), we have $t = \mathsf{lab}\ C(\vec{v}, e_j, \vec{e})$, while using the induction hypothesis alone we can establish (T1)

$\Omega \vdash_c E \cdot e'_j : \mathsf{lab}\, C(\vec{v}, e'_j, \vec{e})$. But, by (A2), we have that $e_j \overset{c}{\rightsquigarrow} e'_j$ and $FV(e_j) = \emptyset$. Thus, to re-establish the conclusion, we use (T-CONV) with (T1) for the first premise, and (TE-CTX) using (TE-REDUCE) with (A2) in the second premise.

Sub-case ($E = \cdot e_1 e_2$): By inversion, we have (A1) an instance of (T-APP), with (A1.4) $\Omega \vdash_c e_1 : t_1$. To re-establish the hypothesis, we will apply (T-APP) again. By the induction hypothesis we have that $\Omega \vdash_c e'_1 : t_1$. By assumption, since e_2 is unchanged, the remaining premises are unchanged.

Sub-case ($E = v \cdot e$): Similar to the previous case, except using the induction hypothesis for the second premise of (T-APP).

Sub-case ($E = \{e'\} \cdot e$): Straightforward use of the induction hypothesis with (T-RELAB).

Sub-case ($E = \{\circ\} \cdot e$): Straightforward use of the induction hypothesis with (T-UNLAB).

Sub-case ($E = \mathsf{match}\ \cdot e\ \mathsf{with} \ldots$): The top-level typing judgment is (T-MATCH). We use the induction hypothesis on the sixth premise. The other premises are unchanged.

Sub-case ($E = :\bullet \mid \bullet := e \mid v := \bullet$): Trivial from use of induction hypothesis.

Case OTHER: : In each case, it is possible that the top-level typing judgment be one of (T-SUBR), (T-CONV), (T-L3), (T-L4), (T-L5) or (T-L6); these are all the "subtyping" judgments in the static semantics.

Sub-case (T-SUBR), (T-CONV), (T-L3), (T-L5): Trivial use of the induction hypothesis.

Sub-case (T-L4), (T-L6): Using the induction hypothesis in the premise, it is easy to establish the (T1) $\Omega \vdash_c e' : \mathsf{lab} \sim e'$. However, by (A2) we have $e \overset{c}{\rightsquigarrow} e'$, and $FV(e) = \emptyset$. Thus, following sub-case (T-L1) in (E-CTX), we establish the conclusion using (T-CONV) using (TE-REDUCE) with (A2) in the premise.

□

Lemma 14. *Given $\Omega[R = x : t]$ well-formed and $\Omega[R = x : t] \vdash_{app} e : t$; then, $x \in FV(e)$.*

Proof. By induction on the structure of the typing derivation. The only interesting case is (T-VAR). There, $R = x : t$ we must have $e = x$ and $x \in FV(x)$. □

Theorem 15 (Discharging relevant assumptions)**.** *Given $\Omega = \Gamma; R; \varepsilon$ well-formed, such that $R = x : t_x$; $t_x = t'\{e_l\}$ and $\cdot \not\vdash t_x :: M$; and an expression e such that,*

(A1) $\Omega \vdash_{app} e : t$

(A2) $\cdot \vdash_{app} v : t_x$

(A3) $\sigma = (x \mapsto v)$

(A4) $(M, \sigma e) \overset{c}{\rightsquigarrow}^* (M', v')$

Then, either (i) v' is a value containing v as a sub-term, and $\cdot \vdash \sigma t :: R$ and $\cdot \not\vdash \sigma t :: M$. or, (ii) (A4) contains a sub-derivation of the form: $(\!|\lambda x : t.e|\!)v'' \overset{c}{\rightsquigarrow} (\!|\lambda x : t.ev''|\!)$ (E-BAPP) where v is a sub-term of v''; or (iii) $\sigma(e)$ contains a subterm $(\!|e'|\!)$ where v is a subterm of e'.

Proof. By induction on the structure of (A1).

Case (T-VAR): $\sigma e = v$, since $R = x : t_x$; and v is irreducible. By (A2) and clause 4 of well-formedness of Ω, we have that $\cdot \vdash t_x :: \mathrm{R}$ and by assumption $\cdot \not\vdash t_x :: \mathrm{M}$.

Case (T-SUBR): Induction hypothesis on the premise.

Case (T-UNIT): Impossible, since (T-UNIT) requires $R = \emptyset$.

Case (T-FIX): Inverting the third premise, we find an application of (T-ABS) with $\Omega.R \neq 0$ (by assumption); but this requires $t = \mathsf{rel}(x : t_1 \to t_2)$, which is impossible. The fixed-point combinator is not applicable in a context with non-empty relevant assumptions.

Case (T-L1): $e = C(\vec{e})$. If σe is a value, then by Lemma 14 v is a sub-term of σe and the conclusion produces the type $\mathsf{rel}\ \mathsf{lab} \sim e$, which, by (K-L2) must be given :: R. Otherwise, x appears in the ith subterm, and e does not diverge, we use the I.H. on the ith premise to ensure that the reduction sequence contains the necessary sub-derivation.

Case (T-L2): Similar to (T-L1).

Case (T-L3), (T-L4), (T-L5), (T-L6), (T-FX): Straightforward with induction hypothesis.

Case (T-TAB): Since $v = \Lambda\alpha::\kappa.e$ is a value, we must show that (i) holds. However, since $\Omega.R = x : t$, the side condition ensures that $t = \mathsf{rel} t_u :: \mathrm{R}$. Finally, by Lemma 14, x is free in e and so σe contains v as a sub-term.

Case (T-TAP): $e = e'[t']$, with $\Omega \vdash_{app} e' : t'$ in the premise (A1.2). By the I.H., if $\sigma e' \overset{c}{\rightsquigarrow} \sigma v'$ contains the necessary sub-derivation then we are done. Otherwise, by the substitution lemma (Lemma 8) we have that $\cdot \vdash_c \sigma e'[t'] : \sigma t$ and by Theorem 13, we have $\cdot \vdash_{app} \sigma v'[t'] : \sigma t$ and, by the converse of Lemma 8, (A1') $\Omega \vdash_{app} v'[t'] : t$. Now, by inverting (A1') we find that v' must be of the form $\Lambda\alpha::\kappa.e$ or $(\!|v''|\!)$. However, this latter possibility is handled trivially since (T-POL) requires $\Omega.R \subseteq FV(v'')$ and so condition (iii) is satisfied by $\sigma(v'')$. Thus, the second premise of (A1'), (A1'.2) must be an application of (T-TAB). We conclude by constructing the necessary derivation by starting with (E-TAP) and then using Lemma 11 followed by the induction hypothesis on (A1'.2)

Case (T-ABS): Similar to (T-TAB).

Case (T-APP): $e = e_1 e_2$. If x is free in e_1 and the reduction sequence $\sigma e_1 \overset{c}{\rightsquigarrow}^* \sigma v_1$ contains the necessary sub-derivation then we are done. Likewise, if x is free in e_2 and the reduction sequence $\sigma e_2 \overset{c}{\rightsquigarrow}^* \sigma v_2$. If not, by the substitution lemma (Lemma 8) we have that $\cdot \vdash_c \sigma(e_1 e_2) : \sigma t_1$ and by Theorem 13, we have $\cdot \vdash_{app} \sigma(v_1 v_2) : \sigma t$ and, by the converse of Lemma 8, (A1') $\Omega \vdash_{app} v_1 v_2 : t$. Now, by inversion of (A1') we have that v_1 is either of the form $\lambda y : t_1.e$ or $(\!|\lambda y : t_1.e|\!)$.

Sub-case ($v_1 = \lambda y : t_1.e$): The fourth premise of (A1'), (A1'.4) is an application of (T-ABS). We proceed by cases on the structure of R_1 and R_2 in (A1').

If $R_1 = \emptyset$ in (A1'), then $R_2 = x : t$, by the definition of (X-R), since $R = R_1 \oplus R_2$. Thus, applying the induction hypothesis to the fifth premise of (A1'), we find that $\Omega[R_2] \vdash_{app} \sigma v_2 : \sigma t_1$ and $\cdot \vdash t_1 :: \mathrm{R}$. Thus, in the second premise of (A1'.4), we have $\Omega[R = y : t_1] \vdash_{app} e : t_2$. For the conclusion, we construct the necessary derivation using (E-APP) and then use Lemma 8 followed by the induction hypothesis on (A1'.4) (applied to the relevant assumption $y : t_1$).

If $R_2 = \emptyset$ then $R_1 = x : t$ and we proceed as previously, but this time applying the induction hypothesis to to the relevant assumption $x : t_1$ in (A1'.4).

Finally, if $R_1 = R_2 = x : t$, then (A1'.4) has the form $[x : t_1, y : t_1] \vdash_{app} e : t_2$ and the induction hypothesis is not directly applicable. To solve this, we consider (A1'.4') $[y : t_1] \vdash_{app} \sigma e : \sigma t_2$, which is well-formed by the substitution lemma and apply the induction hypothesis to (A1'.4').

Sub-case ($v_1 = (\!|v_1'|\!)$): (A1'.4) is an application of (T-POL) and so $R_1 = \emptyset$ (otherwise $\sigma(e)$ trivially satisfies condition (iii)) and $R_2 = x : t_x$ in (A1'). By Lemma 14, v is a sub-term of σv_2. Thus, $\sigma(\!|v_1'|\!) v_2 \overset{c}{\rightsquigarrow} \sigma(\!|v_1'(\!|v_2|\!)|\!)$ (B-APP), satisfying condition (ii).

Case (T-ASNAPP), (T-ASNPOL1), (T-ASNPOL2): Induction hypothesis on second or third premise, as necessary.

Case (T-DRF), (T-CONV): Induction hypothesis.

Case (T-MATCH): If $R_1 = x : t$ then we use the induction hypothesis on the fifth premise. Otherwise, given $e = \mathsf{match}\ v\ \mathsf{with}\ p_1 \to e_1 \ldots p_n \to e_n : t$, by Theorem 5, we are guaranteed that $\exists j$ such that $e \overset{c}{\rightsquigarrow} \sigma_j e_j$. From the premises of (T-MATCH) we have that $\Omega[\vec{x} : t'] \vdash_{app} e_j : t$. Thus, by repeated application of the substitution lemma we have $\Omega \vdash_{app} \sigma_j e_j : \sigma_j t$, to which we can apply the induction hypothesis.

