FABLE: A Language for Enforcing User-defined Security Policies

Extended Version; University of Maryland, Technical Report CS-TR-4876

Nikhil Swamy
nswamy@cs.umd.edu

Michael Hicks
mwh@cs.umd.edu

Abstract

An increasing number of applications have need to enforce some kind of security policy. A variety of language-based techniques have been proposed toward assuring that particular sorts of security policy are properly enforced. However, these approaches typically fix the style of security policy and overall security goal, e.g., information flow policies with a goal of noninterference. This limits the programmer's ability to combine policy styles and to apply customized enforcement techniques while still being assured the system is secure.

As a step to addressing this problem, 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 are expressed by associating *security labels* with the data or actions they protect via dependent types. Programmers define the semantics of labels in a separate part of the program called the *enforcement policy*. Labeled terms may only be constructed and destructed within the enforcement policy, ensuring that it cannot be circumvented. FABLE uses substructural types to track effects in labels so that a policy whose enforcement must consider effects (e.g., due to references) or is itself effectful (e.g., a reference monitor) is still assured of complete mediation. FABLE is flexible enough to implement a wide variety of security policies. As examples, we show how FABLE can be used to express policies for access control, information flow (which we prove enforces noninterference), trusted declassification, and security automata.

1. Introduction

With the growth of networked, information-sharing applications, security is becoming increasingly important. At the disposal of the system designer is a myriad of styles of security policy—access control, information flow control, stack inspection, and separation of duty, among others—and a myriad of enforcement mechanisms, including access control lists, capabilities, state-based monitors, and static analysis. While it is one thing to understand the basic concepts, it is another thing to correctly implement security policies and their enforcement in a large, complicated system. Indeed, Security Focus regularly reports violations of complete mediation [23], in which access control checks are bypassed due to a software error. OWASP lists information leakage through improper error handling among the top-ten web-application vulnerabilities in 2007 [19]. Covert channels have been exploited to break cryptographic protocols [4, 15].

To remedy this problem, researchers have proposed new programming languages and analyses for assuring security policies are properly enforced. Dataflow analysis has been used to ensure complete mediation of type enforcement [32] and access control policies [10]. Static tainting analysis [7, 29] can ensure that untrusted data is properly sanitized before it used. Programming languages like Jif [6] and FlowCaml [21] provably enforce *noninterference* for information flow policies based on a security lattice [8]—data tagged at level l can only flow to a context expecting data at level h if $l \sqsubseteq h$ according to the lattice. In these languages, security levels are embedded in the type language and type correctness implies the policy is correctly enforced.

Unfortunately, these approaches to verifying proper policy enforcement are specialized to a particular style of security policy (e.g., access control or information flow), and system security goal (e.g., strict noninterference vs. timing-insensitive noninterference). Application programs often have a wide range of security objectives, not all of which can be easily cast in terms of, say, a single noninterference property. Instead, an application may wish to establish noninterference on particularly sensitive data, but may be content using a weaker access control policy on less sensitive data. For other data, the application may need only to track provenance information without imposing any restrictions on how that data may be used, in order to support auditing. Finally, there might be some extremely sensitive data for which the standard timing-insensitive noninterference property may not be sufficient; for such data, the application could chose to use a more conservative (and less permissive) state-based enforcement strategy [3, 9].

While there have been approaches that integrate two or more security mechanisms within a single system [20, 14], none of these approaches are sufficiently general to capture the diverse security requirements that we have just described. We believe a general-purpose security-programming infrastructure is in order. Programmers should be able to mix and match security policies of various styles, combining well-known techniques with application-specific enforcement strategies and receive an assurance from the system that their custom enforcement strategies cannot be circumvented. Programmers should also be encouraged to formalize and separate the policy enforcement mechanism from the program itself. In this way, one can reason about the high-level security policy, together

The semantics of FABLE as defined in this technical report differs from that in the short version submitted to POPL 2008 in a few respects. There are several extensions that elaborate on details that we elided from the short version. However, a few changes correct errors in the original system that were discovered after our submission. The following list summarizes the main extensions and corrections.

1. The static semantics for FABLE is extended to include an extremely simple form of effect tracking to ensure that type-level expressions do not have side-effects. This restriction was only informally stated in the short version.
2. Previously, we related type t to t' if $\pi \vdash t \rightsquigarrow t'$. We have now made this an equivalence relation by adding the symmetric case. This simplifies our statement and proof of subject reduction.
3. The rule (E-MATCH) has been slightly modified to correct an oversight in the short version.
4. The rule (E-L2) was mistakenly omitted in the short version, causing some reductions to get stuck. This rule allows redundant labeling operations to be stripped away, thereby allowing reduction to progress as required.

with a particular choice of enforcement mechanism, to establish that a system *implementation* complies with stated high-level security objectives.

1.1 Enforcing user-defined security policies

This paper presents FABLE, a core formalism for a programming language that permits programmers to specify custom security policies and the semantics of their enforcement. FABLE has two key elements. First, security policies are associated with program data and actions via *security labels*. This association is expressed via a dependent type $t\{e\}$ where e is a term of type **lab** whose values l are essentially uninterpreted constructor applications of form $C(\vec{l})$. For example, 0-ary constructors LOW and HIGH might be used as information flow labels, while the label on the type **int**{ACL(user)} might represent an access control policy in which only user is allowed to access the given integer.

Second, labeled terms may only be constructed, destructed, and relabeled by a separate part of the program called the *enforcement policy*. By associating labels with the security-sensitive data and operations in the program, the policy designer can ensure that all manipulations of these resources are mediated by the enforcement policy. Because the program must go through the enforcement policy to interact with labeled resources, the enforcement policy, in effect, defines the security semantics of labels.

As an example, suppose there is library function send having the type **socket** → **bytes** → **unit**. The following code snippet shows a simple enforcement policy that restricts what data might be sent on a socket via send.

```
letpol sock_send_dynamic
  λl:lab.λs:socket{l}.λm:lab.λmsg:bytes{m}.
    if policy_allows l m then
      send <socket>s <bytes>msg
    else ()
```

Here, sock_send_dynamic wraps the send function by interposing a policy check prior to sending a message on a security-sensitive socket. The first parameter to this wrapper is a label term l where the next argument has a dependent type **socket**{l}. This indicates that the socket is protected by policy label l. Similarly, the third and fourth arguments are a label term m and the bytes of a message msg with the dependent type **bytes**{m}. Given a labeled socket, there is no way for the program to avoid using this wrapper to send a message on it, as the send function takes an argument of type **socket**, not **socket**{l}. There is likewise no way to send a protected message on an unprotected socket.

Prior to sending msg on s, the sock_send_dynamic wrapper calls out to an external policy (modeled using the function policy_allows) to check that a socket with label l may transmit a message with label m. As send requires arguments of types **socket** and **bytes**, respectively, the wrapper strips off the labels from the types of s and msg using the relabeling operator <>. This operator is only permitted in enforcement policy code. In practice, as our examples throughout the paper will demonstrate, FABLE contains sufficient polymorphism that we can avoid writing separate wrappers for each sensitive function like send, and instead use a few higher-order policy functions.

In the above example, the label l in the type **socket**{l} is used by the program at run time. However, FABLE also allows labels to exist only at compile-time to support purely static enforcement. For example, the following code is a variation of the socket send policy that is enforced statically.

```
letpol sock_send_static
  φm.λs:socket{ALLOW(m)}.λmsg:bytes{m}.
    send <socket>s <bytes>msg
```

Here, the policy designer chooses the form of labels that protect sockets to be of the form ALLOW(l) indicating that messages with security label l are permitted to be sent on the socket. Unlike the previous case where the first argument is expected to be a label term, here, no label needs to be passed in—the syntax φm binds a *phantom term* that may only appear in types as indices. This makes manifest the programmer's intention (which will be verified by the type checker) that enforcement is to be purely static. The next argument is a socket protected by a label of the form ALLOW(m), for any label m, as long as the last argument is a message that is also labeled with m. The type checker ensures that sock_send_static is only called with labeled terms that satisfy this constraint. Such terms are generated by other policy functions.

Enforcement policies are also responsible for labeling terms (e.g., by wrapping input routines) and for transforming labels in parallel with operations on the data. For example, the following code snippet adds two integers whose labels describe an information flow policy.

```
letpol add φl.λx:int{l}.φm.λy:int{m}.
             <int{lub l m}>(<int>x) + (<int>y);
```

Here the label of the sum of the two integers is computed by calling the policy function lub (not shown) that computes the least upper bound of the two labels according to the security lattice. The type checker may reduce label expressions that appear in types to establish type equivalence. For instance, if we have x : **int**{LOW} and y : **int**{HIGH} then a call to add with x and y as arguments will be given type **int**{lub LOW HIGH}. The type checker can try to reduce the expression lub LOW HIGH if necessary (in this case, to the value HIGH). This makes type checking undecidable in general, but the added flexibility is quite useful. Furthermore, since FABLE gives the programmer control over which label terms are present at runtime, it is always possible to resort to a dynamic label check instead of requiring the type checker to establish type equivalence.

1.2 Contributions

We expect to apply ideas from FABLE to a high-level programming language and a typed intermediate language; our goal is to support secure web-programming. In this paper we focus on the core ideas, detailing the design of FABLE, examples of its use, and some of its formal properties, making the following contributions:

- FABLE is a new dependently-typed core language for defining and enforcing user-defined security policies (Section 2). Among other novel features, FABLE introduces the concept of an *enforcement policy* as the mechanism for implementing a traditional security policy, where FABLE makes enforcement policies explicit in the program. We have proven that FABLE is sound (Section 4.3).
- FABLE is flexible enough to implement a wide variety of security and enforcement policies (Section 3). We illustrate examples of access control, static [22] and dynamic information flow [33], trusted declassification [13], and security automaton-based policies [27]; other policies such as stack inspection, tainting and data provenance are straightforward. Given this flexibility, programmers can mix and match various policy styles in an application.
- FABLE facilitates proving that enforcement policies achieve the high-level security goals of the security policies they enforce, in two ways. First, the type system ensures that security-related actions must be mediated by the enforcement policy. Second, the clear separation of the enforcement policy from the program simplifies reasoning about its correctness. To illustrate, we show that our static enforcement policy for information flow policies satisfies noninterference, and show that it is complete

P	$::=$	$\pi; e$	programs: policy and program term
π	$::=$	$\cdot \mid \textbf{letpol}\ \mathtt{x}\ v \mid \pi_1, \pi_2$	policy: a sequence of bindings
e	$::=$	$() \mid C(\vec{e}) \mid x \mid \mathtt{x} \mid \langle t \rangle e$	unit, label, variables, labeling
	$\mid$	$\varphi\vec{y}.\lambda x : t.e$	abstraction with phantom labels $\vec{y}$
	$\mid$	$\textbf{match}\ e\ \textbf{with}\ \vec{x_i}.p_i \rightarrow e_i$	pattern matching
	$\mid$	$\textbf{fix}\ x.v \mid e_1\, e_2$	recursion, application
	$\mid$	$\Lambda\nu{::}\kappa.e \mid e\,[t]$	type abstraction and application
u	$::=$	$() \mid C(\vec{v}) \mid \varphi\vec{x}.\lambda x : t.e \mid \Lambda\alpha{::}\kappa.e \mid x$	pre-values
v	$::=$	$u \mid \langle t \rangle u$	values: bare or labeled pre-values
p	$::=$	$x \mid C(\vec{p})$	patterns
σ	$::=$	$\cdot \mid (x \mapsto e) \mid \sigma_1, \sigma_2$	substitutions
t	$::=$	$\textbf{unit} \mid \nu \mid \forall\nu{::}\kappa.t$	unit, type var, universal type
	$\mid$	$\vec{y}.x : t_1 \rightarrow t_2$	function type
	$\mid$	$\textbf{lab} \mid \textbf{lab}{\sim}e \mid t\{e\}$	labels: any and singleton, labeled type
ν	$::=$	$\alpha \mid \beta \mid \ldots$	type variables
κ	$::=$	$\mathsf{U} \mid \mathsf{M}$	unlabeled and maybe-labeled kinds

Figure 1. Syntax of FABLECORE.

```
letpol login λuser:string.λpw:string.
    let token = match checkpw user pw with
        USER(k) → USER(k)
        _ → FAILED in
    (token, <unit{token}>())

letpol member fix f.λu:lab.λa:lab.
    match a with
        ACL(u, i) → MEMBER
        ACL(j, tl) → f u tl
        _ → NOT_MEMBER

letpol access φk.λu:lab∼USER(k).λcap:unit{u}.
                   Λα.λacl:lab.λdata:α{acl}.
    match member u acl with
        MEMBER → <α>data
        _ → halt#access denied
```

Figure 2. Enforcing a simple access control policy.

with respect to a functional subset of FlowCaml [22] (Section 3.2). Ultimately we hope to partially automate this process, along the lines of user-defined type systems [5].

- We show how substructural types in FABLE can be used to track state, thereby broadening the policies that can be enforced (Section 4). For example, we show that policies involving effects (e.g., due to references) and state transitions (e.g., for reference monitors) are not circumvented.

Section 5 sketches related work, and Section 6 concludes with our plans for future work.

2. FABLECORE: System F with Labels

In this section, we present the syntax, typing, and operational semantics of FABLECORE, the functional core of FABLE; we extend this system to support substructural types in Section 4.

2.1 Syntax

Figure 1 defines the syntax of the language. 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 $\vec{a}$ to be the set of elements $\{a_1, \ldots, a_n\}$.

Programs P consist of a policy π and a program term e. The policy consists of a list of definitions **letpol** $\mathtt{x}$ v which bind variables $\mathtt{x}$ to values v, which are a subcategory of expressions e. Standard expression forms include unit $()$, variables x, term application $e_1\, e_2$, the fixpoint combinator **fix** $x.v$, type abstraction $\Lambda\nu{::}\kappa.e$ and type application $e\,[t]$. As discussed earlier, $C(\vec{e})$ defines labels and $\langle t \rangle e$ is a relabeling of the expression e. We apply a syntactic restriction on source programs which limits the usage of relabelings to the policy π. Note that **letpol**-bound names $\mathtt{x}$ are syntactically distinct from other variables x, and are treated as globals within the main program expression e.

Term abstraction φy1,...,yn.λx:t.e binds two kinds of variables: the λ-bound variable x is standard, while φ-prefixed list $\vec{y}$ binds *phantom label variables*. These represent label terms that require no run-time witness, and are useful for expressing a kind of bounded polymorphism over the label expressions that appear in the first argument's (dependent) type. For applications $e_1\, e_2$, the type checker infers the necessary instantiation by unifying e_2's type with the type of e_1's formal parameter in which the phantom variables appear.