Case (T-LOP): Inapplicable, since the color index $c = pol$.

Case (T-POL): For $e = (\!|e'|\!)$, we have from the first premise that $x \in FV(e')$. Thus, $\sigma(e)$ satisfies condition (iii).

□

D Correctness of the access control policy

Figure 16 reproduces the access control policy from Section 2.1. In this section we prove the correctness of this policy with respect to the non-observability condition.

Definition 16 (Similarity up to l)**.** *Expressions e and e', identified up to α-renaming, are similar up to label l according*

```
policy login(user:string, pw:string) =
  let token = match checkpw user pw with
    USER(k) =⟩ USER(k)
    _ =⟩ FAILED in
  (token, {token}0)
```

```
policy member(u:lab, a:lab) =
  match a with
    ACL(u, i) =⟩ TRUE
    ACL(j, tl) =⟩ member u tl
    _ =⟩ FALSE
```

```
policy access⟨k,α⟩(u:lab ∼ USER(k),
                   cap:int{u},
                   acl:lab, data:α{acl}) =
  match member u acl with
    TRUE =⟩ {○}data
    _ =⟩ halt#access denied
```

Figure 16. Enforcing a simple access control policy.

to the following relation:

$$e \sim_l e \qquad \{l\}e \sim_l \{l\}e' \qquad \frac{e \sim_l e'}{\{l'\}e \sim_l \{l'\}e'} \qquad \frac{e \sim_l e'}{\lambda x{:}t.e \sim_l \lambda x{:}t.e'} \qquad \frac{i = 1,2 \qquad e_i \sim_l e_i'}{e_1 e_2 \sim_l e_1' e_2'}$$

$$\frac{v \sim_l v'}{\mathsf{fix}\ f{:}t.v \sim_l \mathsf{fix}\ f{:}t.v'} \qquad \frac{e \sim_l e'}{\Lambda\alpha.e \sim_l \Lambda\alpha.e'} \qquad \frac{e \sim_l e' \qquad t \sim_l t'}{e[t] \sim_l e[t']} \qquad \frac{\forall i.e_i \sim_l e_i'}{C(\vec{e}) \sim_l C(\vec{e'})}$$

$$\frac{e \sim_l e' \qquad e_i \sim_l f_i \qquad p_i \sim_l q_i}{\mathsf{match}\ e\ \mathsf{with}\ p_i \to e_i \sim_l \mathsf{match}\ e'\ \mathsf{with}\ q_i \to f_i} \qquad \frac{e \sim_l e'}{(\!|e|\!) \sim_l (\!|e'|\!)} \qquad \frac{e \sim_l e'}{T \cdot e \sim_l T \cdot e'}$$

$$\frac{dom(()\sigma_1) = dom(()\sigma_2) \qquad \forall x.\sigma_1(x) \sim_l \sigma_2(x)}{\sigma_1 \sim_l \sigma_2}$$

The interesting case in the definition of similarity is the second judgment. Arbitrary expressions e and e' are considered similar at label l as long as they are preceded by a labeling operator that gives them the label l. The remaining rules are congruences. The full relation can be found in Appendix D. We extend the similarity relation to a bisimulation in the standard manner.

Definition 17 (Bisimulation)**.** *Expressions e_1 and e_2 are bisimilar at label l, written $e_1 \approx_l e_2$, if, and only if, $e_1 \sim_l e_2$ and for $\{i,j\} = \{1,2\}$, $e_i \overset{c}{\rightsquigarrow} e_i' \Rightarrow e_j \overset{c}{\rightsquigarrow} e_j'$ and $e_1' \approx_l e_2'$.*

Notice that this definition of bisimulation is *timing and termination sensitive*. We require both e_1 and e_2 to be reducible for an identical number of steps, with similarity up to l true of each intermediate term and of the final values, if any. We can now define our security property for the access control enforcement policy in Figure 3.

Lemma 18 (Similarity under substitution)**.** *Given substitutions $\sigma_1 \sim_l \sigma_2$, then $\sigma_1(e) \sim_l \sigma_2(e)$.*

Proof. By construction of $e \sim_l e'$ and definition of substitution. □

Theorem 19 (Non-observability)**.** *Given two label constants acl and $user$, and for*

(A1) $i = 1,2$, $\cdot \vdash_{app} v_i : t\{acl\}$

(A2) *a $(\!|\cdot|\!)$-free expression e such $a : t_a, m : t_m, cap : \mathsf{unit}\{user\}, x : t\{acl\} \vdash_{app} e : t_e$*

(A3) *Then, for type-respecting substitutions $\sigma_i = (a \mapsto access, m \mapsto member, cap \mapsto (\!|\{user\}()|\!), x \mapsto v_i)$,*

(A4) $member\ user\ acl \overset{c_*}{\rightsquigarrow} False$

Then, $\sigma_1(e) \sim_{acl} \sigma_2(e)$; and

$$\sigma_1(e) \overset{c}{\rightsquigarrow} \sigma_1(e') \iff \sigma_2(e) \overset{c}{\rightsquigarrow} \sigma_2(e') \wedge \sigma_1(e') \approx_{acl} \sigma_2(e')$$

Proof. By induction on the structure of the typing derivation (A2).

Case (T-UNIT): Trivial, since $() \sim_{acl} ()$ and $()$ is a value.

Case (T-VAR): The interesting case is when (A1) is of the form $\Gamma \vdash_{app} x : t\{acl\}$. In this case, we have $\sigma_1(x) = v_1$ and $\sigma_2(x) = v_2$. By inverting assumption (A2), we have $v_i = \{acl\}v_i'$, which, by definition, gives us $v_1 \sim_{acl} v_2$. Since v_i is irreducible, we get $v_1 \approx_{acl} v_2$. In any other case, we get $\sigma_1(x) = \sigma_2(x)$.

Case (T-FIX): We have

$$\mathsf{fix}\ f{:}\sigma_1(t).\sigma_1(v) \overset{c}{\rightsquigarrow} (f \mapsto \mathsf{fix}\ f{:}\sigma_1(t).\sigma_1(v))\sigma_1(v) = \sigma_1(f \mapsto \mathsf{fix}\ f{:}t.v)v$$

and

$$\mathsf{fix}\ f{:}\sigma_2(t).\sigma_2(v) \overset{c}{\rightsquigarrow} (f \mapsto \mathsf{fix}\ f{:}\sigma_2 u(t).\sigma_2(v))\sigma_2(v) = \sigma_2(f \mapsto \mathsf{fix}\ f{:}t.v)v$$

and since $\sigma_i(f \mapsto \mathsf{fix}\ f{:}t.v)v$ is a value, we have

$$\sigma_1(e) \sim_{acl} \sigma_2(e) \Rightarrow \sigma_1(e) \approx_{acl} \sigma_2(e)$$

Case T-TAB: We have, from (A1)

$$\sigma_1(\lambda\alpha.e) = v_1' \sim_{acl} v_2' = \sigma_2(\lambda\alpha.e)$$

, and since both are values bisimilarity is as in (T-FIX).

Case (T-TAP): We have $\Gamma \vdash_c e[t]$ and $\sigma_i(/\alpha.e[t]) \overset{c}{\rightsquigarrow} \sigma_i((\alpha \mapsto t)e)$ We have $\sigma_1((\alpha \mapsto t)e) \sim_{acl} \sigma_2((\alpha \mapsto t)e)$ from Lemma 18. But, from the type-substitution lemma (Lemma 11) we have

$$a : t_a, m : t_m, cap : \mathsf{unit}\{user\}, x : t\{acl\} \vdash_{app} (\alpha \mapsto t)e : t_e$$

Thus, for bisimilarity, we use the induction hypothesis applied to this last judgment.

Case (T-ABS): Similar to (T-TAB).

Case (T-APP): The interesting case is when $\sigma_i(e) = v_i v_i'$. In other cases, we use the induction hypothesis and (E-CTX).

Sub-case $(v_i = \lambda x : t.e)$ and e is $(\!|\cdot|\!)$-free:

We have (A2) of the form below, where $\Gamma = a : t_a, m : t_m, cap : \mathsf{unit}\{user\}, x : t\{acl\}$:

$$\frac{\frac{(A2.1.1)\Gamma, x : t \vdash_{app} e : t'}{\cdots}}{\Gamma \vdash_{app} (\lambda x{:}t.e)v'}$$

To conclude, we must show,

$$(\sigma_1, (x \mapsto \sigma_1(v')))e \approx_{acl} (\sigma_2, (x \mapsto \sigma_2(v')))e$$

But, by hypothesis, we have $\sigma_1 \sim_{acl} \sigma_2$ and by the induction hypothesis we have $\sigma_1(v') \sim_{acl} \sigma_2(v')$. Thus, we have

$$(\sigma_1, (x \mapsto \sigma_1(v')))e \sim_{acl} (\sigma_2, (x \mapsto \sigma_2(v')))e$$

immediately, from Lemma 18. To conclude with bisimilarity, we use

$$(\sigma_i, (x \mapsto \sigma_i(v')))e = \sigma_i((x \mapsto v')e)$$

and we use (A2.1.1) and the substitution lemma, Lemma 8 to establish

$$a : t_a, m : t_m, cap : \mathsf{unit}\{user\}, x : t\{acl\} \vdash_{app} (x \mapsto v')e : (x \mapsto v')t_e$$

Finally, we apply the induction hypothesis to this last judgment to get bisimilarity.

Sub-case $(v_i = \lambda x : t.(\!|e|\!))$ and e is $(\!|\cdot|\!)$-free:

In this case, we are unable to use the induction hypothesis to establish bisimilarity since,

$$\frac{\frac{(A2.1.1)\Gamma, x : t \vdash_{pol} e : t'}{\cdots}}{\Gamma \vdash_{app} (\lambda x{:}t.(\!|e|\!))v'}$$

and the hypothesis only applies to *app*-context judgments. However, our assumption was that e is $(\!|\,|\!)$-free. Thus, since we are assuming that $\sigma_i(e)$ is of the form $\lambda x{:}t.(\!|e'|\!)v'$, $(\!|e'|\!)$ must result from an application of $apply : t_a$, where $(a \mapsto apply) \in \sigma_i$. We proceed by cases on the syntactic form of $\lambda x{:}t.(\!|e'|\!)v'$

Sub-case $(\lambda u : \mathsf{lab} \sim USER(k).(\!|\lambda cap{:}t.e|\!))$: After a step of reduction, we get $(u \mapsto v_i')(\!|\lambda cap{:}t.e|\!)$ and since *apply* is a closed term $\sigma_i(\!|\lambda cap{:}t.e|\!) = (\!|\lambda cap{:}t.e|\!)$. Thus, by Lemma 18, we have

$$(u \mapsto v_1')(\!|\lambda cap{:}t.e|\!) \sim_{acl} (u \mapsto v_2')(\!|\lambda cap{:}t.e|\!)$$

To establish bisimilarity, each side reduces in one step to a values using (E-BABS)

$$\lambda cap{:}t.(\!|(u \mapsto v_1')e|\!) \sim_{acl} \lambda cap{:}t.(\!|(u \mapsto v_2')e|\!)$$

Sub-case $(\lambda cap{:}unit u.(\!|e|\!))$: We must have that $u = user$ since, we have an application

$$e = (\lambda x : unit\{u\}.(\!|e|\!)\sigma_i(v')$$

, and, by assumption v' is $(\!|\,|\!)$-free. Thus, we have an application in which $e_i = \lambda x : unit\{user\}.(\!|e|\!)(\!|\{user\}()|\!)$ reduces, as in the previous subcase, to similar values.

Sub-case $(\lambda acl : lab.(\!|e|\!))$: Similar to previous sub-case.

Sub-case $(\lambda x : tl.(\!|e|\!))$: We must have $l = acl$, since we have an application

$$e = (\lambda x : t\{l\}.\sigma_i((\!|e|\!))\sigma_i(v')$$

, and, by assumption v' is $(\!|\,|\!)$-free. Thus, we have an application in which $e_i = \lambda x : t\{acl\}.(\!|e|\!)v_i$ which reduces in one step to

```
match member user acl with
TRUE → {∘}data
_ → halt#access denied
```

But, by assumption we have $member\ user\ acl \overset{c_*}{\rightsquigarrow} FALSE$. Thus, in both cases the program halts, maintaining bisimilarity.

Case (T-LAB): If we have $\Gamma \vdash_{app} C(\vec{u}, e, \vec{e}) : \mathsf{lab}\ C(\vec{e})$, then we can reduce the ith subterm using (E-CTX) and establish the result using the induction hypothesis. Otherwise, we have $\sigma_1(e) = C(\vec{u}) = \sigma_2(e)$, because, if $x \in FV(e) \Rightarrow v_i$ is a subterm of $\sigma(e)$ which is lab-typed. This is impossible since by (A1) v_i has a labeled type and unlabelings are not permitted in *app*-context.

Case (T-HIDE), (T-SHOW): Induction hypothesis.

Case (T-MATCH): The interesting case is when the matched expression is in fact a value v; in all other cases we conclude using (E-CTX) and the induction hypothesis. However, by the first premise, we have $\Gamma \vdash \sigma_i(v) : \mathsf{lab}$ and $\sigma_1(v) \sim_{acl} \sigma_2(v)$, which at type lab implies $\sigma_1(v) \equiv \sigma_2(v)$. Thus, both $\sigma_1(e)$ and $\sigma_2(e)$ reduce to the same pattern branch and we conclude with the induction hypothesis.

Case (T-UNLAB), (T-RELAB): Inapplicable, since by assumption (A2) is in *app*-context.

Case (T-POL): Inapplicable, since by assumption e is $(\!|\,|\!)$-free.

Case (T-CONV): Induction hypothesis. □

E Dynamic Provenance Tracking

Figure 17 reproduces the provenance policy from Section 3.2. Figure E defines a logical relation parameterized by a label l in order to relate terms with similar provenance.

Lemma 20 (Substitution for type-shape relation)**.** *Given a well-formed* Γ *such that*

(A1) $\Gamma \vdash t :: \kappa$

(A2) $\sigma_1 \approx_p \sigma_2 : \Gamma, \Gamma$

Then, $\sigma_1 t \approx \sigma_2 t :: \kappa$

```
typename Prov α= (l:lab{Auditors} × α{{∘}l})
policy apply⟨α,β⟩ (lf:Prov (α → β), mx:Prov α) =
  let l,f = lf in
  let m,x = mx in
  let y = ({∘}f) ({∘}x) in
  let lm = Union({∘}l, {∘}m) in
    ({Auditors}lm, {lm}y)
```

```
policy flatten⟨α⟩ (x:Prov (Prov α)) =
    let l,inner = x in
    let m,a = inner in
    let lm = Union({∘}l, {∘}m) in
      ({Auditors}lm, {lm}a)
```

Figure 17. Enforcing a dynamic provenance tracking policy.

Proof. Straightforward induction on the structure of (A1)—a substitution of free variables in t does not change the shape of t. □

Lemma 21 (Substitution for logical relation)**.** *Given a well-formed* Γ *such that*

(A1) $\Gamma \vdash_{app} e : t$

(A2) e *is* $(\![\cdot]\!)$*-free*

(A3) $\sigma_1 \approx_p \sigma_2 : \Gamma, \Gamma$

(A4) $e_1 \succ e_2 \notin \Gamma$

Then, $\sigma_1 e \approx_p \sigma_2 e : \sigma_1 t, \sigma_2 t$

Proof. By induction on the structure of (A1).

Case (T-INT): Trivial, with (R-INT) and $\sigma_i(t) = t$.

Case (T-VAR): By assumption (A3) $\sigma_1 \approx_p \sigma_2 : \Gamma, \Gamma$ by (R-SUBST) with $\sigma_1(x) \approx_p \sigma_2(x) : \Gamma(x), \Gamma(x)$ in the premise, where $\Gamma(x) = t = \sigma_i(t)$.

Case (T-FIX): Fix can be handled using a standard labeled reduction as in Mitchell [29], section 8.3.4. Note that according to (R-EXPR), we are only concerned with terminating computations.

Case (T-TAB):

$$\frac{\Gamma, \alpha :: \kappa \vdash e : t}{\Gamma \vdash_{app} \Lambda \alpha :: \kappa . e : \forall \alpha :: \kappa . t} \text{ (T-TAB)}$$

We use the induction hypothesis on the premise to show that

$$\sigma_1' e \approx_p \sigma_2' e : \sigma_1' t, \sigma_2' t$$

where $\sigma_i' = \sigma_i, \alpha \mapsto t_i'$ and $t_1' \approx t_2' :: \kappa$. Thus, by (A3) we have $\sigma_1' \approx_p \sigma_2' : (\Gamma, \alpha :: \kappa), (\Gamma, \alpha :: \kappa)$. This suffices to establish the conclusion via (R-UNIV).

Case (T-TAP):

$$\frac{\Gamma \vdash t :: \kappa \qquad \Gamma \vdash_{app} e : \forall \alpha :: \kappa . t'}{\Gamma \vdash_{app} e\,[t] : t''} \text{ (T-TAP)}$$

From the second premise we can use the induction hypothesis to establish that

$$\sigma_1(e) \approx_p \sigma_2(e) : \sigma_1(\forall \alpha :: \kappa . t), \sigma_2(\forall \alpha :: \kappa . t)$$

via (R-UNIV). I.e. $\forall t_1, t_2 . t_1 \approx t_2 :: \kappa \;\Rightarrow\; \sigma_1(e)[t_1] \approx_p \sigma_2(e)[t_2] : (\sigma_1, \alpha \mapsto t_1)t', (\sigma_2, \alpha \mapsto t_2)t'$.
But, from the first premise (T-TAP) and from Lemma 20 we have that $\sigma_1(t) \approx \sigma_2(t) :: \kappa$. Thus, we can conclude

$$\sigma_1(e)[\sigma_1 t] \approx_p \sigma_2(e)[\sigma_2 t] : \sigma_1 t'', \sigma_2 t''$$

as required.

Case (T-ABS):

$$\frac{\Gamma \vdash t_1 :: \kappa \qquad \Gamma, x : t_1 \vdash_c e : t_2}{\Gamma \vdash_{app} \lambda x : t_1 . e : x : t_1 \to t_2} \text{ (T-ABS)}$$

$[\![e]\!]$ Interpretation of labels as sets

$$[\![C]\!] \stackrel{\text{def}}{=} \{C\} \qquad [\![\mathit{Union}(l_1, l_2)]\!] \stackrel{\text{def}}{=} [\![l_1]\!] \cup [\![l_2]\!]$$