Label terms can be examined via pattern matching. For example, the syntax **match** x **with** a1,a2.JOIN(a1,a2)→e evaluates to e if x matches the pattern JOIN(a1,a2), where occurrences of a1 and a2 are substituted in e with the corresponding components in x's run-time value. Pattern variables like a1 and a2 are explicitly bound in pattern clauses to simplify type checking.

Values v consist of either "pre-values" u—unit, labels containing values, type and term abstractions, and variables—or pre-values preceded by a label operator.

Types include the standard **unit**, type variables ν, and universally-quantified types $\forall\nu{::}\kappa.t$. Functions have dependent type $\vec{y}.x : t_1 \rightarrow t_2$ where $\vec{y}$ lists phantom label variables naming components of t_1 and x names the argument; both $\vec{y}$ and x are bound in t_2. We omit variables from function types when they are not used. Label types have two forms: **lab** can be ascribed to any label term, while **lab**$\sim e$ is a singleton, inhabited only by the label value v that e (and any other label expressions) reduces to, (and none at all if e diverges). Finally, as mentioned earlier, a type may also have an associated label, written $t\{e\}$. Using these constructs we can define the type φy.x:**lab**$\sim$C(y) →**int**{x}. This is the type of a function that takes a label x as its first argument, where the label must be a term with a top-level unary constructor C (and any argument); and it returns an integer that is associated with this label.

Finally, types are grouped into two *kinds*, U, which is ascribed only to unlabeled types (i.e., all but $t\{e\}$), and M (for "may be labeled"), which can be given to any type.

Example Figure 2 illustrates an example enforcement policy for access control. The approach is to label protected data with a list of users who may be granted access to the data. Such data thus has the type $t\{$acl$\}$ where acl encodes the ACL as a label.

First, the policy defines a login function that authenticates a user by checking a password. If authentication succeeds (the first pattern), the oracle returns a user token USER(k) where k is some unique identifier for the user.[1] The login function returns a pair consisting of this token and a unit value 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 λ-bound arguments. Since FABLE enforces that only policies can produce

[1] We use a more convenient syntax in all our examples. This includes the use of **let**; a pattern matching notation in which we use "_" as a wildcard, and in which patterns do not have to be prefixed by their free variables; type abbreviations; and syntax for products and dependent products. Dependent products can be encoded using a term such as the following: $\Lambda\alpha.\lambda$l:**lab**.λx:$\alpha\{$l$\}$.$\Lambda\beta.\lambda$f:(fst:**lab** →snd:$\alpha\{$fst$\}$ →β). f l x. Finally, although our syntax requires **letpol** x v, we often write **letpol** x **fix** f.v, even though **fix** f.v is not a value; this can be easily encoded by unrolling once.

labeled values, we are assured that the term with type **unit**{u} can only have been produced by login.[2]

The access function's last two λ-bound variables consist of the protected data's label, acl, and the data itself, data. The access function calls the (recursive) member function to see whether the user token u is present in the ACL. If successful, the label MEMBER is returned, in which case access returns the data after removing the acl label from its type.

2.2 Typing

Figure 3 defines the type rules for FABLECORE. The top-level typing judgment, $\Omega \vdash P : t$ asserts that program P has type t in the context Ω. The typing context Ω consists of three components: Γ, an environment that binds variables to types and type variables to kinds; a policy π; and an assumption environment A which records the results of runtime checks that occur due to pattern matching. We record three forms of variable binding in Γ—the form $x : t$ records the type of a λ-bound variable; $\mathtt{x} : t$ records the type of a policy global variable; and $\varphi x : t$ records the type of a φ-bound variable (in FABLECORE this is always **lab**). The semantics ensure that φ-bound variables do not appear within expressions that will be evaluated at run-time. The rules use the following notation: $\Omega[x : t]$ is the context Ω with the binding $x : t$ *added* to its Γ component; similarly $\Omega[\pi']$ is the context Ω with the policy π' *added* to its policy component. We will also use $\Omega.\Gamma$ etc. to project components from Ω and $\Omega[\Gamma = \Gamma']$ to denote the context Ω with its Γ component *replaced* with Γ'.

The rule (T-PROG) checks the main program expression e in the context that results from checking all the policy expressions. Checking a single policy binding **letpol** $\mathtt{x}\ v$ occurs in (T-LETPOL). It types the remainder of the policy in a context whose Γ component records the type of $\mathtt{x}$ and whose π component includes the definition itself.

The main judgment $\Omega \vdash_{\varepsilon} e : t$ types expressions. The index ε maintains a phase distinction between type and term-level expressions. Since types are erased at runtime, type-level terms (such as those that appear in relabelings or λ-bindings) can safely use φ-bound variables; term-level expressions cannot. Rule (T-VAR) allows λ-bound variables in either phase, while (T-LVAR) allows φ-bound variables only in the φ (type-level) phase. In the access function of the example, phantom variable k may only appear only in type-expressions (such as **lab**$\sim$USER(k)) while u may appear both in type-level expressions (such as **unit**{u} in the second argument) and in the body of the function.

(T-LAB) gives a label term $C(\vec{e})$ a singleton label type **lab**$\sim C(\vec{e})$ as long as each component $e_i \in \vec{e}$ is **lab**-typed. The rule (T-SUB) allows a singleton label type to be subsumed to the type of all labels, **lab**; (T-SUB2) does the converse, allowing the type of a label to be made more precise.

Rules (T-TAB) for type abstraction, (T-TAP) for type application, and (T-FIX) for fixed-points, are standard.

The first premise of the (T-ABS) rule requires the λ-bound variable x to be fresh. The second premise ensures that all φ-bound variables $\vec{y}$ appear in the argument's type t, and are not already mentioned in Γ. The third premise checks t in a context with phantom variables $\vec{y}$ each given **lab** type. Checking the body e extends the context further to include the λ-bound variable x.

The first premise of (T-APP) requires e_1 to have an unlabeled function type. The third premise of (T-APP) checks that the type of t_1' of argument e_2 can be unified with x's type t_1, producing a substitution of the phantom variables $\vec{y}$ (we explain the rules for unification below). For example, when typing the application access USER(Joe), the judgment $k;\Omega.A \vdash$ **lab**$\sim$USER(Joe) $\leq$ **lab**$\sim$USER(k) : σ, produces $\sigma = (\mathsf{k} \mapsto \mathsf{Joe})$ as the result. We apply the substitution σ, augmented with the substitution for the λ-bound variable, to the return type. In our example, the type of the application would be **unit**{USER(Joe)} $\rightarrow \forall\alpha.$ (acl:**lab**) $\rightarrow \alpha$\{acl\} $\rightarrow \alpha$.

The first two premises of (T-MATCH) ensures pattern variables are distinct for each case and that each pattern is prefixed by exactly its free variables that do not occur in the context. Our patterns differ from patterns in, say, ML in that they are allowed to contain variables that are defined in the context. This is convenient in the absence of an equality test. We always require a default case in pattern matching expressions: the third premise requires the last clause to be of the form $x_{default}.x_{default} \rightarrow e_n$. The fourth premise ensures that the matched expression e is a label; the fifth ensures that each pattern p_i is well-formed assuming its free variables each have type **lab**; the final premise checks the body of each branch e_i in a context extended with the assumption that the expression e matches pattern p_i (similar to typecase [11]). These assumptions are used in the unification judgment $t_1 \leq t_2 : \sigma$ used by (T-APP). The following example illustrates how this feature can be used:

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

We give the login function the type **string** $\rightarrow$ **string** $\rightarrow$ (l:**lab** $\times$ **unit**{l}) where (l:**lab** $\times$ **unit**{l}) is a dependent product type. l = USER(_) and a unit with labeled type **unit**{l} as the arguments. To type check the application access tok we must show that **lab**$\sim$tok, the type of tok, is unifiable with USER(k), the type of the formal parameter. This is possible since we check the application in the first branch in a context that includes the assumption tok $\leq$ USER(j). Similarly, we can check that the type of cap, **unit**{tok} is unifiable with **unit**{USER(j)} in the presence of the same assumption.

The rules (T-PFX) and (T-SFX) type the relabeling operation, which can remove or add label associations but not change the base type. Note that the label expressions in (T-SFX) are checked in the type phase; i.e., the index is φ. Also note that the type rules do not restrict the relabeling operator from occurring in normal program terms. This is to facilitate the proof of subject reduction—after a program reduces a policy function, its relabelings may appear in the program term. When the program is initially type-checked, occurrences of relabeling would be flagged.

Finally, (T-EVT) allows expressions that appear in types to be reduced according to the operational semantics of FABLECORE. The relation $\pi \vdash t \rightsquigarrow t'$ simply lifts the standard operational semantics on expressions $\pi \vdash e \rightsquigarrow e'$ to types. This reslation relies on the definition of the type evaluation context T, which contains two kinds of hole $\circ$ and $\bullet$, which can be filled with types and expressions respectively.

The judgment $\Omega \vdash t :: \kappa$ assigns a kind to a type. (K-SUB) allows a type of U kind to be given kind M. (K-SEC) gives kind Mto a labeled type. Notice that in (K-SEC) the label is checked in φ-phase. (K-ABS) defines the scoping rules for names in dependent function types: the phantom variables are in scope in both t_1 and t_2; the λ-bound variable is in scope within t_2 only.

The judgment $\vec{x_2};A \vdash t_1 \leq t_2 : \sigma$ states that in a context where $\vec{x_2}$ are phantom variables free in t_2, and A is a set of matched pattern assumptions, t_1 is unifiable with t_2 producing a substitution σ where $\mathrm{dom}(\sigma) = \vec{x_2}$. This is a highly-restricted form of semi-unification [12]; the substitution σ only applies to label variables and does not alter the structure of the type. (U-ID) is trivial. (U-LAB) lifts the corresponding judgment for expressions for use with singleton label types. (U-UNI) unifies universal types by unifying the bound type variables after stripping them off, and then unifying

[2] One assumption made here is that the labels returned by checkpw will not overlap with labels from other policies in force in the same program. As future work we could extend FABLE to include policy modules with support for type abstraction to avoid namespace collisions.

Syntactic forms used in judgments

Γ	$::=$	$x:t \mid \mathtt{x}:t \mid \varphi x:t \mid \nu :: \kappa \mid \Gamma_1,\Gamma_2$	variable bindings, policy global names, phantom variables, type variables
A	$::=$	$\cdot \mid e_1 \leq e_2 \mid A_1,A_2$	Pattern matching assumptions
Ω	$::=$	$\Gamma;\pi;A$	Typing context triple of bindings, policy and assumptions
ε	$::=$	$\cdot \mid \varphi$	Index used to indicate type-checking phase
E	$::=$	$\bullet \mid \bullet\ e \mid v \bullet \mid \bullet\ [t] \mid C(\vec{v},\bullet,\vec{e}) \mid \mathbf{match}\bullet\ \mathbf{with}\ \vec{x}.p_i \rightarrow e_i \mid \langle t\rangle\bullet$	Evaluation contexts

$\Omega \vdash P : t \quad ; \quad \Omega \vdash \pi : \Omega'$ — Top-level judgments for typing programs and policies

$$\frac{\Omega \vdash \pi : \Omega' \qquad \Omega' \vdash e : t}{\Omega \vdash \pi; e : t}\ \text{(T-PROG)} \qquad \frac{\Omega.\Gamma(\mathtt{x})\ \mathit{undefined} \qquad \Omega \vdash v : t \qquad \Omega[\mathtt{x}:t][\mathbf{letpol}\ \mathtt{x}\ v] \vdash \pi' : \Omega'}{\Omega \vdash \mathbf{letpol}\ \mathtt{x}\ v, \pi' : \Omega'}\ \text{(T-LETPOL)} \qquad \Omega \vdash . : \Omega\ \text{(T-EMPTY)}$$

$\Omega \vdash_{\varepsilon} e : t$ — Expression e has type t in environment Ω

$$\Omega \vdash_{\varepsilon} () : \mathbf{unit}\ \text{(T-UNIT)} \qquad \frac{x:t \in \Omega.\Gamma}{\Omega \vdash_{\varepsilon} x : t}\ \text{(T-VAR)} \qquad \frac{\varphi x : t \in \Omega.\Gamma}{\Omega \vdash_{\varphi} x : t}\ \text{(T-LVAR)} \qquad \frac{\forall i. \Omega \vdash_{\varepsilon} e_i : \mathbf{lab}}{\Omega \vdash_{\varepsilon} C(\vec{e}) : \mathbf{lab}\sim C(\vec{e})}\ \text{(T-LAB)} \qquad \frac{\Omega[f:t] \vdash_{\varepsilon} v : t}{\Omega \vdash_{\varepsilon} \mathbf{fix}\ f.v : t}\ \text{(T-FIX)}$$

$$\frac{\begin{array}{c}\Omega.\Gamma(x)\ \mathit{undefined} \qquad \vec{y} = FV(t) \setminus \mathrm{dom}(\Omega.\Gamma)\\ \Omega[\varphi\vec{y}:\mathbf{lab}] \vdash t :: \kappa \qquad \Omega[\varphi\vec{y}:\mathbf{lab}, x:t] \vdash_{\varepsilon} e : t'\end{array}}{\Omega \vdash_{\varepsilon} \varphi\vec{y}.\lambda x:t.e : (\vec{y}.x:t) \rightarrow t'}\ \text{(T-ABS)} \qquad \frac{\begin{array}{c}\Omega.\Gamma(\alpha)\ \mathit{undefined}\\ \Omega[\alpha::\kappa] \vdash_{\varepsilon} e : t\end{array}}{\Omega \vdash_{\varepsilon} \Lambda\alpha::\kappa.e : \forall\alpha::\kappa.t}\ \text{(T-TAB)} \qquad \frac{\begin{array}{c}\Omega \vdash t :: \kappa\\ \Omega \vdash_{\varepsilon} e : \forall\alpha::\kappa.t'\end{array}}{\Omega \vdash_{\varepsilon} e[t] : (\alpha \mapsto t)t'}\ \text{(T-TAP)}$$

$$\frac{\begin{array}{c}\Omega \vdash_{\varepsilon} e_1 : (\vec{y}.x:t_1) \rightarrow t_2 \qquad \Gamma \vdash_{\varepsilon} e_2 : t_1'\\ \vec{y};\Omega.A \vdash t_1' \leq t_1 : \sigma \qquad \sigma' = \sigma, (x \mapsto e_2)\end{array}}{\Omega \vdash_{\varepsilon} e_1\, e_2 : \sigma'(t_2)}\ \text{(T-APP)} \qquad \frac{\begin{array}{c}\forall i,j. i \neq j\ \ (\vec{x_i} \cap \vec{x_j} = \emptyset) \qquad (\vec{x_i} = FV(p_i) \setminus \mathrm{dom}(\Omega.\Gamma)) \qquad \vec{x_n} = p_n = x_{default}\\ \Omega \vdash_{\varepsilon} e : \mathbf{lab} \qquad \Omega[\vec{x_i}:\mathbf{lab}] \vdash_{\varepsilon} p_i : \mathbf{lab} \qquad \Omega[\vec{x_i}:\mathbf{lab}][e \leq p_i] \vdash_{\varepsilon} e_i : t\end{array}}{\Omega \vdash_{\varepsilon} \mathbf{match}\ e\ \mathbf{with}\ \vec{x_1}.p_1 \rightarrow e_1 \ldots \vec{x_n}.p_n \rightarrow e_n : t}\ \text{(T-MATCH)}$$

$$\frac{\Omega \vdash_{\varepsilon} e : t\{\vec{e'}\}}{\Omega \vdash_{\varepsilon} \langle t\rangle e : t}\ \text{(T-PFX)} \qquad \frac{\begin{array}{c}\Omega \vdash_{\varepsilon} e : t\\ \forall i. \Omega \vdash_{\varphi} e_{l_i} : \mathbf{lab}\end{array}}{\Omega \vdash_{\varepsilon} \langle t\{\vec{e_l}\}\rangle e : t\{\vec{e_l}\}}\ \text{(T-SFX)} \qquad \frac{\Omega \vdash_{\varepsilon} e : \mathbf{lab}\sim e'}{\Omega \vdash_{\varepsilon} e : \mathbf{lab}}\ \text{(T-SUB)} \qquad \frac{\Omega \vdash_{\varepsilon} e : \mathbf{lab}}{\Omega \vdash_{\varepsilon} e : \mathbf{lab}\sim e}\ \text{(T-SUB2)} \qquad \frac{\begin{array}{c}\Omega \vdash_{\varepsilon} e : t\\ \Omega.\pi \vdash t \rightsquigarrow t'\end{array}}{\Omega \vdash_{\varepsilon} e : t'}\ \text{(T-EVT)}$$