$t \leq t'$ Prefixing relation on types

$$t \leq t \qquad \frac{t \leq t'}{t \leq t'\{e\}}$$

$e \approx_p e' : t, t'$ e and e' are related at types t and t', parameterized by a provenance color p

$$\frac{\begin{array}{c} i \in \{1,2\} \quad \cdot \vdash_c v_i : t_i \quad t_i'\{e_i\} \leq t_i \quad t \leq t_i \quad e_i \overset{pol_*}{\rightsquigarrow} v_i^{lab} \\ c \in [\![v_1^{lab}]\!] \cap [\![v_2^{lab}]\!] \quad \vee \quad \mathit{Auditors} \in [\![v_1^{lab}]\!] \cap [\![v_2^{lab}]\!] \end{array}}{v_1 \approx_p v_2 : t_1, t_2} \text{ (R-EQUIVC)}$$

$$n \approx_p n : \mathsf{int}, \mathsf{int} \text{ (R-INT)} \qquad \frac{\begin{array}{c} i \in \{1,2\} \qquad \cdot \vdash_c e_i : t_i \\ e_i \overset{c_*}{\rightsquigarrow} v_i \quad \Rightarrow \quad v_1 \approx_p v_2 : t_1, t_2 \end{array}}{e_1 \approx_p e_2 : t_1, t_2} \text{ (R-EXPR)}$$

$$\frac{\begin{array}{c} \cdot \vdash_c v : (x{:}t_1) \to t_2 \qquad \cdot \vdash_c v' : (x{:}t_1') \to t_2' \\ \forall v_1, v_1'.\ v_1 \approx_p v_1' : t_1, t_1' \quad \Rightarrow \quad v v_1 \approx_p v' v_1' : (x \mapsto v_1)t_2, (x \mapsto v_1')t_2' \end{array}}{v \approx_p v' : (x{:}t_1) \to t_2, (x{:}t_1') \to t_2'} \text{ (R-ABS)}$$

$$\frac{\begin{array}{c} i \in \{1,2\} \qquad \cdot \vdash_c v_i : \forall \alpha{::}\kappa.t_i \\ \forall t_1', t_2'.\ t_1' \approx t_2' :: \kappa \quad \Rightarrow \quad v_1[t_1'] \approx_p v_2[t_2'] : (\alpha \mapsto t_1')t_1, (\alpha \mapsto t_2')t_2 \end{array}}{v_1 \approx_p v_2 : \forall \alpha{::}\kappa.t_1, \forall \alpha{::}\kappa.t_2} \text{ (R-UNIV)}$$

$$\frac{i \in \{1,2\} \quad \cdot \vdash_c C(\vec{e^i}) : \mathsf{lab} \quad \forall j. e_j^1 \approx_p e_j^2 : \mathsf{lab}, \mathsf{lab}}{C(\vec{e^1}) \approx_p C(\vec{e^2}) : \mathsf{lab}, \mathsf{lab}} \text{ (R-LAB)} \qquad \frac{e_1 \approx_p e_2 : \mathsf{lab}, \mathsf{lab}}{e_1 \approx_p e_2 : \mathsf{lab} \sim e_1, \mathsf{lab} \sim e_2} \text{ (R-LAB2)}$$

$$\frac{e_1 \approx_p e_2 : t_1, t_2 \quad i \in \{1,2\} \quad \cdot \vdash t_i \cong t_i'}{e_1 \approx_p e_2 : t_1', t_2'} \text{ (R-CONV)} \qquad \frac{i \in \{1,2\} \quad \cdot \vdash_c (\!|v_i|\!) : t_i \quad v_1 \approx_p v_2 : t_1, t_2}{(\!|v_1|\!) \approx_p (\!|v_2|\!) : t_1, t_2} \text{ (R-BRAC)}$$

$$\frac{v \approx_p v' : t, t'}{\{e\}v \approx_p \{e'\}v' : t\{e\}, t'\{e'\}} \text{ (R-RELAB)} \qquad \frac{\begin{array}{c} \mathrm{dom}(\sigma) = \mathrm{dom}(\sigma') \qquad \forall \alpha \in \mathrm{dom}(\sigma).\Gamma(\alpha) = \Gamma'(\alpha) \\ \sigma(\alpha) \approx \sigma'(\alpha) : \Gamma(\alpha) \quad \forall x \in \mathrm{dom}(\sigma).\sigma(x) \approx_p \sigma'(x) : \Gamma(x), \Gamma'(x) \end{array}}{\sigma \approx_p \sigma' : \Gamma, \Gamma'} \text{ (R-SUBST)}$$

$t \approx t' :: \kappa$ Types t and t' are related (have the same shape) at kind κ

$$\frac{\cdot \vdash t :: \kappa}{t \approx t :: \kappa} \text{ (RT-ID)} \qquad \frac{i \in \{1,2\} \quad \cdot \vdash \mathsf{lab} \sim e_i :: \kappa}{\mathsf{lab} \sim e_1 \approx \mathsf{lab} \sim e_2 :: \kappa} \text{ (RT-LAB)}$$

$$\frac{i \in \{1,2\} \quad \cdot \vdash t_i\{e_i\} :: \mathrm{M} \quad t_1 \approx t_2 :: \kappa}{t_1\{e_1\} \approx t_2\{e_2\} :: \mathrm{M}} \text{ (RT-LABELED)}$$

$$\frac{\begin{array}{c} i \in \{1,2\} \qquad \cdot \vdash (x{:}t_i) \to t_i' :: \kappa \qquad t_1 \approx t_2 :: \kappa' \\ \forall v_1, v_2 . \cdot \vdash_c v_i : t_i \quad \Rightarrow \quad (x \mapsto v_1)t_1' \approx (x \mapsto v_2)t_2' :: \kappa' \end{array}}{(x{:}t_1) \to t_1' \approx (x{:}t_2) \to t_2' :: \kappa} \text{ (RT-FUN)}$$

$$\frac{\begin{array}{c} i \in \{1,2\} \qquad \cdot \vdash \forall \alpha{::}\kappa.t_i : \kappa' \\ \forall t, t' . t \approx t' :: \kappa \quad \Rightarrow \quad (\alpha \mapsto t)t_1 \approx (\alpha \mapsto t')t_2 :: \kappa' \end{array}}{\forall \alpha{::}\kappa.t_1 \approx \forall \alpha{::}\kappa.t_2 :: \kappa'} \text{ (RT-UNIV)}$$

Figure 18. A logical relation for dynamic provenance tracking. (Limited to FABLECORE)

Our goal is to establish $\sigma_1(\lambda x{:}t_1.e) \approx_p \sigma_2(\lambda x{:}t_1.e) : \sigma_1((x{:}t_1) \to t_2), \sigma_2((x{:}t_1) \to t_2)$ via (R-ABS).
To use (R-ABS), we must first show for $i \in \{1,2\}$,

(T1) $\cdot \vdash_{app} \sigma_i(\lambda x{:}t_1.e) : \sigma_i((x{:}t_1) \to t_2)$

But this follows directly from the substitution lemma, Lemma 8.
Next, we must show

$$(R3) \forall v_1, v_2. v_1 \approx_p v_2 : \sigma_1 t_1, \sigma_2 t_1 \;\Rightarrow\; \sigma_1(\lambda x{:}t_1.e)v_1 \approx_p \sigma_2(\lambda x{:}t_1.e)v_2 : (\sigma_1, x \mapsto v_1)t_2, (\sigma_2, x \mapsto v_2)t_2$$

which, by (R-EXPR) requires showing

$$(R4)(\sigma_1(\lambda x{:}t_1.e)v_1 \overset{c_*}{\rightsquigarrow} v_1' \wedge \sigma_2(\lambda x{:}t_1.e)v_2 \overset{c_*}{\rightsquigarrow} v_2') \;\Rightarrow\; v_1' \approx_p v_2' : (\sigma_1, x \mapsto v_1)t_2, (\sigma_2, x \mapsto v_2)t_2$$

By (E-APP), this is equivalent to showing

$$(R4')(\sigma_1'(e) \overset{c}{\rightsquigarrow}{}^* v_1' \wedge \sigma_2'(e) \overset{c}{\rightsquigarrow}{}^* v_2') \;\Rightarrow\; v_1' \approx_p v_2' : (\sigma_1, x \mapsto v_1)t_2, (\sigma_2, x \mapsto v_2)t_2$$

where $\sigma_i' = \sigma_i, x \mapsto v_i$ and $v_1 \approx_p v_2 : \sigma_1 t_1, \sigma_2 t_1$. This follows from the induction hypothesis applied to the first premise of (A1).

Case (T-APP):

$$\frac{\Gamma \vdash_c e_1 : (x{:}t_1) \to t_2 \qquad \Gamma \vdash_c e_2 : t_1}{\Gamma \vdash_c e_1\, e_2 : t_2'} \text{ (T-APP)}$$

Our goal is to show, via (R-EXPR),

$$\sigma_1(e_1 e_2) \approx_p \sigma_2(e_1 e_2) : \sigma_1 t_2', \sigma_2 t_2'$$

By the induction hypothesis applied to the premises of (A1) we have

(R1) $\sigma_1(e_1) \approx_p \sigma_2(e_1) : \sigma_1((x{:}t_1) \to t_2), \sigma_2((x{:}t_1) \to t_2)$

(R2) $\sigma_1(e_2) \approx_p \sigma_2(e_2) : \sigma_1 t_1, \sigma_2 t_1$

If e_1 is not a value, then by inversion (R1) is an instance of (R-EXPR) and we have

$$\sigma_1 e_1 \overset{c_*}{\rightsquigarrow} v_1 \;\wedge\; \sigma_2 e_1 \overset{c_*}{\rightsquigarrow} v_2 \;\Rightarrow\; v_1 \approx_p v_2 : \sigma_1((x{:}t_1) \to t_2), \sigma_2((x{:}t_1) \to t_2)$$

If e_1 is a value we have the conclusion of the previous implication directly. Inverting this relation, an instance of (R-ABS), we can derive

(R1.1) $\forall v, v', v \approx_p v' : \sigma_1 t_1, \sigma_2 t_1 \;\Rightarrow\; (\sigma_1(e_1)v \overset{c}{\rightsquigarrow}{}^* v_1 \wedge \sigma_2(e_1)v' \overset{c}{\rightsquigarrow}{}^* v_2) \Rightarrow v_1 \approx_p v_2 : (\sigma_1, x \mapsto v)t_2, (\sigma_2, x \mapsto v')t_2$

Similarly, inverting (R2) we can conclude

(R2.1) $\sigma_1(e_2) \overset{c_*}{\rightsquigarrow} v_1 \wedge \sigma_2(e_2) \overset{c_*}{\rightsquigarrow} v_2 \Rightarrow v_1 \approx_p v_2 : \sigma_1 t_1, \sigma_2 t_1$

If either, $\sigma_i e_1$ or $\sigma_i e_2$ diverge then we can establish the result trivially using (R-EXPR) since the guard in the implication of the last premise is false.

If neither diverges, we use (R1.1) instantiating v and v' to v_1 and v_2, respectively, from (R2.1).

Case (T-LAB), (T-HIDE), (T-SHOW): Induction hypothesis on the premise and concluding with either (R-LAB2) or reusing the premise of (R-LAB2) to conclude with (R-LAB).

Case (T-MATCH):

$$\frac{\begin{array}{ccc}\Gamma \vdash_c e : \mathsf{lab} & \Gamma \vdash t :: \kappa & p_n = x \quad \text{where } x \notin \mathrm{dom}(\Gamma) \\ \vec{x_i} = FV(p_i) \setminus \mathrm{dom}(\Gamma) & \Gamma, \vec{x_i} : \mathsf{lab} \vdash_c p_i : \mathsf{lab} & \Gamma, \vec{x_i} : \mathsf{lab}, e \succ p_i \vdash_c e_i : t\end{array}}{\Gamma \vdash_c \mathsf{match}\ e\ \mathsf{with}\ p_1 \Rightarrow e_1 \ldots p_n \Rightarrow e_n : t} \text{ (T-MATCH)}$$

Applying the IH to the first premise we get that

- (R1) $\sigma_1(e) \approx_p \sigma_2(e) : \mathsf{lab}, \mathsf{lab}$

However, by inverting (R1), we find that it must be an instance of (R-EXPR) or (R-LAB). We are only concerned with the case in which both $\sigma_1(e)$ and $\sigma_2(e)$ reduce to a values v_1 and v_2 respectively, since otherwise $\sigma_1 \mathsf{match}... \approx_p \sigma_2 \mathsf{match}... : t$ vacuously, using (R-EXPR) and the substitution lemma. In this case (where they both converge), we can consider the last premise of (R-EXPR) to be an instance of (R-LAB). However, $v_1 \approx_p v_2 : \mathsf{lab}$, via (R-LAB) only if $v_1 = v_2 = v$.

Thus, if $\mathsf{match} v_1 ... \overset{c}{\rightsquigarrow} \sigma^* \sigma_1(e_j)$, iff $\mathsf{match} v_2 ... \overset{c}{\rightsquigarrow} \sigma^* \sigma_2(e_j)$.

But, from (A1) we have

$$\Gamma, \vec{x_i} : \mathsf{lab}, v \succ p_j \vdash_c e_j : t$$

By Corollary 10 we have $\Gamma, \vec{x_i} : \mathsf{lab} \vdash_c e_j : t$. Furthermore, since $v_1 = v_2$, we have

$$\sigma^* \approx_p \sigma^* : \vec{x_i} : \mathsf{lab}, \vec{x_i} : \mathsf{lab}$$

using (R-LAB) repeatedly, as necessary. Therefore,

$$\sigma_1, \sigma^* \approx_p \sigma_2, \sigma^* : \Gamma, \vec{x_i} : \mathsf{lab}, \Gamma, \vec{x_i} : \mathsf{lab}$$

.

Finally, using the induction hypothesis on e_j, σ_1, σ^*, σ_2, σ^* we have our conclusion.

Case (T-UNLAB), (T-RELAB): Impossible; the statement only applies to $\Gamma \vdash_{app} e : t$.

Case (T-POL): Impossible; the statement only applies to $(\!|\cdot|\!)$-free terms.

Case (T-CONV):

$$\frac{\Gamma \vdash_c e : t \qquad \Gamma \vdash t \cong t'}{\Gamma \vdash_c e : t'} \text{ (T-CONV)}$$

By the induction hypothesis applied to

$$\Gamma \vdash_{app} e : t$$

we have

$$\sigma_1(e) \approx_p \sigma_2(e) : \sigma_1 t, \sigma_2 t$$

By the second premise of (A1), we have that $t \cong t'$. But, by definition of (TE-REDUCE), $t \cong t'$ iff $\forall \sigma. \sigma(t) \overset{c}{\rightsquigarrow}^* \sigma(t')$. Thus, we establish the conclusion using (R-CONV).

□

Note that this relation permits terms at different types to be related. For instance, suppose we have

$$l : \mathsf{lab}\{Auditors\} \vdash \Lambda\alpha.\lambda x{:}\alpha.x\ [\mathsf{int}\{\{\circ\}l\}] : (x{:}\mathsf{int}\{\{\circ\}l\} \to \mathsf{int}\{\{\circ\}l\})$$

And $l1 = \{Auditors\}Union(P, Red)$ and $l2 = \{Auditors\}Red$. Clearly, we have $l1 \approx_p l2 : \mathsf{lab}\{Auditors\}, \mathsf{lab}\{Auditors\}$.

Lemma 21 states that after substitution,

$$\lambda x{:}\mathsf{int}\{Union(P,Red)\}.x \approx_p \lambda x{:}int\{Red\}.x : (x{:}\mathsf{int}\{Union(P,Red)\}) \to \mathsf{int}\{Union(P,Red)\}, (x{:}\mathsf{int}\{Red\} \to \mathsf{int}\{Red\}$$

which requires showing that for

$$v_1 \approx_p v_2 : \mathsf{int}\{Union(P,Red)\}, \mathsf{int}\{Red\} \Rightarrow \lambda x : \mathsf{int}\{Union(P,Red)\}.x\ v_1 \approx_p \lambda x{:}int\{Red\}.x\ v_2 : \ldots$$

This result is justified easily in two ways:

1. The guard is impossible to satisfy, since, by inversion, the only potentially applicable rule is (R-EQUIVC) and the disjunction in the last premise cannot be satisfied.

2. Assuming the guard, we can show the result using (R-EXPR) and (E-APP) since each side reduces to v_1, v_2 resp. and the resulting implication is a tautology.

We can now use Lemma 21 to state and prove the main result of this section.

Theorem 22 (Dependency correctness)**.** *Given all of the following:*

(A1) *a $(\!|\cdot|\!)$-free expression e such that $a : t_a, f : t_f, x : \mathit{Prov}\ t \vdash_{\mathit{app}} e : t'$,*

(A2) *a type-respecting substitution $\sigma = (a \mapsto \mathit{apply}, f \mapsto \mathit{flatten})$.*

(A3) *$\vdash_{\mathit{app}} v_i : \mathit{Prov}\ t$ for $i = 1,2$ and $v_1 \approx_p v_2 : \mathit{Prov}\ t$*

(A4) *for $i \in \{1,2\}$, $\sigma_i = \sigma, x \mapsto v_i$*

Then, $(\sigma_1(e) \overset{\mathit{app}_*}{\rightsquigarrow} v_1' \ \wedge \ \sigma_2(e) \overset{\mathit{app}_*}{\rightsquigarrow} v_2') \Rightarrow v_1' \approx_p v_2' : \sigma_1 t', \sigma_2 t'$.

Proof. It suffices to show that $\mathit{apply} \approx_p \mathit{apply} : t_a$ and $\mathit{flatten} \approx_p \mathit{flatten} : t_f$. Then using (A3) to establish $\sigma_1 \approx_p \sigma_2 : a : t_a, f : t_f, x : \mathit{Prov}\ t$ and can use Lemma 21 for the conclusion.

Case (apply): The interesting case of $\mathit{apply} \approx_p \mathit{apply} : t_a$ amounts to showing

$$\mathit{apply}[t_1,t_2]v_1 \approx_p \mathit{apply}[t_1,t_2]v_2 : t_a$$

where (A5) $v_1 \approx_p v_2 : \mathit{Prov}\ t_1 \to t_2, \mathit{Prov}\ t_1 \to t_2$. We write (A5) using the more convenient notation of dependent tuples.

$$\text{(A5)} \qquad \frac{(\!|\{\mathit{Auditors}\}l|\!) \approx_p (\!|\{\mathit{Auditors}\}l'|\!) : \mathsf{lab}\{\mathit{Auditors}\}, \mathsf{lab}\{\mathit{Auditors}\} \qquad (\!|\{l\}f|\!) \approx_p (\!|\{l'\}g|\!) : (t_1 \to t_2)\{l\}, (t_1 \to t_2)\{l'\}}{((\!|\{\mathit{Auditors}\}l|\!), (\!|\{l\}f|\!)) \approx_p ((\!|\{\mathit{Auditors}\}l'|\!), (\!|\{l'\}g|\!)) : \mathit{Prov}\ (t_1 \to t_2), \mathit{Prov}\ (t_1 \to t_2)}$$

The first premise is an instance of (R-BRAC) followed by (R-EQUIVC) since we have a labeling with $\mathit{Auditors}$. We now proceed by cases on the structure of the second premise of (A5), (A5.2).

Inverting (A5.2), we find that it must be an instance of (R-EQUIVC) or (R-BRAC).

Case (R-EQUIVC): In this case, we have from the last premise, that $c \in [\![l_1]\!] \cap [\![l_2]\!]$. Since $\mathit{apply}[t_1,t_2]v_1$ and $\mathit{apply}[t_1,t_2]v_2$ are values with type $t_1 \to \mathit{Prov}\ t_2$, we must establish the relation using (R-ABS). However, from inspection of apply, we find that whenever $\mathit{apply}[t_1,t_2]v_i x$ terminates, it does so with a value of the form $(l_i, v_i^*) : \mathit{Prov}\ t_2$, i.e. $v_i^* : t_2\{l_i\}$. Since we have already established t hat $c \in [\![l_1]\!] \cap [\![l_2]\!]$, we can establish that $(l_1, v_1^*) \approx_p (l_2, v_2^*) : \mathit{Prov}\ t_2, \mathit{Prov}\ t_2$ using the same form as (A5).

Case (R-BRAC):

$$\frac{\cdot \vdash_{\mathit{app}} (\!|\{l\}f|\!) : t \qquad \cdot \vdash_{\mathit{app}} (\!|\{l'\}g|\!) : t' \qquad \{l\}f \approx_p \{l'\}g : t, t'}{(\!|\{l\}f|\!) \approx_p (\!|\{l'\}g|\!) : t, t'} \text{ (R-BRAC)}$$

Inverting the third premise, we find that it must be an instance of (R-RELAB), with (R-ABS) in the premise. That is, we have that

$$(A6) \qquad \forall v_1', v_2'. v_1' \approx_p v_2' : t_1, t_1 \Rightarrow f v_1' \approx_p g v_2' : (x \mapsto v_1')t_2, (x \mapsto v_2')t_2$$

To establish the conclusion

$$\mathit{apply}[t_1,t_2]v_1 v_1' \approx_p \mathit{apply}[t_1,t_2]v_2 v_2' : t_2', t_2''$$

we notice that if both expressions do not diverge, the left side reduces to $(l_1, (\!|\{\mathit{Union}(l_1, l_1')\} f v_1'|\!))$ and the right side to $(l_2, (\!|\{\mathit{Union}(l_2, l_2')\} g v_2'|\!))$. To establish the goal, we use reuse (A5) with (R-BRAC), (R-RELAB) and the conclusion of (A6) for the second premise.