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

$$\frac{\Omega \vdash t :: \kappa}{\Omega \vdash t :: \mathrm{M}}\ \text{(K-SUB)} \qquad \Omega \vdash \mathbf{unit} :: \mathrm{U}\ \text{(K-UNIT)} \qquad \frac{\alpha :: \kappa \in \Omega.\Gamma}{\Omega \vdash \alpha :: \kappa}\ \text{(K-VAR)} \qquad \frac{\Omega \vdash t :: \kappa \quad \Omega \vdash_{\varphi} e : \mathbf{lab}}{\Omega \vdash t\{e\} :: \mathrm{M}}\ \text{(K-SEC)} \qquad \Omega \vdash \mathbf{lab} :: \mathrm{U}\ \text{(K-LAB)}$$

$$\frac{\Omega \vdash_{\varphi} e : \mathbf{lab}}{\Omega \vdash \mathbf{lab}\sim e :: \mathrm{U}}\ \text{(K-ELAB)} \qquad \frac{\begin{array}{c}\Omega.\Gamma(x,\vec{y})\ \mathit{undefined} \qquad \Omega[\varphi\vec{y}:\mathbf{lab}] \vdash t_1 :: \kappa'\\ \Omega[\varphi\vec{y}:\mathbf{lab}, x:t_1] \vdash t_2 :: \kappa\end{array}}{\Omega \vdash (\vec{y}.x:t_1) \rightarrow t_2 :: \mathrm{U}}\ \text{(K-ABS)} \qquad \frac{\begin{array}{c}\Omega.\Gamma(\alpha)\ \mathit{undefined}\\ \Omega[\alpha::\kappa] \vdash t :: \kappa'\end{array}}{\Omega \vdash \forall\alpha::\kappa.t :: \mathrm{U}}\ \text{(K-UNIV)}$$

$\vec{x_2};A \vdash t_1 \leq t_2 : \sigma \quad ; \quad \vec{x_1};\vec{x_2};A \vdash e_1 \leq e_2 : \sigma$ — Unification of types and expressions

$$\vec{x_2};A \vdash t \leq t : \cdot\ \text{(U-ID)} \qquad \frac{\cdot;\vec{x_2};A \vdash e_1 \leq e_2 : \sigma}{\vec{x_2};A \vdash \mathbf{lab}\sim e_1 \leq \mathbf{lab}\sim e_2 : \sigma}\ \text{(U-LAB)} \qquad \frac{\vec{x_2};A \vdash t_1 \leq (\beta \mapsto \alpha)t_2 : \sigma}{\vec{x_2};A \vdash \forall\alpha::\kappa.t_1 \leq \forall\beta::\kappa.t_2 : \sigma}\ \text{(U-UNI)}$$

$$\frac{\begin{array}{c}\sigma_0 = (y \mapsto x, \vec{y} \mapsto \vec{x}) \qquad \vec{x_2};A \vdash t_1 \leq \sigma_0 t_1' : \sigma_1\\ \vec{x_2};A \vdash t_2 \leq \sigma_1\sigma_0 t_2' : \sigma_2\end{array}}{\vec{x_2};A \vdash (\vec{x}.x:t_1) \rightarrow t_2 \leq (\vec{y}.y:t_1') \rightarrow t_2' : \sigma_1,\sigma_2}\ \text{(U-FUN)} \qquad \frac{\begin{array}{c}\vec{x_2};A \vdash t_1 \leq t_2 : \sigma\\ \cdot;\vec{x_2};A \vdash e_1 \leq \sigma(e_2) : \sigma'\end{array}}{\vec{x_2};A \vdash t_1\{e_1\} \leq t_2\{e_2\} : \sigma,\sigma'}\ \text{(U-SEC)} \qquad \vec{x_1};\vec{x_2};A \vdash e \leq e : \cdot\ \text{(U-EXPID)}$$

$$\frac{\begin{array}{c}\forall i. \sigma_i^* = (\sigma_i,\ldots,\sigma_{i-1})\\ \vec{x_1};\vec{x_2};A \vdash \sigma_i^* e_i \leq \sigma_i^* e_i' : \sigma_i\end{array}}{\vec{x_1};\vec{x_2};A \vdash C(\vec{e}) \leq C(\vec{e'}) : \vec{\sigma}}\ \text{(U-CON)} \qquad \frac{\begin{array}{c}A = A_1, e_1 \leq e_1', A_2\\ \vec{x_1};\vec{x_2};A_1,A_2 \vdash e_1' \leq e_2 : \sigma\end{array}}{\vec{x_1};\vec{x_2};A \vdash e_1 \leq e_2 : \sigma}\ \text{(U-AS)} \qquad \frac{\begin{array}{c}x \in \vec{x_1}\\ \sigma = x \mapsto v\end{array}}{\vec{x_1};\vec{x_2};A \vdash x \leq v : \sigma}\ \text{(U-VL)} \qquad \frac{\begin{array}{c}x \in \vec{x_2}\\ \sigma = x \mapsto v\end{array}}{\vec{x_1};\vec{x_2};A \vdash v \leq x : \sigma}\ \text{(U-VR)}$$

$\pi \vdash e \rightsquigarrow e'$ — Small-step typed reduction rules

$$\frac{\pi \vdash e \rightsquigarrow e'}{\pi \vdash E\cdot e \rightsquigarrow E\cdot e'}\ \text{(E-CTX)} \qquad \frac{\mathbf{letpol}\ \mathtt{x}\ v \in \pi}{\pi \vdash \mathtt{x} \rightsquigarrow v}\ \text{(E-POL)} \qquad \pi \vdash \Lambda\alpha::\kappa.e\ [t] \rightsquigarrow [\alpha \mapsto t]e\ \text{(E-TAP)} \qquad \frac{v_1 = \varphi\vec{x}.\lambda x:t.e}{\pi \vdash v_1^{\sigma} v_2 \rightsquigarrow (\sigma, x \mapsto v_2)e}\ \text{(E-APP)}$$

$$\frac{\begin{array}{c}e = \mathbf{fix}\ f.v\\ v' = (f \mapsto e)v\end{array}}{\pi \vdash e \rightsquigarrow (f \mapsto v')v}\ \text{(E-FIX)} \quad \pi \vdash \langle t\rangle\langle t'\rangle u \rightsquigarrow \langle t\rangle u\ \text{(E-L1)} \quad \frac{t \neq t'\{e\}}{\pi \vdash \langle t\rangle u \rightsquigarrow u}\ \text{(E-L2)} \quad \frac{\begin{array}{c}\forall i < j. FV(v);FV(p_i);\cdot \nvdash v \leq p_i : \sigma_i\\ FV(v);FV(p_j);\cdot \vdash v \leq p_j : \sigma_j \qquad \mathrm{dom}(\sigma_j) \subseteq \vec{b_j}\end{array}}{\pi \vdash \mathbf{match}\ v\ \mathbf{with}\ \vec{b_i}.p_i \rightarrow e_i \rightsquigarrow \sigma_j(e_j)}\ \text{(E-MATCH)}$$

$\pi \vdash t \rightsquigarrow t'$ — Evaluation relation on terms lifted to types

$$T\ ::=\ \text{◑} \mid \text{◑}\{e\} \mid (\vec{y}.x:\text{◑}) \rightarrow t \mid (\vec{y}.x:t) \rightarrow \text{◑} \mid \forall\alpha::\kappa.\text{◑} \mid \mathbf{lab}\sim\bullet \mid t\{\bullet\}$$

$$\pi \vdash t \rightsquigarrow t\ \text{(TE-ID)} \qquad \frac{\pi \vdash t \rightsquigarrow t'}{\pi \vdash T.t \rightsquigarrow T.t'}\ \text{(TE-CTXT)} \qquad \frac{\pi \vdash e \rightsquigarrow e'}{\pi \vdash T\cdot e \rightsquigarrow T\cdot e'}\ \text{(TE-CTXE1)} \qquad \frac{\pi \vdash e' \rightsquigarrow e}{\pi \vdash T\cdot e \rightsquigarrow T\cdot e'}\ \text{(TE-CTXE2)}$$

Figure 3. Semantics of FABLECORE.

the bare types. (U-FUN) is similar to (U-UNI) in that the bound phantom variables are renamed (by σ_0) in t'_1 and in t'_2. Prior to unifying t'_2 with t_2, the third premise requires all free variables in t'_2 also in t'_1 to be substituted using the substitution σ_1 produced by unifying t_1 and t'_1. (U-SEC) is similar to (U-FUN).

The judgment $\vec{x_1};\vec{x_2};A \vdash e_1 \leq e_2 : \sigma$ is more general than the corresponding judgment for types. Here, given $\vec{x_1}$ the free variables in e_1 and $\vec{x_2}$ the free variables in e_2 and a set of assumptions A, σ is a substitution where $\mathrm{dom}(\sigma) \subseteq \vec{x_1} \cup \vec{x_2}$ such that $\sigma(e_1) = \sigma(e_2)$. When attempting to unify types t_1 and t_2, this judgment is always used in the form where $\vec{x_1}$ is empty. However, in the operational semantics, the full generality of this unification judgment is sometimes necessary during pattern matching.

The rule (U-EXPID) is trivial. (U-CON) is similar to (U-SEC) in that unification decisions from sub-expressions are propagated before proceeding. (U-AS) is the only rule that uses the assumption context A. For instance, when trying to show that $\mathsf{tok} \leq \mathsf{USER(k)}$ in a context that includes $\mathsf{tok} \leq \mathsf{USER(j)}$ as an assumption, it suffices to unify $\mathsf{USER(j)} \leq \mathsf{USER(k)} : (k \mapsto j)$. Finally, (U-VL) and (U-VR) allow substitutions of free variables on the lhs and rhs, respectively.

2.3 Operational semantics

The remainder of Figure 3 defines the operational semantics. We define a small-step reduction relation $\pi \vdash e \rightsquigarrow e'$ using evaluation contexts E, where the hole is written $\bullet$. Context application is written with a dot, e.g., $E \cdot e$, instead of the usual brackets $E[e]$, to avoid confusion with the type application notation. The evaluation contexts are standard and specify a left-to-right evaluation order for a call-by-value semantics. The judgment $\pi \vdash t \rightsquigarrow t'$ lifts term evaluation to types (used by (T-EVT)).

The (E-POL) rule looks up a global policy binding in π, rather substituting it away, following Sewell et al.'s *redex-time* reduction strategy. This is observationally equivalent to the standard semantics [24], and preserves π during evaluation. (E-TAP) and (E-FIX) are standard. (E-LAB) strips redundant labelings, preserving the outermost one. (E-APP) is also standard, though to prove subject reduction we assume a substitution σ for bound phantom labels is available. As mentioned earlier, this can be constructed during typing.

The (E-MATCH) rule is complicated by the fact that while runtime values are closed terms, (T-EVT) can be applied during typing to reduce patterns that match values containing free variables. When v contains no free variables, (E-MATCH) is straightforward. The first premise identifies pattern p_j as the first that value v does not fail to match. The second premise produces a substitution σ_j for that match. In this case, the third premise is trivial since $FV(v) = \emptyset$. Thus reduction proceeds by applying the substitution to the body of the pattern.

For the case that v contains free variables, i.e., during typing, we might have a situation illustrated in the following example.

```
letpol lub λl:lab.λm:lab.match l, m with
      l, l → l
      LOW, MED → MED
      MED, HIGH → HIGH
      ...
letpol f φl.λx:int{l}.<int{lub l LOW}>x
```

One type of the function f is $t_1 = \varphi \mathsf{l.x}{:}\mathbf{int}\{\mathsf{l}\} \rightarrow \mathbf{int}\{\mathsf{lub\ l\ LOW}\}$. We permit the type checker to attempt to reduce $\mathsf{lub\ l\ LOW}$, even though it contains a free variable l. (If the type checker is able to reduce this using : $\pi \vdash \mathsf{lub\ l\ LOW} \rightsquigarrow e$, then it can also give f the type $t_2 = \varphi \mathsf{l.x}{:}\mathbf{int}\{\mathsf{l}\} \rightarrow \mathbf{int}\{e\}$.) Now consider a usage of this function $\mathsf{f} <\mathbf{int}\{\mathsf{MED}\}>0$. The type given to this expression should not depend on the choice of t_1 or t_2 as the type of f. That is, given $\sigma = (\mathsf{l} \mapsto \mathsf{MED})$ we require,

$$\pi \vdash \sigma(\mathsf{lub\ l\ LOW}) \rightsquigarrow^* e' \iff \pi \vdash \sigma(e) \rightsquigarrow^* e'$$

To enforce this property, we must take care when matching label terms containing free variables. In our example, attempting to reduce $\mathsf{lub\ l\ LOW}$ requires matching l against LOW, or against MED etc. Since the value of l varies depending on the context in which the function f is used (our example usage gives the l the value MED, but other usages might give other values to l), we cannot determine which branch of the match statement to take; at this point, the only sound option is to prevent further reduction of the expression.

By contrast, consider the following function: $\varphi \mathsf{l}.\lambda \mathsf{x}{:}\mathbf{int}\{\mathsf{l}\}.\ \mathsf{add\ x\ x}$. This function doubles its argument x using the add function defined in Section 1. We can type $\mathsf{add\ x\ x}$ as $\mathbf{int}\{\mathsf{lub\ l\ l}\}$. However, since lub defines a lattice, we would like to be able to reduce this to just l. When attempting to reduce $\mathsf{lub\ l\ l}$, it is easy to see that the first pattern in the definition of lub will always be matched, irrespective of the substitution chosen for l. Thus, we can permit the reduction $\pi \vdash \mathsf{lub\ l\ l} \rightsquigarrow^* \mathsf{l}$.

There are many possible techniques that can be used to determine when a reduction is permissible. One (not very good) option is to analyze the whole program. If it can be established that the only usage of f instantiates l to MED, then we can proceed with reduction on this assumption for l. We chose a much simpler condition that states that if it is possible to substitute the free variables in a label term so that it unifies with a pattern, then we must prevent further reduction. However, if no free variable in the label term needs to be substituted to match a pattern, then reduction can proceed. We opted for this property in (E-MATCH) because it is relatively simple to state. There are other good choices, too. For example, we can allow type reduction to proceed using type information—if the declared type of l is $\mathbf{lab}\sim\mathsf{LOW}$, then we can use this information to decide which branch to take. Additionally, we can use the context $\Omega.A$ that reflects the result of runtime checks to further refine the analysis. Our technical report [25] discusses these last two options in greater detail.

We defer stating a formal soundness property for FABLECORE until Section 4.3, where we state progress and preservation theorems for the full FABLE language.

3. Example Policies Encoded in FABLECORE

This section demonstrates the expressive power of FABLECORE by presenting encodings of a range of policies. Section 2.1 illustrated an access control policy; here we focus on information flow policies. The first two examples illustrate languages with noninterference properties, supporting compile-time labels, as in FlowCaml [22], and dynamic labels, in the style of Zheng and Myers [33], respectively. The last example illustrates a downgrading policy in the style of trusted declassifiers [13]. We have proved that our first encoding satisfies noninterference and is complete with respect to the functional subset of FlowCaml. Encodings of state-based policies are presented in the next section.

3.1 Static Information Flow

Information flow policies based on a security lattice [8] indicate that data labeled with level h in the lattice may not flow to contexts expecting data labeled l if $h \sqsupset l$. Static information flow type systems, exemplified by FlowCaml [22], enforce this property by labeling program types with security levels and defining subsumption according to the lattice: $t_l \leq t_h \iff l \sqsubseteq h$. In these systems, labels have no runtime witness—assuming the initial assignment of types to values is correct and the policy does not change during execution, the type checker can prove the program is secure.

Figure 4 illustrates a static information flow enforcement policy in FABLE, along with a small sample program. We define several

```
letpol lub λx:lab.λy:lab. match x,y with
           x, x → x
           LOW, _ → y
           MED, LOW → x
           MED, _ → y
           _ , _ → x

letpol low Λα.λx:α.<α{LOW}>x

letpol join Λα.φl,m.λx:α{l}{m}.<α{lub l m}>x

letpol sub Λα.φl.λx:α{l}.λm:lab.<α{lub l m}>x

letpol app Λα.Λβ.φl.λf:(α → β){l}.φm.λx:α{m}.
             <β{lub l m}> (<α → β>f) (<α> x)

abbrv bool{l} = ∀α.(α → α→ α){l}
Λα.λb: bool{HIGH}.λx:α{LOW}.λy: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 4. Enforcing a static information flow policy.

policy functions. lub implements the least-upper-bound operation for the three-point security lattice LOW $\sqsubset$ MED $\sqsubset$ HIGH. Expressions protected by the policy will have labeled type, where the label is drawn from this lattice; an initial labeling of LOW can be added via the low function. The join function can be used to combine multiple labels on a type into a single label.

Next, the policy defines the basic subsumption function sub, which takes as arguments a term x with type α\{l\} and a label m and allows x to be used at the type α\{lub l m\}. This is a restatement of the subsumption rule above, as $l \sqsubseteq m$ implies $l \sqcup m = m$.

When a function f is applied, the result of the application carries information about both the identity of f as well as the arguments to f. This behavior is specified by the app policy function which must be called to apply a labeled function to a labeled argument. It takes two arguments f and x; strips the labels of f and x and applies f to x; and, importantly, ensures that the result is protected at a level no lower than the least upper bound of the security level l of the function and m of the argument. It does this by adding lub l m to the result. Note that the app function, as defined here, relies on the fact that FABLECORE programs do not have side-effects; this is to keep the example simple. Our technical report [25] shows how to use the ideas of Section 4.4 to enforce an information flow policy for effectful programs.

The non-policy function at the bottom of the figure takes a HIGH-security boolean b, and LOW- and MED-labeled arguments x and y, respectively. If b is true, then the function returns x otherwise y. However, since the returned value depends on the value of HIGH-labeled boolean, the security level of the return value must be no lower than HIGH—this is the classic example of the kind of implicit flow that is tracked by an information flow policy. The function can be given the following type:
$\forall\alpha.$ **bool**\{HIGH\} $\to \alpha$\{LOW\} $\to \alpha$\{MED\} $\to \alpha$\{HIGH\}
This type reflects the control dependence of the output on the HIGH input, thereby tracking the implicit information flow.

In this example, we use a Church encoding of booleans as defined by the type abbreviation **bool**\{l\}, where the top-level label l defines the security level of the boolean value. The first line in the body of the function instantiates the type variable in the boolean with the type of the values to be returned from each branch of the if-expression, in this case α\{MED\}. Since we will pass x to this function, it must have type α\{MED\}, so we use sub to coerce x from label LOW to x' having label lub LOW MED, which is reduced by the type checker to MED. The third line passes x' to b', giving tmp the type ($\alpha\to\alpha$)\{lub HIGH MED\}; the next line passes in y producing the final result of type α\{lub (lub HIGH MED) MED\}. This type reduces to α\{HIGH\}.

3.2 Noninterference for the Static Information Flow Policy

While it is evident that labels are completely mediated by the policy, it remains to be shown that the policy meets the system's overall security goals. For an information flow policy, this goal is typically some kind of noninterference property.

Theorem (Noninterference)**.** *Given π, the policy of Figure 4, and an expression e that does not contain any relabeling operations, and an empty context Ω ($\Omega.\Gamma = \Omega.\pi = \Omega.A = \cdot$) such that $\Omega[x : t\{\mathsf{HIGH}\}] \vdash \pi; e : t'\{\mathsf{LOW}\}$, where t' is of U-kind; and values v_1 and v_2:*

$$(\forall i \in \{1,2\}.\Omega \vdash \pi; v_i : t\{\mathsf{HIGH}\} \wedge \pi \vdash (x \mapsto v_i)e \leadsto^* v_i') \Rightarrow v_1' = v_2'$$

To prove this theorem, we follow the proof technique of Pottier and Simonet [21] to represent a pair of FABLECORE executions within a single bracketed expression in an extension of FABLECORE. This involves adding an extra typing judgment of the form shown below, to indicate that differences between the two executions only occur within HIGH-security expressions.

$$\frac{\Omega \vdash_{\bar{\varepsilon}} e_1 : t\{\mathsf{HIGH}\} \qquad \Omega \vdash_{\bar{\varepsilon}} e_2 : t\{\mathsf{HIGH}\}}{\Omega \vdash_{\bar{\varepsilon}} \{\!\{e_1 \parallel e_2\}\!\} : t\{\mathsf{HIGH}\}} \text{ (T-BRACKET)}$$

We also must add additional operational rules to allow reduction to proceed inside bracketed terms. The goal then is to show subject reduction for this augmented language; noninterference follows directly. The proof is straightforward since reduction within each side of the bracket obeys the normal typing and evaluation rules of FABLECORE. This allows us to use the subject reduction property of FABLECORE as a lemma; the rest of the proof reasons about the way each of the policy functions use the relabeling operation.

We believe this general approach should work for other security policies as well. In a sense, this technique elevates the policy expressed in FABLE to the level of axioms and rules in the proof system which can then be used to prove the desired properties. In future work we hope to mechanize this idea, thus sharing the goals of user-defined type systems [5].

We have also proved that our static information flow policy is at least as expressive as policies that are enforceable in (a core functional subset of) Core-ML, the formal language of FlowCaml [21]. The proof proceeds by a translation from Core-ML *typing derivations* to FABLECORE programs. The main typing judgment in Core-ML is of the form $\mathscr{L};\Gamma \vdash_{ML} \mathsf{e} : \mathsf{t}$. The judgment states that the Core-ML expression e has type t in a context where $\mathscr{L}$ is the lattice of security labels, and Γ is the usual context associating variables with their types. We define a family of translations, $[\![\cdot]\!]$, that relate Core-ML types, contexts and derivations to FABLECORE types, contexts and expressions respectively. The translation is mostly straightforward; the one tricky case is translating usages of the subsumption rule in Core-ML. We show how to rewrite λ-terms in Core-ML by inserting the appropriate calls to sub; this allows FABLECORE function types to be, as usual, contravariant in the labels of their arguments and covariant in their return type.

Theorem (Expressiveness of static information flow)**.** *Given the policy π of Figure 4, a lattice $\mathscr{L}$ equivalent to the function lub of π, and a Core-ML derivation $\mathscr{D}$ such that, $\mathscr{D} = (\mathscr{L};\Gamma \vdash_{ML} \mathsf{e} : \mathsf{t})$; then $[\![\Gamma]\!];\cdot;\cdot \vdash \pi; [\![\mathscr{D}]\!] : [\![\mathsf{t}]\!]$.*

Proofs of both theorems can be found in our technical report [25].

```
letpol flow λsrc:lab.λdest:lab.
        let f = if oracle src dest then
                   FLOW(src,dest)
               else NOFLOW in
             (f, <unit{f}>())

letpol low Λα.λx:α.let l = oracle_low() in (l,<α{l}>x)

letpol sub Λα.φsrc,dest.λcap:unit{FLOW(src,dest)}.
                       λx:α{src}.<α{dest}>x

letpol app Λα.Λβ.φl.λf:(α →β){l}.φm.λx:α{m}.
             <β{JOIN(l, m)}> (<α →β>f) (<α> x)

Λα.λlb:lab.λb:bool{lb}.λlx:lab.λx:α{lx}.λly:lab.λy: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 5. Enforcing a dynamic information flow policy.

3.3 Dynamic Information Flow

Realistic information flow policies are rarely as simple as that of the previous example. 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 [18] or some other higher-level policy language, such as RT [26]. Figure 5 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 [33].

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:**lab**) → (dest:**lab**) → (l:**lab** × **unit**{l}).
If the oracle permits the flow, the flow function returns a capability similar to that provided by the login function of Figure 2. The sub function takes this capability as its first argument as proof that type α\{src\} may be coerced to type α\{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 **lab**∼fx to **lab**∼FLOW(lx,lxy) and the type of capx to be refined to **unit**{FLOW(lx,lxy)}, and similarly for capy. The remainder of the program is similar to the static case, but requires more uses of

```
letpol flow λsrc:lab.λdest: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, <unit{f}>())

letpol aes_enc φsrc,dest.λcap:unit{ENC(src,dest)}.
                    λdata:bytes{src}.λkey:bytes{dest}.
                        <bytes{dest}> ... key ... data
```

Figure 6. A trusted declassifier, explicit in the policy.

subsumption since less is known statically about the labels. The type inferred for this program is:

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

3.4 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 6 shows how a simple, but practical, downgrading policy in the form of a trusted declassifier [13] 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 ⊑ 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.

4. FABLE: Embedding FABLECORE in a Substructural Type System

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 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 3 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. We also show how *affine* types (values with an affine type may be used at most once in the program) can be used to encode security automata to enforce arbitrary safety properties, in a manner similar to Walker [27].

4.1 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:

```
letpol update Λα.φl.λx:α ref{l}.λy:α.
               <unit{C(LOW,EFFECT(l))}>(<α ref>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 program will type check:

```
φl.λx:int ref{l}. let _ = update [int] x 0 in 0
```

This function has the type l.x:**int ref**{l} →**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. Since values of relevant type must be used at least once, relevant types can be used to solve this problem.

4.2 Adding Relevant Types to FABLECORE.

FABLE augments FABLECORE with support for *relevant* labels. A relevant label literal is of the form $¡C(\vec{e})$ and, as previously, only the policy may associate such labels with program terms. A relevant label can be given the type ¡**lab**, a *relevant type*. An expression e with type, say, $t\{¡C\}$ also has relevant type. The type system ensures that expressions with relevant type are always passed, at least once, to a policy function, where they can be consumed using a relabeling. With this machinery in hand, the policy can ensure that a term that is generated as a witness to a side-effect (with a type like **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 product); or, it must be passed to the policy which can decide how to propagate the effect information.

Figure 7 shows the main elements of the FABLE language, presented, where possible, as an extension of FABLECORE. The top part of the figure shows the syntax extensions. First, we extend expressions e to include relevant label literals $¡C(\vec{e})$. Next, abstractions in FABLE now allow abstraction over both relevant and normal phantom labels. For example, in φ¡l.λx:t{l}.e, l is a relevant phantom label variable. As the type system extends subtyping to allow an irrelevant type to be treated as a relevant one, it would be acceptable to pass as an argument to this function a term with a type such as t{HIGH}, or t{¡HIGH}. In contrast, it would be illegal to pass a term with type t{¡HIGH} to the function φm.λy:t{m}.e, since this requires treating a relevant type as an irrelevant type, thereby allowing the program to fail to discharge a relevant assumption. We also include memory locations x_ℓ as values, and the standard dereferencing and assignment constructs. Note that for simplicity we do not model dynamic allocation; accounting for allocation effects poses no additional technical challenge.

The type language of FABLE now includes relevant types and reference types. For any type t, $¡t$ is a relevant type. In addition, a type such as $t\{e\}$ is a relevant type if e is a label with relevant type. The language of type kinds includes an additional kind R, which is the kind of relevant types. As previously, U stands for unlabeled types, and M for types that may be labeled. Kind R includes both types such as ¡**lab** as well as t{¡HIGH}, i.e., both unlabeled and labeled types. This induces a convenient sub-kinding relation U < M < R. Finally, the typing environment Ω includes a R component that records the set of relevant assumptions in Ω.Γ that must be discharged by an expression.

The remainder of Figure 7 is divided into three sections showing selected judgments from the semantics of FABLE. Space constraints preclude a full presentation here; the full semantics can be found in our companion technical report [25].

The judgment $R = R_1 \oplus R_2$ is used to split relevant assumptions when typing the components of an expression. Relevant type systems permit contraction and permutation of contexts but rule out weakening. The rules (X-L), (X-R) and (X-LR) encode contraction by ensuring that when a relevant context is split, every assumption in R is included in R_1 or R_2 *or both*. Traditional presentations of substructural type systems (for instance, in Walker's tutorial [28]) split the context Γ. However, in our case this can be problematic due to type-level dependencies. For instance, given $\Gamma = y : ¡\textbf{lab}, x : ¡t\{y\}$, if we split this as $\Gamma = (\Gamma_1 = x : t\{y\}) \oplus (\Gamma_2 = y : ¡\textbf{lab})$, then we are left with an ill-formed environment Γ_1 since the label name y appears in the type of x, but y is free in Γ_1. To avoid such complications, we maintain a separate context R of relevant assumptions in Γ and never split Γ itself.

The judgment $\Omega \vdash_{\vec{\varepsilon}} e : t$ types expressions similarly to the corresponding judgment in FABLECORE, but must track relevant assumptions. The first rule (T-SUBR) states that any type t may be treated as a relevant type $¡t$. The rule (T-VAR) is only applicable when $(\Omega.R = \emptyset)$; this ensures that weakening of R is not used to discharge relevant assumptions. Likewise, (T-RVAR) ensures that the usage of a relevant variable x discharges only a single relevant assumption, namely the assumption for the variable x. These three rules are standard for a relevant type system. We also include (T-VARφ) that allows a φ-bound variable to be used within type-level terms; note, that we do not impose any restriction on relevant assumptions for type-level terms since these have no runtime significance.

The rule (T-ABS) follows the same structure as the corresponding rule for FABLECORE. As previously, the first three premises requires the bound names to be fresh, and for all free phantom variables to be mentioned in the list of φ-bound variables. However, here, the φ-bound variables may include some relevant variables $¡\vec{z}$. When checking the ascribed type t_1 and the body of the abstraction, we must record the relevant variables $\vec{z}$ as having type ¡**lab** in the environment. Additionally, when checking the body expression e, if the type t_1 is a relevant type, then we must record $x : t_1$ as a relevant assumption and ensure that e discharges this assumption. This is achieved by the construction of R' in the side-condition of the last premise. Finally, as is standard in this setting, if a function is typed in a context with relevant assumptions $(\Omega.R \neq \emptyset)$ than we must give the function itself a relevant type. If we failed to do this, then a program could use a relevant assumption in an abstraction and then discard the abstraction, thereby violating the desired invariant. The corresponding elimination rule for (T-ABS) is (T-APP). This rule is identical to the FABLECORE case, except that (T-APP) splits the set of relevant assumptions so that e_1 is checked using R_1 and e_2 using R_2. The unification judgment $\vec{b};\Omega \vdash t_1 \leq t_2 : \sigma$ is similar to FABLECORE, except that we now must make sure that a relevant type/expression in t_1 is only unified with a corresponding relevant variable in t_2.

When typing assignments, the first premise of (T-ASN) ensures that only unlabeled references can be assigned to; as usual, assignment to labeled references must be mediated by the policy. The last premise of (T-ASN) ensures that only irrelevant values can escape into the heap. It might be possible to permit relevant values to be stored in the heap, but this complicates the system greatly. In particular, one must ensure that every assignment into a heap cell that contains a relevant value must be correlated with a corresponding dereference of that heap cell. This could be useful in a

Additional syntactic forms for FABLE.

e	$::=$	$\ldots \mid ¡C(\vec{e}) \mid \varphi\vec{b}.\lambda x:t.e \mid x_\ell \mid !e \mid e_1 := e_2$	relevant labels, abstraction, locations, deref, assignment
b	$::=$	$x \mid ¡x$	φ-bound names may or may not be relevant
t	$::=$	$\ldots \mid ¡t \mid t\,\mathbf{ref}$	relevant type; reference
κ	$::=$	$\ldots \mid \mathsf{R}$	relevant kind
R	$::=$	$\cdot \mid x:t \mid R_1,R_2$	relevant assumption context
Ω	$::=$	$\Gamma;\pi;A;R$	Typing context includes relevant assumptions

$R = R_1 \oplus R_2$ — Splitting of relevant assumptions

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

$\Omega \vdash_{\varepsilon} e : t$ — Expression e has type t in environment Ω

$$\frac{\Omega \vdash_{\varepsilon} e:t}{\Omega \vdash_{\varepsilon} e: ¡t}\;(\text{T-SUBR}) \qquad \frac{x:t \in \Omega.\Gamma \qquad \Omega \vdash t :: \mathsf{M}}{\Omega[R=\emptyset] \vdash_{\varepsilon} x:t}\;(\text{T-VAR}) \qquad \frac{x:t \in \Omega.\Gamma}{\Omega[R = x:t] \vdash_{\varepsilon} x:t}\;(\text{T-RVAR}) \qquad \frac{\varphi x:t \in \Omega.\Gamma}{\Omega \vdash_{\varphi} x:t}\;(\text{T-VAR}\varphi)$$

$$\frac{\begin{array}{c}\Omega.\Gamma(x)\ \textit{undefined} \qquad \vec{b} = \vec{y}, ¡\vec{z} \qquad \vec{y},\vec{z} = (FV(t_1) \setminus \mathrm{dom}(\Omega.\Gamma)) \qquad \Omega' = \Omega[\varphi\vec{y}:\mathbf{lab}, \varphi\vec{z}: ¡\mathbf{lab}] \\ \Omega' \vdash t_1 :: \kappa \qquad \Omega'[R=R'][x:t_1] \vdash_{\varepsilon} e:t_2 \ \ \textbf{where}\ R' = \Omega.R,x:t_1\ \textbf{if}\ \kappa = \mathsf{R}\ \textbf{else}\ R' = \Omega.R\end{array}}{\Omega \vdash_{\varepsilon} \varphi\vec{b}.\lambda x:t_1.e:t \ \ \textbf{where}\ t = (\vec{b}.x:t_1) \to t_2\ \textbf{if}\ \Omega.R = \emptyset\ \textbf{else}\ t = ¡(\vec{b}.x:t_1) \to t_2}\;(\text{T-ABS})$$

$$\frac{\begin{array}{c}\Omega[R=R_1] \vdash_{\cdot} e_1 : t\,\mathbf{ref} \\ \Omega[R=R_2] \vdash_{\cdot} e_2 : t \qquad \Omega \vdash t :: \mathsf{M}\end{array}}{\Omega[R = R_1 \oplus R_2] \vdash_{\cdot} e_1 := e_2 : \mathit{unit}}\;(\text{T-ASN})$$

$$\frac{\begin{array}{c}\Omega[R=R_1] \vdash_{\varepsilon} e_1 : ¡(\vec{b}.x:t_1) \to t_2 \qquad \Omega[R=R_2] \vdash_{\varepsilon} e_2 : t_1' \\ \vec{b};\Omega.A \vdash t_1' \leq t_1 : \sigma \qquad \sigma' = \sigma,(x \mapsto e_2)\end{array}}{\Omega[R = R_1 \oplus R_2] \vdash_{\varepsilon} e_1 e_2 : \sigma'(t_2)}\;(\text{T-APP}) \qquad \frac{\Omega \vdash_{\cdot}^{ESC} e : t\{\vec{e'}\}}{\Omega \vdash_{\varepsilon} \langle t \rangle e : t}\;(\text{T-PFX}) \qquad \frac{\begin{array}{c}\mathit{relevant}(\Omega.\Gamma) = R_1 \oplus R_2 \\ \exists i.\Omega[R=R_i] \vdash_{\varepsilon} e:t\end{array}}{\Omega \vdash_{\varepsilon}^{ESC} e:t}\;(\text{ESC})$$

$\Gamma \vdash t :: \kappa$ — Type t has kind κ in environment Γ (K-UNIT, K-ABS, K-UNIV, K-VAR unchanged)

$$\frac{\Omega \vdash t :: \kappa \qquad \kappa \leq \kappa'}{\Omega \vdash t :: \kappa'}\;(\text{K-1}) \qquad \frac{\Omega \vdash t :: \kappa}{\Omega \vdash ¡t :: \mathsf{R}}\;(\text{K-2}) \qquad \frac{\Omega \vdash t :: \kappa \qquad \Omega \vdash_{\varphi}^{ESC} e : ¡\mathbf{lab}}{\Omega \vdash t\{e\} : \mathsf{R}}\;(\text{K-3}) \qquad \frac{\Omega \vdash t :: \kappa \qquad \Omega \vdash_{\varphi}^{ESC} e : \mathbf{lab}}{\Omega \vdash t\{e\} : \kappa \sqcup \mathsf{M}}\;(\text{K-4})$$

Figure 7. Semantics of FABLE—selected judgments. (The full semantics in Appendix A use a modified judgment in order to ensure that type-level expressions are effectless. The structure of each rule shown here is, however, unchanged.)

system that attempts to track, say, resource usage; however, for our purposes, the additional complication does not provide any obvious benefit. We leave the investigation of relaxing this restriction to future work.

The (T-PFX) rule shows how we allow *policy* terms to be more liberal with the way in which they manipulate relevant assumptions, since our objective is only to ensure relevant terms created in the policy are eventually destructed by the policy. The first premise of (T-PFX) uses the judgment $\Omega \vdash_{\varepsilon}^{ESC} e : t$, defined by the rule (ESC). In the first premise of (ESC), $\mathit{relevant}(\Omega.\Gamma)$ stands for *all* assumptions $x_i : t_i$ in $\Omega.\Gamma$ where t_i is a relevant type. (ESC) lifts the restriction on relevant assumptions by allowing e to be typed in a context with as many or as few relevant assumptions as necessary. In practice, implicitly lifting all restrictions on relevant assumptions within a relabeling operation is likely to be undesirable. Errors can easily be made by policy programmers that result in relevant arguments being dropped when they should not. To reduce the likelihood of such errors, our implementation requires the policy designer to explicitly declare which relevant assumptions are to be dropped.

Finally, Figure 7 shows some of the kinding judgments. (K-1) encodes the sub-kinding relation; (K-2) gives a relevant type the R kind; (K-3) forces a type labeled with a relevant label to be a relevant type; and (K-4) allows a type t labeled with an irrelevant label to have M kind, as long as t is not itself a relevant type. Note that (K-3) and (K-4) use the (ESC) judgment to check the label terms; as with (T-VARφ) we do not impose any restriction on relevant assumptions when checking terms that appear in types.

4.3 Soundness of FABLE

We state the soundness theorems of FABLE with respect to a typed reduction relation $\pi \vdash (M,e) \leadsto (M',e')$. This relation is a com-

```
letpol update Λα::M.φ¡lλx:α ref{l}.λy:α.
                <unit{¡C(LOW, EFFECT(l))}>(<α ref>x) := y

letpol combine λl:¡lab.λm:¡lab. match l m with
                ¡C(l1,fx1),¡C(l2,fx2) →¡C(lub l1 l2, glb fx1 fx2)