Case ($\mathit{flatten}$): Following an argument similar to apply and concluding with (R-EQUIVC), since $\mathit{Auditor}$-labeled terms are always considered equivalent in the relation. □

```
policy lub (x:lab, y:lab) = match x,y with
        HIGH, _ → HIGH
        _, HIGH → HIGH
        _, MED → MED
        MED, _ → MED
        _ → LOW

policy low⟨α⟩(x:α) = {LOW}_1 x

policy join⟨α,l,m⟩ (x:α{l}{m}) = {lub l m}_2 x

policy sub⟨α,l⟩ (x:α{l}, m:lab) = {lub l m}_3 x

policy def⟨α⟩ (l:lab, x:α) = sub [α] (low [α] x) l
```

```
policy app⟨α::U, β,l,m⟩ (f:(α → β){l}, x:α{m}) =
    {lub l m}_4 (({∘}_5 f) ({∘}_6 x))

policy app2⟨α,β,l⟩ (f:(α → β){l}, x:α) =
            {l}_7 (({∘}_8 f) x)

typename bool l = ∀α.(α → α → α){l}
let client⟨α⟩ (b: bool HIGH, x:α{LOW}, y:α{MED}) =
  let b' = b [α] in
  let x' = sub [α] x MED in
  let tmp = app [α] [α → α] b' x' in
    app [α] [α] tmp y
```

Figure 19. Enforcing a static information flow policy.

Additional syntactic forms

$$
\begin{array}{lcll}
e & ::= & \ldots \mid \{\!\{e_1 \parallel e_2\}\!\} & \text{Bracketed expressions represent multiple executions} \\
v_c & ::= & \ldots \mid \{\!\{v_c \parallel v_c\}\!\} \mid \{\circ\}\{\!\{v_c \parallel v_c\}\!\} & \text{Extensions to values} \\
E_c & ::= & \ldots \mid \{\!\{\bullet \parallel e\}\!\} \mid \{\!\{e \parallel \bullet\}\!\} & \text{Evaluation contexts}
\end{array}
$$

$\Omega \vdash_c e : t$ — Additional type rules

$$\frac{t = t'\{\vec{e_l}\} \quad \Omega \vdash t' :: \mathrm{U} \quad lub\ \vec{e_l} \overset{c_*}{\rightsquigarrow} HIGH \quad \Omega \vdash_c e_1 : t \quad \Omega \vdash_c e_2 : t \quad \forall i. e_i \neq \{\!\{e_i' \parallel e_i''\}\!\}}{\Omega \vdash_c \{\!\{e_1 \parallel e_2\}\!\} : t} \text{ (T-BRACKET)}$$

$e \overset{c}{\rightsquigarrow} e'$ — Additional reduction rules

$$\frac{i \in \{1,2\}}{\lfloor \{\!\{v_1 \parallel v_2\}\!\} \rfloor_i \equiv v_i} \text{ (PROJ-1)} \qquad \frac{i \in \{1,2\}}{\lfloor \langle t \rangle \{\!\{v_1 \parallel v_2\}\!\} \rfloor_i \equiv \langle t \rangle v_i} \text{ (PROJ-2)} \qquad \frac{i \in \{1,2\} \quad v \notin \{\{\!\{v_1 \parallel v_2\}\!\}, \langle t \rangle \{\!\{v_1 \parallel v_2\}\!\}\}}{\lfloor v \rfloor_i \equiv v} \text{ (PROJ-3)}$$

$$\frac{v_2 \neq \{\circ\}_6 \{\!\{v_2' \parallel v_2''\}\!\}}{(\lambda x : t.e) v_2 \overset{c}{\rightsquigarrow} (x \mapsto v_2) e} \text{ (E-APP)}$$

$$\{e\}_{4,7}(\{\circ\}_{5,8}\{\!\{v_f \parallel v_f'\}\!\})v_x \overset{c}{\rightsquigarrow} \{\!\{\{e\}_{4,7}(\{\circ\}_{5,8} v_f)\lfloor v_x \rfloor_1 \parallel \{e\}_{4,7}(\{\circ\}_{5,8} v_f')\lfloor v_x \rfloor_2\}\!\} \text{ (E-BAPP1)}$$

$$\frac{v_f \neq \langle t' \rangle \{\!\{v_1 \parallel v_2\}\!\}}{\{e\}_4 (v_f (\{\circ\}_6 \{\!\{v_x \parallel v_x'\}\!\})) \overset{c}{\rightsquigarrow} \{\!\{\{e\}_4 v_f (\{\circ\}_6 v_x) \parallel \{e\}_4 v_f (\{\circ\}_6 v_x')\}\!\}} \text{ (E-BAPP2)}$$

Figure 20. Semantics of FABLECORE^2.

F Correctness of the static information-flow policy

Figure 19 reproduces the policy of Figure 7, expanding it to a three-point lattice and showing an example program *client*. However, to assist with the proof, we have annotated each relabeling operator with a unique index corresponding to its location in the source program. These annotations are similar to the allocation site indices commonly used in pointer analyses.

Figure F gives the semantics of FABLECORE^2, an extension of FABLECORE in the spirit of Core-ML^2 [33], Pottier and Simonet's technique for representing multiple program execution within the syntax of a single program.

Our first lemma, Progress for FABLECORE^2, is necessary to establish that our augmented operational semantics are sufficiently expressive to capture the evaluation of a pair of FABLECORE programs. It relies on the indices given to each relabeling operation.

Lemma 23 (Progress for FABLECORE^2)**.** *Given well-formed $\Gamma = \vec{x} : \vec{t}$, where $\vec{t} = t_{lub}, \ldots, t_{app2}$, the types of the policy π, of Figure 19, and a $\{\!|\}$-free FABLECORE^2 program e such that $\Gamma; \cdot; \cdot \vdash_c e : t$; then $\sigma(e) \overset{c}{\rightsquigarrow} e'$, or $\sigma(e)$ is a value, where $\sigma = (\vec{x} \mapsto \vec{t})$*

Proof. By induction on the structure of $\Omega \vdash \pi; e : t$, relying on Theorem 5 for most cases, where $\Omega = \Gamma; \cdot; \cdot$. (limited

to the functional core)

Case (T-UNIT, T-VAR, T-SUBR, T-L1, ..., T-L6): Trivial, by using Theorem 5 and appealing to evaluation context $\{\!\{\bullet \parallel e\}\!\}$ or $\{\!\{e \parallel \bullet\}\!\}$.

Case (T-UNLAB): If $e \neq \{\!\{v_1 \parallel v_2\}\!\}$ then we follow Theorem 5, appealing to evaluation context $\{\!\{\bullet \parallel e\}\!\}$ or $\{\!\{e \parallel \bullet\}\!\}$, if necessary. Otherwise, if $e = \{\!\{v_1 \parallel v_2\}\!\}$, then, by definition $\{\circ\}e$ is a value.

Case (T-APP): If e_1 and e_2 are not bracketed values, then we follow Theorem 5. If e_1 is bracketed, then e_1 must be $\{\circ\}\{\!\{v_1 \parallel v_2\}\!\}$, since, by (T-BRACKET) $\Omega \vdash_c \{\!\{v_1 \parallel v_2\}\!\} : t'e$, and from the first premise of (T-APP) we must have e_1 of unlabeled type, and so t must be an unlabeled type.

We now proceed by cases on the possible unlabeling indices $i \in \{5,6,8\}$. In the first case, we have an evaluation context

$$E \cdot (\{\circ\}_5\{\!\{v_f \parallel v'_f\}\!\})v_x$$

No reduction rule is applicable at this context, but, from the structure of *app*, we can expand this context to:

$$E \cdot \{e\}_4(\{\circ\}_5\{\!\{v_f \parallel v'_f\}\!\})v_x$$

Similarly, for

$$E \cdot (\{\circ\}_8\{\!\{v_f \parallel v'_f\}\!\})v_x$$

and

$$E \cdot \{e\}_7(\{\circ\}_8\{\!\{v_f \parallel v'_f\}\!\})v_x$$

In either of these cases (E-BAPP1) is applicable and a reduction is possible.

Finally, we observe that it is impossible for $e_1 = \{\circ\}_6\{\!\{v_1 \parallel v_2\}\!\}$. From the source program, we note that such a term is introduced by $e' = e'_1e'_2$ where $e'_2 = \{\circ\}_6\{\!\{v_1 \parallel v_2\}\!\}$. Reduction of e' in this case can only be handled by (E-BAPP1) or (E-BAPP2), since (E-APP) specifically rules out this case. By inspection of (E-BAPP1), if $e'_2 = v_x = \{\circ\}_6\{\!\{v_1 \parallel v_2\}\!\}$, then, in one step of reduction, using the definition of $\lfloor v_x \rfloor$, the $\{\circ\}_6\{\!\{v_1 \parallel v_2\}\!\}$ term is immediately destructed. Similarly for (E-BAPP2). Thus, it is impossible for $\{\circ\}_6\{\!\{v \parallel v'\}\!\}$ to appear in the left-side of an application.

By the above argument, it is also straightforward to establish that if $e_2 = \{\circ\}_6\{\!\{v \parallel v'\}\!\}$, then, although (E-APP) rules out this case, a step is possible via (E-BAPP2).

Case (T-MATCH, T-TAP, T-FIX): In each of these cases, we must consider the possibility that the expression v in redex position is of the form $\{\circ\}\{\!\{v_1 \parallel v_2\}\!\}$. Note that the bracketed term must be prefixed by an unlabeling operator since by (T-BRACKET) $\{\!\{v_1 \parallel v_2\}\!\}$ always has a labeled type, and the premises of each of these rules require unlabeled types for v. Again, the only unlabeling operators in the program are labeled $5,6,8$; but, as our examination in (T-APP) has shown, all these unlabeled bracketed terms are destructed immediately. Thus, the case of $v = \{\circ\}\{\!\{v_1 \parallel v_2\}\!\}$ is ruled out and we can rely on Theorem 5.