letpol seq Λα::U.Λβ::U.φ¡l.λe1:unit→α{l}.φ¡m.λe2:unit→β{m}.
         let x = e1 () in
           <β{combine l m}>(e2 ())
```

Figure 8. Tracking effects using FABLE.

pletely standard extension of the memory-less relation $\pi \vdash e \leadsto e'$ defined in Figure 3.[3] Proving type erasure is also straightforward as none of the reduction rules ever inspect the types.

Our well-formedness condition requires $\mathrm{dom}(\Omega.\Gamma) = \mathrm{dom}(M)$; for each location in the store M to be given a reference type in $\Omega.\Gamma$; and for all values in the store to be well-typed closed terms. This is the content of the judgment $\Omega \models M$. The progress theorem is standard. The preservation theorem, as indicated in Section 2, equates types that are related by the type reduction relation.

Theorem (Progress)**.** *Given $\Omega \equiv \Gamma;\cdot;\cdot;\cdot$ and memory M such that $\Omega \models M$, and $\pi;e$ such that $\Omega \vdash \pi;e:t$, then either $\pi \vdash (M,e) \leadsto (M',e')$ for some M',e' or e is a value.*

Theorem (Preservation)**.** *Given $\Omega \equiv \Gamma;\cdot;\cdot;\cdot$ and memory M such that $\Omega \models M$, and $\pi;e$ such that $\Omega \vdash \pi;e:t$, then $\pi \vdash (M,e) \leadsto (M',e')$ implies $\Omega \models M'$ and $\Omega \vdash \pi;e':t$.*

4.4 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 includes a function update to assign to a labeled reference; it returns a labeled unit as a witness to the side-effect. We use a relevant composite label to produce a witness of type **unit**{¡C(LOW,EFFECT(l))} so that the program cannot ignore the returned value. Figure 8 shows the full policy.

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 relevant label. The body of seq executes the first computation to produce the value x, which has a relevant type. The final term in the body of seq is a relabeling expression, and according to (T-PFX), it is free to use or ignore as many relevant assumptions as necessary. Thus, seq consumes the relevant assumption for x, and executes the computation e2 and returns the result. Importantly, for effects to be tracked properly, seq must reflect the effects of both e1 and e2 in the label of the value it returns. 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. Our technical report contains the details of how this basic idea can be used to encode an information flow policy in the presence of side-effects.

4.5 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 [27].

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 [3]. Figure 9 shows an encoding of this policy in FABLE. The notation ?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 7—we simply disable the (X-LR) rule for affine assumptions, and permit affine assumptions to be dropped at will. Again, our technical report contains the details.

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∼e. Since state∼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∼e → α→β; 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 9 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

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

letpol 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, ?(t, <?unit{t}>())

letpol 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 9. A security automaton in FABLE using affine types.

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∼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.

5. Related Work

FABLE is most closely related to Walker's type system for expressive security policies [27]. Both systems use a limited form of dependent types—whereas we allow arbitrary label expressions to appear in types, in Walker's language, type indices are uninterpreted predicates. Not interpreting the indices allows type-checking in Walker's language to be decidable, but limits its expressiveness. In Walker's basic system, establishing the identity of the current state of the automaton must always be accomplished by means of a run-

[3] It is important to note that the type reduction relation in FABLE remains $\pi \vdash t \rightsquigarrow t'$; i.e., side effects cannot occur in expressions that appear in types.

time check. FABLE is able to enforce some policies (e.g., Figure 4) without any runtime checks. Walker shows how to perform certain optimizations to eliminate runtime checks by augmenting the type system with policy-specific axioms (which can, if the policy designer is not careful, render type checking undecidable). FABLE essentially permits programmers to encode such extensions in the enforcement policy. Programmers embed custom label expressions in types and define the policy so that statically-enforceable properties can be handled purely by the type checker, if possible, relying on dynamic checks otherwise. Finally, Walker's language uses a single capability to track the global state of the program relevant to a security policy. This makes it difficult to combine security policies and reason in isolation about each. We allow security labels to be associated with sub-expressions in the program making our approach more compositional. Furthermore, labelings allow precise dataflow properties to be associated with expressions.

Dependent types have been used in several other contexts. Xi and Pfenning [31] use linear integer inequalities as type indices to show that array access are within bounds. In subsequent work, Xi's ATS system [30] has shown how type indices drawn from a custom index language (including, say, our label language) can be integrated into a dependent typing system. However, indices in ATS have no runtime representation as they are drawn from a language intentionally kept separate from program expressions. Thus, while some of the statically enforceable policies shown in this paper can be encoded in ATS, there does not appear to be a uniform way of encoding policies that additionally require dynamic checks.

Cayenne [2] is a pure language in which the type and term languages coincide. This results in an extremely powerful system for which, like FABLE, type-checking can be undecidable. Cayenne focuses on static verification, while in FABLE static and dynamic checks can be mixed. Cayenne also does not support state tracking.

Millstein et al [5, 1] have proposed *user-defined type extensions*, which allow the programmer to introduce custom type qualifiers into C or Java programs by inserting new rules into an extensible type checker. Importantly, they allow programmers to specify data invariants that the qualifiers are intended to represent and, in several cases, they are able to automatically verify that these invariants are correctly enforced by the type rules. The invariants expressible in their system are much simpler than the invariants, such as noninterference, that we would like to show for our security policies. Nevertheless, it remains one of our key objectives to develop a framework in which the verification of correctness properties for enforcement policies defined in FABLE can be (partially) automated. Marino et al. [17] have proposed to partially mechanize the proof of correctness of user-defined type extensions by relying on a proof assistant. We expect that this idea will be useful for proving the correctness of FABLE policies too.

Li and Zdancewic show how to encode information flow policies in Haskell [16]. 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. They only show an encoding of an information flow policy.

Zheng and Myers also use dependent types to associate security labels with sensitive data [33]. Their system, implemented in Jif [6], is not customizable—it enforces information flow policies. Figure 5 shows an encoding of a core subset of their system in FABLE.

Inasmuch as FABLE's design ensures complete mediation (i.e., the program must use the policy in order to manipulate labeled data), it is related to work by Zhang et al. [32] and Fraser et al. [10] on enforcing complete mediation in access control systems.

P	$::=$	$\pi; e$	programs : policy and program term
π	$::=$	$\cdot \mid \textbf{letpol}\ \mathtt{x}\ v \mid \pi_1, \pi_2$	policy: a sequence of bindings
e	$::=$	$() \mid x \mid \mathtt{x} \mid \langle t \rangle e$	unit, variables, relabeling
	$\mid$	$C(\vec{e}) \mid ¡C(\vec{e})$	label, relevant label
	$\mid$	$\varphi\vec{b}.\lambda x : t.e$	abstraction with phantom labels $\vec{b}$
	$\mid$	$\textbf{match}\ e\ \textbf{with}\ \vec{b}_i.p_i \rightarrow e_i$	pattern matching
	$\mid$	$\textbf{fix}\ x.v \mid e_1 e_2$	recursion, application
	$\mid$	$\Lambda \nu :: \kappa.e \mid e[t]$	type abstraction and application
	$\mid$	$x_\ell \mid e_1 := e_2 \mid !e$	locations, assignment, deref
u	$::=$	$() \mid C(\vec{v}) \mid ¡C(\vec{v}) \mid \varphi\vec{b}.\lambda x : t.e \mid \Lambda \alpha :: \kappa.e \mid x \mid x_\ell$	pre-values
v	$::=$	$u \mid \langle t \rangle u$	values: bare or labeled pre-values
b	$::=$	$x \mid ¡x$	binding normal and relevant names
p	$::=$	$x \mid C(\vec{p}) \mid ¡C(\vec{p})$	patterns
σ	$::=$	$\cdot \mid (x \mapsto e) \mid \sigma_1, \sigma_2$	substitutions
t	$::=$	$\textbf{unit} \mid \nu \mid \forall \nu :: \kappa.t_2$	unit; type var; universal type
	$\mid$	$\vec{b}.x : t_1 \xrightarrow{\psi} t_2$	function type, with effect annotation
	$\mid$	$\textbf{lab} \mid \textbf{lab} \sim e \mid t\{e\}$	label types; labeled type
	$\mid$	$¡t \mid t\,\textbf{ref}$	relevant type; reference
ν	$::=$	$\alpha \mid \beta \mid \ldots$	type variables
ψ	$::=$	$0 \mid 1$	effects
κ	$::=$	$\mathrm{U} \mid \mathrm{M} \mid \mathrm{R}$	unlabeled, maybe-labeled and relevant kinds

Figure 10. Full syntax of FABLE.

6. Conclusions and Future Work

This paper has described FABLE, a core formalism for expressing and properly enforcing a wide range of security policies.

We have implemented a prototype interpreter for FABLE. Ultimately we plan to develop a high-level language and a typed intermediate language based on FABLE which targets web applications. To ease high-level programming, we are currently exploring the possibility of augmenting the policy language with rules for rewriting programs automatically; e.g., for programs using one of the example information flow policies, we would automatically rewrite function calls to use the policy function `app`. We are also looking at how to support the semi-automated verification of enforcement policies with respect to high-level security objectives. The first step is to formalize a language in which policy correctness properties can be stated. We can then automatically translate FABLE policies and security objectives to declarations and goals in one of several interactive theorem proving frameworks. Our experience with a manual proof of noninterference for the static information flow example suggests that, equipped with a library of lemmas from the proof of soundness of FABLE, the burden on the policy analyst to discharge security goals will be somewhat reduced.

A. Soundness of FABLE

Definition 1 (Well-formed environment). *$\Omega = \Gamma; \pi; A; R$ is well-formed if and only if*

(i.) $\Gamma = \Gamma_1, x : t, \Gamma_2 \quad \Rightarrow$
$\exists \kappa. \Omega[\Gamma = \Gamma_1, \Gamma_2] \vdash t :: \kappa$

(ii.) $\Gamma = \Gamma_1, \varphi x : t, \Gamma_2 \quad \Rightarrow$
$t \in \{\textbf{lab}, ¡\textbf{lab}\}$

(iii.) $\Gamma = \Gamma_1, \mathtt{x} : t, \Gamma_2 \quad \Rightarrow$
$\textbf{letpol}\ \mathtt{x}\ v \in \pi \ \wedge\ \Omega \vdash v : t$

(iv.) *All names bound in Γ are distinct*

(v.) $A = A_1, e_1 \leq e_2, A_2 \quad \Rightarrow$
$\Omega \vdash e_1 : ¡\textbf{lab} \ \wedge\ \Omega \vdash e_2 : ¡\textbf{lab}$

(vi.) $R = R_1, x : t, R_2 \Rightarrow \Gamma = \Gamma_1, x : t, \Gamma_2$

Additional syntactic forms used in judgments

Γ	$::=$	$x:t \mid \mathtt{x}:t \mid \varphi x:t \mid \nu :: \kappa \mid \Gamma_1,\Gamma_2$	variable bindings, policy global names, phantom variables, type variables
A	$::=$	$\cdot \mid e_1 \leq e_2 \mid A_1,A_2$	Pattern matching assumptions
R	$::=$	$\cdot \mid x:t \mid R_1,R_2$	Context to track relevant assumptions
Ω	$::=$	$\Gamma;\pi;A;R;\psi$	Typing context quintuple of bindings, policy, pattern assumptions, relevant assumptions and effect
ε	$::=$	$\cdot \mid \varphi$	Index used to indicate type-checking phase

$\Omega \vdash P : t \quad ; \quad \Omega \vdash \pi : \Omega'$ — Top-level judgments for typing programs and policies