□

Lemma 24 (Subject reduction for FABLECORE^2)**.** *Given well-formed $\Omega = \Gamma;\cdot;\cdot$, where $\Gamma = \vec{x} : \vec{t}$ as in Lemma 23, and a ⟦⟧-free* FABLECORE^2 *program e such that $\Omega \vdash_c e : t$. Then,*

$$e \overset{c}{\rightsquigarrow} e' \Rightarrow \Omega \vdash e' : t$$

Proof. By induction on the structure of $e \overset{c}{\rightsquigarrow} e'$. We must consider on the additional rules in Figure F, relying on Theorem 13 for the other cases.

Case (E-APP): Again, v_1 remains a lambda-abstraction as in the FABLECORE case, so no change in the analysis is required. If v_1 contains a bracketed sub-term, then we must extend the substitution lemma to handle the (T-BRACKET) case. However, this is straightforward by using the substitution lemma, Lemma 8 on the fourth and fifth premises, relying on the last premise to establish that the e_i are not nested bracketed values.

Case (E-BAPP1): The left-side of the reduction is typed using :

$$\begin{array}{lll} \mathsf{e} & ::= & () \mid x \mid fix f.\lambda x.\mathsf{e} \mid \mathsf{e}_1\mathsf{e}_2 \quad \textit{expressions} \\ \mathsf{t} & ::= & \mathrm{unit} \mid (\mathsf{t}_1 \to \mathsf{t}_2)^l \quad \textit{types} \end{array} \qquad (\ominus \to \oplus)^{\oplus} \text{ (Subtyping)} \qquad \text{Guards} \quad l \lhd \mathrm{unit} \qquad \frac{\mathscr{L} \vdash l \sqsubseteq l'}{l \lhd (\mathsf{t} \to \mathsf{t}')^{l'}}$$

$$\Gamma \vdash_{ML} () : \mathrm{unit} \text{ (ML-UNIT)} \qquad \Gamma \vdash_{ML} x : \Gamma(x) \text{ (ML-VAR)}$$

$$\frac{\Gamma[x:\mathsf{t}][f:(\mathsf{t}\to\mathsf{t}')^l] \vdash_{ML} \mathsf{e} : (\mathsf{t}\to\mathsf{t}')^l}{\Gamma \vdash_{ML} fix f.\lambda x.\mathsf{e} : (\mathsf{t}\to\mathsf{t}')^l} \text{ (ML-ABS)} \qquad \frac{\Gamma \vdash_{ML} \mathsf{e}_1 : (\mathsf{t}\to\mathsf{t}')^l \quad \Gamma \vdash_{ML} \mathsf{e}_2 : \mathsf{t} \quad l \lhd \mathsf{t}'}{\Gamma \vdash_{ML} \mathsf{e}_1\mathsf{e}_2 : \mathsf{t}'} \text{ (ML-APP)}$$

$$\frac{\Gamma \vdash_{ML} \mathsf{e} : \mathsf{t}' \quad \mathsf{t}' \leq \mathsf{t}}{\Gamma \vdash_{ML} \mathsf{e} : \mathsf{t}} \text{ (ML-SUB)} \qquad t \leq t \text{ (SUB-ID)} \qquad \frac{t_1' \leq t_1 \quad t_2 \leq t_2' \quad l \sqsubseteq l'}{(t_1 \to t_2)^l \leq (t_1' \to t_2')^{l'}} \text{ (SUB-FN)}$$

Figure 21. Core-ML syntax and typing (functional fragment).

$$\frac{\dfrac{}{\Omega \vdash_c \{\circ\}_{5,8}\{\!\{v_f \parallel v_f'\}\!\}} \text{ (T-BRACKET)}}{\Omega \vdash_c \{e\}_{4,7}(\{\circ\}_{5,8}\{\!\{v_f \parallel v_f'\}\!\})v_x : t}$$

It is straightforward to show that $\{\circ\}_{5,8}v_f$ and $\{\circ\}_{5,8}v_f'$ have the same type as $\{\circ\}_{5,8}\{\!\{v_f \parallel v_f'\}\!\}$, using the third and fourth premises of (T-BRACKET). Similarly for v_x and $\lfloor v_x \rfloor_i$, and finally that in $e \overset{c}{\rightsquigarrow} \{\!\{e_1 \parallel e_2\}\!\}$ that e has the same type as e_1 and e_2. The critical point to show is that $\{\!\{e_1 \parallel e_2\}\!\}$ is itself well-typed; i.e. protected at level *HIGH*. However, $\{e\}_4 = \{lublm\}_4$ where $t_f\{l\}$ is the type of $\{\!\{v_f \parallel v_f'\}\!\}$. Thus, by (T-BRACKET), it must be the case that $l = \mathit{HIGH}$. Given that e_i has type t, and we have just established that t is guarded at *HIGH*, we have sufficient evidence to show that $\{\!\{e_1 \parallel e_2\}\!\}$ can be typed using (T-BRACKET) to satisfy its first two premises.

Case (E-BAPP2): Similar to the previous case to show that each e_i is well-typed. Now, must still provide the evidence for the first two premised of (T-BRACKET) in order to type $\{\!\{e_1 \parallel e_2\}\!\}$. However, by inspection of the policy function *app*, in which $\{\circ\}_6$ appears, we can conclude that $t_x :: \mathrm{U}$ and thus has no label. However, as previously $\{e\}_4 = \{lub\, l\, m\}_4$ where $t_x\{m\}$ is the type of $\{\!\{v_x \parallel v_x'\}\!\}$. From, (T-BRACKET) we can conclude that $t_x\{m\}$ is guarded at level *HIGH* and thus t is also guarded at *HIGH*, which is the type of e_1 and e_2. This is sufficient for (T-BRACKET). □

Theorem 25 (Noninterference)**.** *Given $\vec{p} : \vec{t}, x : t\{HIGH\} \vdash_c e : t'\{LOW\}$, where e is $(\!|\,|\!)$-free and $\Omega \vdash t' :: U$; and, for $i = 1,2$, $\cdot \vdash_c v_i : t\{HIGH\}$. Then, for type-respecting substitutions $\sigma_i = (\vec{p} \mapsto \pi, x \mapsto v_i)$, where π is the policy of Figure 19, $\sigma_1(e) \overset{c_*}{\rightsquigarrow} v_1' \wedge \sigma_2(e) \overset{c_*}{\rightsquigarrow} v_2' \Rightarrow v_1' = v_2'$.*

Proof. Straightforward from the substitution lemma, Lemma 8; Lemma 24 and from construction, $\Omega \vdash_c v : t$ where t not guarded at *HIGH*, implies $\lfloor v \rfloor_1 = \lfloor v \rfloor_2$. □

G Completeness of the static information-flow policy

In this section, we show that the information flow policy of Appendix F, Figure 19 is complete with respect to to the purely functional fragment of Pottier and Simonet's Core-ML [33]. Figure 21 reproduces the syntax and the static semantics of a minimal functional fragment of Core-ML.

Definition 26 (Non-degeneracy of Core-ML typing)**.** *A Core-ML type $(t_1 \to t_2)^l$ is non-degenerate if, and only if, $l \lhd t_2$ and both t_1 and t_2 are non-degenerate; unit is non-degenerate. A typing derivation $\mathscr{D} = \Gamma \vdash_{ML} \mathsf{e} : \mathsf{t}$ is non-degenerate if, and only if, for every sub-derivation $\mathscr{D}'$ with conclusion t', t' is non-degenerate.*

The non-degeneracy condition above assures that all function-typed expressions e are given types that permit the application of e. Note the third premise of (ML-APP) that requires the type of the function to be non-degenerate. So, while in programs such as $(\lambda x.0)e_1$, e_1 may be given a degenerate type since it is never applied. It is straightforward to transform a typing derivation for such programs into a non-degenerate derivation.

Figure G shows a translation from Core-ML typing derivations $\mathscr{D}$ to FABLECORE programs e.

Theorem 27 (Completeness of static information flow)**.** *Given e such that, $\mathscr{D} = \Gamma \vdash_{ML} \mathsf{e} : \mathsf{t}$ is non-degenerate; then $[\![\Gamma]\!] \vdash \pi; [\![\mathscr{D}]\!] : [\![\mathsf{t}]\!]$, where π is the policy of Figure 19.*

$[\![\mathsf{t}]\!], [\![\Gamma]\!]$ Translation of Core-ML types and environment

$$[\![\text{unit}]\!] \equiv \mathsf{unit} \qquad \frac{e_l \in \{LOW, MED, HIGH\}}{[\![\text{unit}]\!] \equiv \mathsf{unit}\{e_l\}} \qquad [\![(\mathsf{t}_1 \to \mathsf{t}_2)^l]\!] \equiv ([\![\mathsf{t}_1]\!] \to [\![\mathsf{t}_2]\!])\{[\![l]\!]^{lab}\}$$

$$[\![x : \mathsf{t}, \Gamma]\!]^{env} \equiv x : [\![\mathsf{t}]\!], [\![\Gamma]\!]^{env} \qquad \frac{[\![\Gamma]\!]^{env} \equiv \Gamma}{[\![\Gamma]\!] \equiv \Omega = \Gamma; \cdot; \cdot}$$

$[\![l]\!]^{lab}$ Translation of Core-ML labels to FABLECORE terms

$$[\![L]\!]^{lab} \equiv LOW \qquad [\![M]\!]^{lab} \equiv MED \qquad [\![H]\!]^{lab} \equiv HIGH$$

$[\![\mathscr{D}]\!]$ Translation of derivations $\mathscr{D}$ to FABLECORE expressions.

$$[\![\Gamma \vdash_{ML} () : \text{unit}]\!] \equiv () \text{ (X-U)} \quad [\![\Gamma \vdash_{ML} x : \Gamma(x)]\!] \equiv x \text{ (X-V)} \quad [\![\frac{\mathscr{D}}{\Gamma \vdash_{ML} fix f.\lambda x.e : (\mathsf{t}_1 \to \mathsf{t}_2)^l}]\!] \equiv \mathsf{fix}\ f.\langle [\![(\mathsf{t}_1 \to \mathsf{t}_2)^l]\!] \rangle \lambda x : [\![\mathsf{t}_1]\!].[\![\mathscr{D}]\!] \text{ (X-ABS)}$$

$$[\![\frac{\mathscr{D}_1 = \Gamma \vdash_{ML} e_1 : (\mathsf{t}_1 \to \mathsf{t}_2)^l \qquad \mathscr{D}_2 \qquad l \triangleleft \mathsf{t}_2}{\Gamma \vdash_{ML} e_1 e_2 : \mathsf{t}_2}]\!][(\mathsf{t}_2 = \mathsf{t}')] \equiv join\ [\mathsf{t}_2]\ (app2\ [[\![\mathsf{t}_1]\!]][[\![\mathsf{t}_2]\!]][\![\mathscr{D}_1]\!][\![\mathscr{D}_2]\!]) \text{ (X-APP1)}$$

$$[\![\frac{\mathscr{D}_1 = \Gamma \vdash_{ML} e_1 : (\mathsf{t}_1 \to \mathsf{t}_2)^l \qquad \mathscr{D}_2 \qquad l \triangleleft \mathsf{t}_2}{\Gamma \vdash_{ML} e_1 e_2 : \mathsf{t}_2}]\!][(\mathsf{t}_2 = \text{unit})] \equiv app2\ [[\![\mathsf{t}_1]\!]][[\![\mathsf{t}_2]\!]][\![\mathscr{D}_1]\!][\![\mathscr{D}_2]\!] \text{ (X-APP2)}$$

$$[\![\frac{\mathscr{D} \qquad \mathsf{t} \leq \mathsf{t}'}{\Gamma \vdash_{ML} e : \mathsf{t}'}]\!] \equiv [\![\mathsf{t} \leq \mathsf{t}']\!] \text{ (X-SUB)}$$

$[\![\mathsf{t} \leq \mathsf{t}']\!] \equiv e'$ Translation of the subtyping judgment given as a translation of a $[\![\mathsf{t}]\!]$-typed FABLECORE expression, e.

$$[\![\mathsf{t} \leq \mathsf{t}]\!] \equiv e \text{ (SUB-ID)}$$

$$[\![\frac{\mathscr{D}_1 = \mathsf{t}_1' \leq \mathsf{t}_1 \qquad \mathscr{D}_2 = \mathsf{t}_2 \leq \mathsf{t}_2' \qquad l \sqsubseteq l'}{(\mathsf{t}_1 \to \mathsf{t}_2)^l \leq (\mathsf{t}_1' \to \mathsf{t}_2')^{l'}}]\!] \equiv def\ [t_1' \to t_2']\ e_{l'}\ (\lambda x : t_1'.[\![\mathscr{D}_2]\!]) \text{ (SUB-FN)}$$

where
$t_1 = [\![\mathsf{t}_1]\!], \quad t_1' = [\![\mathsf{t}_1']\!]$
$\mathsf{t}_2 = \mathsf{t}^{l_2}, e_{l_2} = [\![l_2]\!]$
$t_2 = [\![\mathsf{t}_2]\!] = t\{e_{l_2}\}, \quad t_2' = [\![\mathsf{t}_2']\!]$
$e_l = [\![l]\!], e_{l'} = [\![l']\!]^*,$ and,
$e' = join[t](app\ [t_1]\ [t_2]\ e\ [\![\mathscr{D}_1]\!])$

Figure 22. Translation from a Core-ML derivation $\mathscr{D}$ to FABLECORE.

Proof. By induction on the structure of the translation $[\![\mathscr{D}]\!]$.

Case (X-U): Trivial.

Case (X-V): Trivial.

Case (X-ABS): By the induction hypothesis we have

$$[\![\Gamma, f : (\mathsf{t}_1 \to \mathsf{t}_2)^l, x : \mathsf{t}_1]\!] \vdash_c \pi; [\![\mathscr{D}]\!] : [\![\mathsf{t}_2]\!]$$

To establish the conclusion we use, (T-FIX) with (T-SFX) followed by (T-ABS), with the induction hypothesis applicable in the third and fourth premises of (T-ABS), and $\vec{x} = \emptyset$.

Case (X-APP1, X-APP2): By the induction hypothesis we have both

$$\begin{array}{l} i. [\![\Gamma]\!] \vdash \pi; [\![\mathscr{D}_1]\!] : [\![(t_1 \to t_2)^l]\!], \quad \text{and,} \\ ii. [\![\Gamma]\!] \vdash \pi; [\![\mathscr{D}_2]\!] : [\![t_1]\!] \end{array}$$

The type of *app2* is $\forall\alpha{::}\mathsf{M}.\,\forall\beta{::}\mathsf{M}.\varphi l,f{:}(\alpha\to\beta)\{l\} \to x{:}\alpha\to\beta\{l\}$
Thus, we have the type of $app2 \ldots [\![\mathscr{D}_2]\!]$ to be $[\![\mathsf{t}_2^l]\!]$. In case (X-APP2) this is specifically $()\{[\![l]\!]^{lab}\}$ which is an acceptable translation of the Core-ML type unit. In case (X-APP1), this type is specifically $[\![t]\!]\{[\![l']\!]^{lab}\}\{[\![l]\!]^{lab}\}$ which is not yet an acceptable translation of t_2. Thus, we apply $join \ldots$ which has type $\forall\alpha.\varphi l,m,x{:}\alpha\{l\}\{m\} \to \alpha\{lub\ l\ m\}$ to obtain $[\![t]\!]\{lub[\![l']\!]^{lab}[\![l]\!]^{lab}\}$. To conclude, we use the final premise of (ML-APP) which asserts that $l \lhd \mathsf{t}_2$, which requires $l \sqsubseteq l'$. Thus, $lub[\![l']\!]^{lab}[\![l]\!]^{lab} = [\![l']\!]$.

Case (CASE X-SUB): We proceed by induction on the structure of the subtyping derivation using the induction hypothesis to establish that $[\![\Gamma]\!] \vdash \pi; [\![\mathscr{D}]\!] : [\![\mathsf{t}]\!]$. That is, we wish to establish that given

$$\begin{array}{ll} [\![\Gamma]\!] \vdash \pi; e : [\![\mathsf{t}]\!], & \text{then} \\ [\![\Gamma]\!] \vdash \pi; [\![\mathsf{t} \le \mathsf{t}']\!] : [\![\mathsf{t}']\!] & \end{array}$$

The (SUB-ID) case is trivial. We examine first the type of e' in (SUB-FN). By assumption we have that the type of e is $(t_1 \to t_2)\{e_l\}$. We have that $x : t_1'$ by ascription in the lambda binding. Thus, by the inductive hypothesis we have that $[\![\Gamma]\!], x : t_1' \vdash [\![\mathscr{D}_1]\!] : t_1$. Now, using the type for *app* given in (Case X-APP1), we conclude that $app \ldots [\![\mathscr{D}_1]\!]$ has type $t_2\{e_l\}$. After the application of *join* we conclude that e' has type $t\{lub\ e_{l_2}\ e_l\}$. However, from the non-degeneracy assumption, we have $l \lhd \mathsf{t}_2$ or $l_2 \sqsubseteq l$; thus, the type of e' is $t\{e_{l_2}\} = [\![\mathsf{t}_2]\!]$. To type $\lambda x : t_1'.[\![\mathscr{D}_2]\!]$ we use the induction hypothesis to establish that $[\![\mathscr{D}_2]\!]$ has type t_2', to arrive at the type $t_1' \to t_2'$ using (T-ABS) with $\vec{x} = \emptyset$. Finally, the type of *def* is $\forall\alpha{::}\mathsf{M}.l{:}\mathsf{lab} \to \alpha \to \alpha\{l\}$, which is sufficient to establish the type of $(t_1' \to t_2')\{e_{l'}\} \equiv [\![(\mathsf{t}_1' \to \mathsf{t}_2')^{l'}]\!]$ for the translation, which is our goal. □

H Secure Policy Composition

Theorem 28. *Given*

(A1) $\vec{x} : \vec{t}, \vec{y} : \vec{s} \vdash_{app} e : t_e$, *such that e is* $(\!|\cdot|\!)$*-free*

(A2) $\vec{x} : \vec{t}, \vec{y} : \vec{s} \vdash t_e \cong t_e'\{P(\ldots)\}$

(A3) $\{e_1, \ldots, e_n\}$ *such that* $\forall i. \cdot \vdash_{pol} e_i : t_i \quad \wedge \quad \mathsf{Composes}(P, t_i)$

(A4) $\{f_1, \ldots, f_m\}$ $\{g_1, \ldots, g_m\}$ *such that* $\forall i. \cdot \vdash_{pol} f_i : s_i \quad \wedge \quad \cdot \vdash_{pol} g_i : s_i \quad \wedge \quad \mathsf{Composes}(Q, s_i)$ *with* $P \neq Q$

(A5) $\sigma_f = (\vec{x} \mapsto ((\!|e_1|\!), \ldots, (\!|e_n|\!)), \vec{y} \mapsto ((\!|f_1|\!), \ldots, (\!|f_m|\!)))$ *and* $\sigma_g = (\vec{x} \mapsto ((\!|e_1|\!), \ldots, (\!|e_n|\!)), \vec{y} \mapsto ((\!|g_1|\!), \ldots, (\!|g_m|\!)))$

(A6) $\sigma_f(e) \overset{c}{\rightsquigarrow}^* v_f \quad \wedge \quad \sigma_g(e) \overset{c}{\rightsquigarrow}^* v_g$

Then, $v_f = v_g$

Proof. This encoding is an application of the static information flow policy using a degenerate lattice where P and Q are incomparable. The result follows from Theorem 25. □

A policy term that wraps all its types with a P namespace is composable

$$\mathsf{Composes}(P, \mathsf{unit}) \qquad \frac{\mathsf{Composes}(P,t)}{\mathsf{Composes}(P, \forall\, \alpha :: \kappa . t)} \qquad \frac{\cdot \vdash t\{e\} \cong t\{P(\vec{e})\}}{\mathsf{Composes}(P, t\{e\})} \qquad \frac{\mathsf{Composes}(P,t_1) \qquad \mathsf{Composes}(P,t_2)}{\mathsf{Composes}(P, x : t_1 \to t_2)}$$

Figure 23. A type-based composability criterion.