$$\frac{\begin{array}{c}\Omega[R=\cdot] \vdash \pi : \Omega' \\ \Omega'[R=\Omega.R] \vdash e : t\end{array}}{\Omega \vdash \pi; e : t}\ (\text{T-PROG}) \qquad \frac{\begin{array}{c}\Omega.\Gamma(\mathtt{x})\ \mathit{undefined} \quad \Omega[\psi=0] \vdash_{\cdot} v : t \\ \Omega[\mathtt{x}:t][\textbf{letpol}\ \mathtt{x}\ v] \vdash \pi' : \Omega'\end{array}}{\Omega \vdash \textbf{letpol}\ \mathtt{x}\ v, pi' : \Omega'}\ (\text{T-LETPOL}) \qquad \Omega \vdash \cdot : \Omega\ (\text{T-EMPTY})$$

$\Omega \vdash_{\varepsilon} e : t$ — Expression e has type t in environment Ω

$$\frac{x:t \in \Omega.\Gamma \quad \Omega \vdash t :: \mathsf{M}}{\Omega[R=\emptyset] \vdash_{\varepsilon} x : t}\ (\text{T-VAR}) \qquad \frac{x:t \in \Omega.\Gamma}{\Omega[R=x:t] \vdash_{\varepsilon} x : t}\ (\text{T-RVAR}) \qquad \frac{\varphi x:t \in \Omega.\Gamma}{\Omega \vdash_{\varphi} x : t}\ (\text{T-VAR}\varphi) \qquad \frac{\Omega.\Gamma = \Gamma_1,\mathtt{x}:t,\Gamma_2}{\Omega[R=\emptyset] \vdash_{\varepsilon} \mathtt{x} : t}\ (\text{T-PVAR})$$

$$\frac{\Omega \vdash_{\varepsilon} e : t}{\Omega \vdash_{\varepsilon} e : \text{¡}t}\ (\text{T-SUBR}) \qquad \frac{\Omega.R = \emptyset}{\Omega \vdash_{\varepsilon} () : \textbf{unit}}\ (\text{T-UNIT}) \qquad \frac{\Omega.R = R_1 \oplus \ldots \oplus R_n \quad \Omega.\psi = 0 \quad e \in \{C(\vec{e}), \text{¡}C(\vec{e})\} \ \ \forall i.\Omega[R=R_i] \vdash_{\varepsilon} e_i : \text{¡}\textbf{lab}}{\Omega \vdash_{\varepsilon} e : \textbf{lab} \sim e}\ (\text{T-L1})$$

$$\frac{\Omega.R = R_1 \oplus \ldots \oplus R_n \quad \forall i.\Omega[R=R_i] \vdash_{\varepsilon} e_i : \textbf{lab}}{\Omega \vdash_{\varepsilon} C(\vec{e}) : \textbf{lab}}\ (\text{T-L2}) \qquad \frac{\Omega \vdash_{\varepsilon} e : \textbf{lab} \sim e}{\Omega \vdash_{\varepsilon} e : \text{¡}\textbf{lab}}\ (\text{T-L3}) \qquad \frac{\Omega \vdash_{\varepsilon} e : \textbf{lab} \sim e' \quad \Omega \vdash_{\varepsilon} e' : \textbf{lab}}{\Omega \vdash_{\varepsilon} e : \textbf{lab}}\ (\text{T-L4})$$

$$\frac{\Omega.\psi = 0 \quad \Omega \vdash_{\varepsilon} e : \text{¡}\textbf{lab}}{\Omega \vdash_{\varepsilon} e : \textbf{lab} \sim e}\ (\text{T-L5}) \qquad \frac{\Omega \vdash t :: \mathsf{M} \quad \Omega[f:t] \vdash_{\varepsilon} v : t}{\Omega \vdash_{\varepsilon} \textbf{fix}\ f.v : t}\ (\text{T-FIX}) \qquad \frac{\Omega.\Gamma(\alpha)\ \mathit{undefined} \quad \Omega[\alpha{::}\kappa] \vdash_{\varepsilon} e : t \quad t_u = \forall \alpha{::}\kappa.t}{\Omega \vdash_{\varepsilon} \Lambda \alpha{::}\kappa.e : t\ \ \textbf{where}\ t = t_u\ \textbf{if}\ \Omega.R = \cdot\ \textbf{else}\ t = \text{¡}t_u}\ (\text{T-TAB})$$

$$\frac{\begin{array}{c}\Omega.\Gamma(x)\ \mathit{undefined} \quad \vec{b} = \vec{y}, \text{¡}\vec{z} \quad \vec{y},\vec{z} = (FV(t_1) \setminus \mathrm{dom}(\Omega.\Gamma)) \quad \Omega' = \Omega[\varphi\vec{y} : \textbf{lab}, \varphi\vec{z} : \text{¡}\textbf{lab}] \\ \Omega' \vdash t_1 :: \kappa \quad \Omega'[R=R'][x:t_1][\psi=\psi'] \vdash_{\varepsilon} e : t_2\ \ \textbf{where}\ R' = \Omega.R, x:t_1\ \textbf{if}\ \kappa = \mathsf{R}\ \textbf{else}\ R' = \Omega.R\end{array}}{\Omega \vdash_{\varepsilon} \varphi\vec{b}.\lambda x:t_1.e : t\ \ \textbf{where}\ t = (\vec{b}.x:t_1) \xrightarrow{\psi'} t_2\ \textbf{if}\ \Omega.R = \emptyset\ \textbf{else}\ t = \text{¡}(\vec{b}.x:t_1) \xrightarrow{\psi'} t_2}\ (\text{T-ABS}) \qquad \frac{\begin{array}{c}\Omega \vdash t :: \kappa \\ \Omega \vdash_{\varepsilon} e : \text{¡}\forall \alpha{::}\kappa.t'\end{array}}{\Omega \vdash_{\varepsilon} e[t] : (\alpha \mapsto t)t'}\ (\text{T-TAP})$$

$$\frac{\begin{array}{c}\Omega[R=R_1] \vdash_{\varepsilon} e_1 : \text{¡}(\vec{b}.x:t_1) \xrightarrow{\psi_1} t_2 \quad \Omega[R=R_2] \vdash_{\varepsilon} e_2 : t_1' \\ \vec{b};\Omega \vdash t_1' \leq t_1 : \sigma \quad \sigma' = \sigma, (x \mapsto e_2) \quad \psi_1 \leq \Omega.\psi\end{array}}{\Omega[R=R_1 \oplus R_2] \vdash_{\varepsilon} e_1 e_2 : \sigma'(t_2)}\ (\text{T-APP}) \qquad \frac{\begin{array}{c}\Omega.\psi = 1 \quad \Omega[R=R_1] \vdash_{\cdot} e_1 : t\,\textbf{ref} \\ \Omega[R=R_2] \vdash e_2 : t \quad \Omega \vdash t :: \mathsf{M}\end{array}}{\Omega[R=R_1 \oplus R_2] \vdash_{\cdot} e_1 := e_2 : \textbf{unit}}\ (\text{T-ASN}) \qquad \frac{\Omega[\psi=0] \vdash_{\varepsilon} e : t}{\Omega \vdash_{\varepsilon} e : t}\ (\text{T-FX})$$

$$\frac{\begin{array}{c}\Omega.R = R_1 \oplus R_2 \quad \vec{b_i} = \vec{x_i}, \text{¡}\vec{y_i} \quad (\vec{b_n}, p_n) = (\text{¡}x_{def}, x_{def}) \quad \vec{b_i} \cap \vec{b_j} = \emptyset \quad \vec{b_i} = FV(p_i) \setminus \mathrm{dom}(\Omega.\Gamma) \\ \Omega[R=R_1] \vdash_{\varepsilon} e : \text{¡}\textbf{lab} \quad R_i = \Omega.R, \vec{y_i} : \text{¡}\textbf{lab} \quad \Omega' = \Omega[R=R_i][\vec{x_i} : \textbf{lab}, \vec{y_i} : \text{¡}\textbf{lab}] \quad \Omega' \vdash_{\varepsilon} p_i : \text{¡}\textbf{lab}; 0 \quad \Omega'[e \leq p_i] \vdash_{\varepsilon} e_i : t\end{array}}{\Omega \vdash_{\varepsilon} \textbf{match}\ e\ \textbf{with}\ \vec{b}_1.p_1 \to e_1 \ldots \vec{b}_n.p_n \to e_n : t}\ (\text{T-MATCH})$$

$$\frac{\begin{array}{c}\Omega.\psi = 1 \\ \Omega \vdash_{\cdot} e : t\,\textbf{ref}\end{array}}{\Omega \vdash_{\cdot} !e : t}\ (\text{T-DRF}) \qquad \frac{\Omega \vdash_{\cdot}^{ESC} e : t\{\vec{e'}\}}{\Omega \vdash_{\varepsilon} \langle t \rangle e : t}\ (\text{T-PFX}) \qquad \frac{\begin{array}{c}\Omega \vdash_{\cdot}^{ESC} e : t \\ \forall i.\Omega[\psi=0] \vdash_{\varphi}^{ESC} e_{l_i} : \text{¡}\textbf{lab}\end{array}}{\Omega \vdash_{\varepsilon} \langle t\{\vec{e_l}\} \rangle e : t\{\vec{e_l}\}}\ (\text{T-SFX}) \qquad \frac{\begin{array}{c}\Omega \vdash_{\varepsilon} e : t \\ \Omega.\pi \vdash t \rightsquigarrow t' \quad \Omega \vdash t' :: \kappa\end{array}}{\Omega \vdash_{\varepsilon} e : t'; \psi}\ (\text{T-EVT})$$

$R = R_1 \oplus R_2$ — Splitting of relevant assumptions

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

Figure 11. Static semantics of FABLE (Part 1).

Definition 2 (Store typing)**.** *A store M is modeled by the environment Ω ($\Omega \models M$) if all of the following are true.*

- *(i)* $\mathit{dom}(\Omega.\Gamma) = \mathit{dom}(M)$
- *(ii)* $\Omega.\Gamma = \Gamma_1, x:t, \Gamma_2 \ \Rightarrow$ $\exists t', e.t \in \{t'\textbf{ref}, t'\textbf{ref}\{e\}\} \ \wedge\ \Omega \vdash t :: \mathsf{M} \ \wedge\ \Omega \vdash t' :: \mathsf{M}$
- *(iii)* $\mathit{range}(M) = \{v_1, \ldots, v_n\}$
- *(iv)* $\Gamma(x) \in \{t'\textbf{ref}, t'\textbf{ref}\{e\}\} \Rightarrow \Omega \vdash M(x) = t'$

Theorem 3 (Progress)**.** *Given $\Omega = \Gamma;\cdot;\cdot;\cdot$, Ω well-formed, $\Omega \vdash \pi; e : t$, and memory M such that $\Omega \models M$; then either $\exists e'.\pi \vdash e \rightsquigarrow e'$ or $\exists v.e = v$.*

Proof. By induction on the structure of the second premise of (T-PROG).

CASE (T-VAR): By assumption, $\mathrm{dom}(\Gamma) = \mathrm{dom}(M)$. Thus, at the second premise of (T-PROG), $\mathrm{dom}(\Gamma) = \mathrm{dom}(M) \cup \{\mathtt{x} \mid \textbf{letpol}\ x\ v \in \pi\}$. Thus, if (T-VAR) is applicable, then $x = x_\ell$ then x_ℓ is a value and is thus irreducible.

CASE (T-RVAR): Impossible; by assumption R is empty.

CASE (T-VARφ): Impossible; by construction, the second premise of (T-PROG) uses phase index $\varepsilon = \cdot$.

CASE (T-PVAR): By well-formedness of Ω, we have $\textbf{letpol}\ \mathtt{x}\ v \in \pi$ and $\pi \vdash \mathtt{x} \rightsquigarrow v$ by the redex-time reduction rule (E-POL).

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

$\Gamma \vdash t :: \kappa$ Type t has kind κ in environment Γ

$$\Omega \vdash \mathbf{unit} :: \mathrm{U}\ \text{(K-UNIT)} \qquad \frac{\alpha :: \kappa \in \Omega.\Gamma}{\Omega \vdash \alpha :: \kappa}\ \text{(K-VAR)} \qquad \frac{\begin{array}{c}\Omega.\Gamma(x,\vec{y})\ \textit{undefined} \qquad \Omega[\varphi\vec{y} : \mathbf{lab}] \vdash t_1 :: \kappa' \\ \Omega[\varphi\vec{y} : \mathbf{lab}, x : t_1] \vdash t_2 :: \kappa\end{array}}{\Omega \vdash (\vec{y}.x : t_1) \to t_2 :: \mathrm{U}}\ \text{(K-ABS)} \qquad \frac{\begin{array}{c}\Omega.\Gamma(\alpha)\ \textit{undefined} \\ \Omega[\alpha :: \kappa] \vdash t :: \kappa'\end{array}}{\Omega \vdash \forall \alpha :: \kappa.t :: \mathrm{U}}\ \text{(K-UNIV)}$$

$$\frac{\Omega \vdash t :: \mathrm{L}}{\Omega \vdash t :: \mathrm{R}}\ \text{(K-LR)} \qquad \frac{\Omega \vdash t :: \kappa \qquad \kappa \neq \mathrm{R}}{\Omega \vdash t :: \mathrm{M}}\ \text{(K-KA)} \qquad \frac{t \in \{\text{¡}\mathbf{lab}, \mathbf{lab}\sim e\}}{\Omega \vdash t : \mathrm{R}}\ \text{(K-RLAB)} \qquad \Omega \vdash \mathbf{lab} : \mathrm{U}\ \text{(K-LAB)} \qquad \frac{\Omega[\psi = 0] \vdash_{\varphi} e : \mathbf{lab}; 0}{\Omega \vdash \mathbf{lab}\sim e : \mathrm{U}}\ \text{(K-ULAB)}$$

$$\frac{\begin{array}{c}\Omega \vdash t :: \kappa \\ \Omega[\psi = 0] \vdash_{\varphi} e : \mathbf{lab}\end{array}}{\Omega \vdash t\{e\} : \kappa \sqcup \mathrm{L}}\ \text{(K-SEC)} \qquad \frac{\begin{array}{c}\Omega \vdash t :: \kappa \\ \Omega[\psi = 0] \vdash_{\varphi} e : \text{¡}\mathbf{lab}\end{array}}{\Omega \vdash t\{e\} : \mathrm{R}}\ \text{(K-RSEC)} \qquad \frac{\Omega \vdash t :: \kappa}{\Omega \vdash \text{¡}t :: \mathrm{R}}\ \text{(K-REL)} \qquad \frac{\begin{array}{c}\textit{relevant}(\Omega.\Gamma) = R_1 \oplus R_2 \\ \exists i.\Omega[R = R_i] \vdash_{\varepsilon} e : t\end{array}}{\Omega \vdash_{\varepsilon}^{ESC} e : t}\ \text{(ESC)}$$

$\vec{b_2}; \Omega \vdash t_1 \leq t_2 : \sigma \quad ; \quad \vec{b_1}; \vec{b_2}; \Omega \vdash e_1 \leq e_2 : \sigma$ Unification of types and expressions

$$\vec{b_2}; \Omega \vdash t \leq t : \cdot\ \text{(U-TID)} \qquad \vec{b_1}; \vec{b_2}; \Omega \vdash e \leq e : \cdot\ \text{(U-EID)} \qquad \frac{\cdot; \vec{b_2}; \Omega \vdash e_1 \leq e_2 : \sigma}{\vec{b_2}; \Omega \vdash \mathbf{lab}\sim e_1 \leq \mathbf{lab}\sim e_2 : \sigma}\ \text{(U-LAB)} \qquad \frac{\vec{b_2}; \Omega[\alpha :: \kappa] \vdash t_1 \leq (\beta \mapsto \alpha)t_2 : \sigma}{\vec{b_2}; \Omega \vdash \forall \alpha :: \kappa.t_1 \leq \forall \beta :: \kappa.t_2 : \sigma}\ \text{(U-UNI)}$$

$$\frac{\begin{array}{c}\sigma_0 = (y \mapsto x, \vec{b_y} \mapsto \vec{b_x}) \qquad \vec{b_2}; \Omega[\vec{b_x}] \vdash t_1 \leq \sigma_0 t_1' : \sigma_1 \\ \vec{b_2}; \Omega[\vec{b_x}, x : t_1] \vdash t_2 \leq \sigma_1 \sigma_0 t_2' : \sigma_2\end{array}}{\vec{b_2}; \Omega \vdash (\vec{b_x}.x : t_1) \to t_2 \leq (\vec{b_y}.y : t_1') \to t_2' : \sigma_1, \sigma_2}\ \text{(U-FUN)} \qquad \frac{\begin{array}{c}\vec{b_2}; \Omega \vdash t_1 \leq t_2 : \sigma \\ \cdot; \vec{b_2}; \Omega \vdash e_1 \leq \sigma(e_2) : \sigma'\end{array}}{\vec{b_2}; \Omega \vdash t_1\{e_1\} \leq t_2\{e_2\} : \sigma, \sigma'}\ \text{(U-SEC)} \qquad \frac{\begin{array}{c}\Omega.A = A_1, e_1 \leq e_1', A_2 \\ \vec{b_1}; \vec{b_2}; \Omega \vdash e_1' \leq e_2 : \sigma\end{array}}{\vec{b_1}; \vec{b_2}; \Omega \vdash e_1 \leq e_2 : \sigma}\ \text{(U-AS)}$$

$$\frac{\forall i.\sigma_i^* = (\sigma_i, \ldots, \sigma_{i-1}) \qquad \vec{b_1}; \vec{b_2}; \Omega \vdash \sigma_i^* e_i \leq \sigma_i^* e_i' : \sigma_i}{\vec{b_1}; \vec{b_2}; \Omega \vdash C(\vec{e}) \leq C(\vec{e'}) : \vec{\sigma}}\ \text{(U-CON)} \qquad \frac{\forall i.\sigma_i^* = (\sigma_i, \ldots, \sigma_{i-1}) \qquad \vec{b_1}; \vec{b_2}; \Omega \vdash \sigma_i^* e_i \leq \sigma_i^* e_i' : \sigma_i}{\vec{b_1}; \vec{b_2}; \Omega \vdash \text{¡}C(\vec{e}) \leq \text{¡}C(\vec{e'}) : \vec{\sigma}}\ \text{(U-RCON)}$$

$$\frac{x \in \vec{b_1} \qquad \sigma = x \mapsto v}{\vec{b_1}; \vec{b_2}; \Omega \vdash x \leq v : \sigma}\ \text{(U-VL)} \qquad \frac{(\{x, \text{¡}x\} \cap \vec{b_2} \neq \emptyset) \qquad \Omega, \vec{b_1} \vdash v : \mathbf{lab}; 0}{\vec{b_1}; \vec{b_2}; \Omega \vdash v \leq x : x \mapsto v}\ \text{(U-VR1)} \qquad \frac{\text{¡}x \in \vec{b_2} \qquad \Omega, \vec{b_1} \vdash v : \text{¡}\mathbf{lab}; 0}{\vec{b_1}; \vec{b_2}; \Omega \vdash v \leq x : x \mapsto v}\ \text{(U-VR2)}$$

Figure 12. Static semantics of FABLE (Part 2).

Additional syntactic forms used in judgments

$$\begin{array}{lll} M & ::= & (\ell, v) \mid M_1, M_2 \\ E & ::= & \bullet \mid \bullet\, e \mid v \bullet \mid \bullet\, [t] \mid C(\vec{v}, \bullet, \vec{e}) \mid \mathbf{match} \bullet \mathbf{with}\ \vec{x}.p_i \to e_i \mid \langle t \rangle \bullet \mid \mid \text{¡}C(\vec{v}, \bullet, \vec{e}) \mid\ !\bullet \mid \bullet := e \mid v := \bullet \end{array}$$

Memory: finite map from locations to values
Evaluation contexts

$\pi \vdash e \leadsto e' \qquad \pi \vdash (M, e) \leadsto (M', e')$ Small-step typed reduction rules

$$\frac{\pi \vdash e \leadsto e'}{\pi \vdash (M, E \cdot e) \leadsto (M, E \cdot e')}\ \text{(E-ECTX)} \qquad \frac{\pi \vdash (M, e) \leadsto (M', e')}{\pi \vdash (M, E \cdot e) \leadsto (M', E \cdot e')}\ \text{(E-CTX)} \qquad \frac{\mathbf{letpol}\ \mathtt{x}\ v \in \pi}{\pi \vdash \mathtt{x} \leadsto v}\ \text{(E-POL)}$$

$$\pi \vdash \Lambda \alpha :: \kappa.e\ [t] \leadsto (\alpha \mapsto t)e\ \text{(E-TAP)} \qquad \pi \vdash \langle t \rangle \langle t' \rangle u \leadsto \langle t \rangle u\ \text{(E-LAB)} \qquad \frac{t \neq t'\{e\}}{\pi \vdash \langle t \rangle u \leadsto u}\ \text{(E-LAB2)} \qquad \frac{v_1 = \varphi\vec{x}.\lambda x : t.e}{\pi \vdash v_1^{\sigma} v_2 \leadsto (\sigma, x \mapsto v_2)e}\ \text{(E-APP)}$$

$$\frac{\forall i < j.FV(v); FV(p_i); \cdot \nvdash v \leq p_i : \sigma_i \qquad FV(v); FV(p_j); \cdot \vdash v \leq p_j : \sigma_j \qquad \mathrm{dom}(\sigma_j) \subseteq \vec{b_j}}{\pi \vdash \mathbf{match}\ v\ \mathbf{with}\ \vec{b_i}.p_i \to e_i \leadsto \sigma_j(e_j)}\ \text{(E-MATCH)} \qquad \frac{e = \mathbf{fix}\ f.v \qquad v' = (f \mapsto e)v}{\pi \vdash e \leadsto (f \mapsto v')v}\ \text{(E-FIX)}$$

$$\frac{M = M_1, (\ell, v), M_2}{\pi \vdash (M, !\ell) \leadsto (M, v)}\ \text{(E-DRF)} \qquad \frac{M = M_1, (\ell, v'), M_2}{\pi \vdash (M, \ell := v) \leadsto ((M_1, (\ell, v), M_2), ())}\ \text{(E-UPD)}$$

$\pi \vdash t \leadsto t'$ Evaluation relation on terms lifted to types

$$T \quad ::= \quad ◗ \mid \text{¡}◗ \mid ◗\{e\} \mid (\vec{y}.x : ◗) \to t \mid (\vec{y}.x : t) \to ◗ \mid \forall \alpha :: \kappa.◗ \mid \mathbf{lab}\sim \bullet \mid t\{\bullet\} \qquad \pi \vdash t \leadsto t\ \text{(TE-ID)} \qquad \frac{\pi \vdash t \leadsto t'}{\pi \vdash T.t \leadsto T.t'}\ \text{(TE-CTXT)}$$

$$\frac{\pi \vdash e \leadsto e'}{\pi \vdash T \cdot e \leadsto T \cdot e'}\ \text{(TE-CTXE1)} \qquad \frac{\pi \vdash e' \leadsto e}{\pi \vdash T \cdot e \leadsto T \cdot e'}\ \text{(TE-CTXE2)}$$

Figure 13. Operational semantics of FABLE.

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

CASE (T-L1): e is either $C(\vec{v}, \cdot e, \vec{e})$ or $¡C(\vec{v}, \cdot e, \vec{e})$, both of which are legal evaluation contexts. By the induction hypothesis $\pi \vdash e \rightsquigarrow e'$. Thus, by (E-CTX) the goal is established. Otherwise, $e = C(\vec{v})$ or $¡C(\vec{v})$, both of which are values.

CASE (T-L2): Same as the previous case.

CASE (T-L3), (T-L4): Straightforward by application of the induction hypothesis.

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

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

CASE (T-TAP): By the definition of evaluation contexts $e = \cdot e\ [t]$ or $v[t]$. In the first case, we use the induction hypothesis and apply (E-CTX) to show that $\pi \vdash (M, \cdot e\ [t]) \rightsquigarrow (M', \cdot e'\ [t])$. In the second case, by second premise of (T-TAP) we have $\Omega \vdash v : \forall \alpha{::}\kappa.t$. By enumeration of the possible syntactic forms of v we find that this premise must be an either an application of the (T-TAB) rule and $v = \Lambda \alpha{::}\mathrm{k}.e$ and (E-TAP) is applicable. (The possiblity of v being a labeled pre-value $\langle t \rangle u$ is ruled out, since we syntactically require t to be a labeled type; $\forall \alpha{::}\kappa.t$ is not a labeled type.)

CASE (T-APP): If e is either $\cdot e_1^{\sigma} e_2$ or $v^{\sigma} \cdot e_2$, then, by applying the induction hypothesis to the second and third premises respectively, we get our result using (E-CTX). If, e is $v_1^{\sigma} v_2$ then, we can conclude that the second premise of (T-APP) must be an application of the (T-ABS) rule and $v_1 = \varphi \vec{y}.x : t.e$. Following an argument similar to (T-TAP), we have that (E-APP) is applicable and $\pi \vdash v_1^{\sigma} v_2 \rightsquigarrow (x \mapsto v_2, \sigma)e$.

CASE (T-ASN): If we have $e = \cdot e_1 := e_2$ 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, $e = (l := v_2)$, then by assumption, $\mathrm{dom}(\Omega.\Gamma) = \mathrm{dom}(M) \cup \{\mathtt{x} \mid \textbf{letpol}\ \mathtt{x}\ v \in \pi\}$. Then, by $\Omega \models M$, $M = M_1, (\ell, v), M_2$, satisifying the premise of (E-UPD) and enabling a reduction.

CASE (T-MATCH): If e is **match** e' **with**... then we just use the induction hypothesis on the sixth premise of (T-MATCH) and we have our result via (E-CTX). If, however, e is **match** v **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 $(\vec{b_n}, p_n) = (¡x_{def}, x_{def})$. Thus, it suffices to show that $\cdot; ¡x_{def}; \cdot \vdash v \leq x_{def} : x_{def} \mapsto v$, for all closed label-typed values v. As $\Omega \vdash v :$ ¡**lab** is established in the premises of (T-MATCH), we find that (U-VR) is applicable and thus a reduction is possible using (E-MATCH).

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-DRF) is possible.

CASE (T-PFX), (T-SFX): If $e = \langle t \rangle e'$ then by using the induction hypothesis on the second premise of either rule we have our result via (E-CTX). If $e = \langle t \rangle u$, and t is a labeled type, then, e is a value. If, t is an unlabeled type, then reduction can proceed via (E-LAB2). What remains is $e = \langle t \rangle \langle t' \rangle u$. In this case, (E-LAB) is applicable.

CASE (T-EVT): Straightforward after the induction hypothesis is used on the first premise. □

Lemma 4 (Preservation of well-formedness). *If Ω is well-formed, and $\Omega \vdash \pi; e : t$ contains a sub-derivation of the form $\Omega' \vdash_{\varepsilon'} e : t$ or $\Omega' \vdash t :: k$ then Ω' is well-formed.*

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

Lemma 5 (Expression unification). *Given $\cdot; \vec{b_2}; \Omega \vdash e_1 \leq e_2 : \sigma$ such that following conditions hold:*

(A1) $FV(e_1) \setminus dom(\Omega.\Gamma) = \emptyset$ *and* $\Omega \vdash_{\varphi} e_1 :$ ¡**lab**
(A2) $FV(e_2) \setminus dom(\Omega.\Gamma) \subseteq \{x | \{x, ¡x\} \cap \vec{b_2} \neq \emptyset\}$
(A3) $\{x | \{x, ¡x\} \cap \vec{b_2} \neq \emptyset\} \cap dom(\Omega.\Gamma) = \emptyset$

Then, the following are true:

i. $dom(\sigma) \subseteq \{x | \{x, ¡x\} \cap \vec{b_2} \neq \emptyset\}$
ii. $range(\sigma) = \{v_1, \ldots, v_n\}$
iii. $v \in range(\sigma) \Rightarrow \Omega \vdash_{\varphi} v :$ ¡**lab**
iv. $(FV(e_2) \setminus dom(\Omega.\Gamma)) = dom(\sigma)$
v. $\Omega.A = \emptyset \Rightarrow \sigma(e_2) = e_1$ *(subject to α-renaming of bound variables)*

Proof. CASE (U-EID): By assumption (A1) we have $FV(e) \setminus \mathrm{dom}(\Gamma) = \emptyset = \mathrm{dom}(\sigma) \subseteq \vec{b_2}$. Proposition (v) is trivial.

CASE (U-CON, U-RCON): We use the induction hypothesis for the first $i-1$ premises and the assumptions to show that the induction hypothesis is applicable to the ith premise. From assumption (A1) and proposition (i) we can conclude that $\mathrm{dom}(\sigma_i^*) \cap FV(e_1) = \emptyset$; thus $\sigma_i^*(e_i) = e_i$ and (A1) is re-established. From proposition (ii) and (iii) we can conclude that $(FV(\sigma_i^*(e_i') \setminus \mathrm{dom}(\Omega.\Gamma)) \subseteq (FV(e_i') \setminus \mathrm{dom}(\Omega.\Gamma))$, thus re-establishing (A2). (A3) is trivial, since $\vec{b_2}$ does not change for each premise and is true initially by assumption. Thus, the induction hypothesis is applicable to the ith premise. To conclude, we must show that proposition (iv) holds. However, this is straightforward by observing that $i \neq j \Rightarrow \mathrm{dom}(\sigma_i) \cap \mathrm{dom}(\sigma_j) = \emptyset$ and $\mathrm{dom}(\vec{\sigma}) = \bigcup_i \mathrm{dom}(\sigma_i)$. Finally, proposition (v.) holds by using the induction hypothesis on each e_i, and by noting that (U-CON) and (U-RCON) require the top-level constructor in e_1 and e_2 to be the same.

CASE (U-VL): Impossible, since by assumption $\vec{b_1} = \emptyset$.

CASE (U-VR1, U-VR2): For proposition (i), we use the first premise to confirm that $¡x \in \vec{b_2}$; proposition (ii) is immediate since v is a value and from the second premise (using T-SUBR to conclude in the case of U-VR1); proposition (iii) is immediate from the assumption; proposition (iv) follows from assumptions (A2) and (A3); proposition (v) is true by construction since $(x \mapsto v)x = v$.

CASE (U-AS): By clause (v) of the well-formedness of Ω (Definition 1), we have that $\Omega \vdash_{\varphi} e_1' :$ ¡**lab**; thus assumption (A1) is established and we can use the induction hypothesis on the second premise. Proposition (v) is true vacuously, since $\Omega.A \neq \emptyset$. □

Lemma 6 (Type unification)**.** *Given $\vec{b_2};\Omega \vdash t_1 \leq t_2 : \sigma$, such that the following conditions hold:*

A1 $FV(t_1) \setminus dom(\Omega.\Gamma) = \emptyset$
A2 $FV(t_2) \setminus dom(\Omega.\Gamma) \subseteq \{x | \{x, ¡x\} \cap \vec{b_2} \neq \emptyset\}$
A3 $\{x | \{x, ¡x\} \cap \vec{b_2} \neq \emptyset\} \cap dom(\Omega.\Gamma) = \emptyset$

Then, the following are true:

i. $dom(\sigma) \subseteq \{x | \{x, ¡x\} \cap \vec{b_2} \neq \emptyset\}$
ii. $(FV(t_2) \setminus dom(\Gamma)) = dom(\sigma)$
iii. $range(\sigma) = \{v_1, \ldots, v_n\} \wedge \forall i.\Omega \vdash v_i :$ **¡lab**.
iv. $\Omega.A = \emptyset \Rightarrow \sigma(t_2) = t_1$ *(subject to α-renaming of bound variables)*

Proof. CASE (Case U-ID): : By (A1) $FV(t) \setminus \mathrm{dom}(\Gamma) = \emptyset = \mathrm{dom}(\sigma)$. Proposition (iv) is trivial.

CASE (Case U-LAB): : Lemma 5 is applicable on the premise to establish our conclusion.

CASE (Case U-UNI): : We use the induction hypothesis on the premise and note that $FV((\beta \rightarrow \alpha)t_2) \setminus \mathrm{dom}((\Gamma, \alpha :: \kappa))$ is the same as $FV(\forall\beta :: \kappa.t_2) \setminus \mathrm{dom}(\Gamma)$. By the induction hypothesis, we have that $t_1 = (\sigma, \beta \mapsto \alpha)t_2$. Thus, by α-renaming of the bound variable in $\forall\beta{::}k.t_2$, we have proposition (iv.).

CASE (Case U-FUN): : We must first show that the induction hypothesis is applicable on the second and third premises.

For premise 2: (A1) is established by noting that by (A1) $FV(\vec{b_x}.x : t_1 \rightarrow t_2) \setminus \mathrm{dom}(\Omega.\Gamma) = \emptyset$ and, by definition, $(FV(t_1) \setminus \vec{b_x}) \subseteq FV(\vec{b_x}.x : t_1 \rightarrow t_2)$; thus $FV(t_1) \setminus (\mathrm{dom}(\Omega.\Gamma) \cup \vec{b_x}) = \emptyset$. (A2) is established using a similar argument for $\sigma_0 t_1'$ and noting that $\vec{b_x}$ are all fresh (by α-conversion, if necessary). (A3) also follows from the freshness of $\vec{b_x}$.

For premise 3: (A1) is established by noting additionally that x is within scope in t_2 and, thus, is added to $\Omega.\Gamma$. For (A2), we use the induction hypothesis on premise 1 to conclude that $\mathrm{dom}(\sigma_1) \subseteq \vec{b_2}$; thus $(FV(\sigma_1\sigma_0 t_2') \setminus \mathrm{dom}(\Omega.\Gamma)) \subseteq (FV(\sigma_0 t_2') \setminus \mathrm{dom}(\Omega.\Gamma))$. (A3) follows from freshness as previously.

To establish proposition (i), we simply use $\mathrm{dom}(\sigma_1, \sigma_2) = \mathrm{dom}(\sigma_1) \cup \mathrm{dom}(\sigma_2)$ and from the induction hypothesis we have $\forall i.\mathrm{dom}(\sigma_i) \subseteq \vec{b_2}$. Similarly, for proposition (iii).

Proposition (iv) is straightforward from the induction hypothesis, and α-renaming in the conclusion, if necessary, as in the (U-UNI) case.

To establish proposition (ii), our goal is

$$\begin{array}{rcl} \mathrm{dom}(\sigma_1, \sigma_2) & = & FV((\vec{b_y}.y : t_1') \rightarrow t_2') \setminus \mathrm{dom}(\Gamma) \\ & = & (FV(t_1') \cup FV(t_2')) \setminus (\{y, y_1, \ldots, y_n\} \cup \mathrm{dom}(\Gamma)) \end{array}$$

To show this, note that $\mathrm{dom}(\sigma_0) = y, \vec{b_y}$ and $\mathrm{range}(\sigma_0) = x, \vec{b_x}$. By the induction hypothesis on the second premise, we can conclude that

$$\begin{array}{rcl} \mathrm{dom}(\sigma_1) & = & FV(\sigma_0 t_1') \setminus (\mathrm{dom}(\Gamma) \cup \vec{b_x}) \\ & = & FV(t_1') \setminus (\mathrm{dom}(\Gamma) \cup \{y_1, \ldots, y_n\}) \end{array}$$

since y does not appear in t_1'. By proposition (iii) of Lemma 5 $FV(\mathrm{range}(\sigma_1)) \subseteq \mathrm{dom}(\Gamma)$.
So, applying the induction hypothesis to the third premise, we have

$$\begin{array}{rcl} \mathrm{dom}(\sigma_2) & = & FV(\sigma_1\sigma_0 t_2') \setminus (\mathrm{dom}(\Gamma) \cup \vec{b_x} \cup x) \\ & = & FV(\sigma_0 t_2') \setminus (\mathrm{dom}(\Gamma) \cup \vec{b_x} \cup x \cup \mathrm{dom}(\sigma_1)) \\ & = & FV(t_2') \setminus (\mathrm{dom}(\Gamma) \cup \{y, y_1, \ldots, y_n\} \cup \mathrm{dom}(\sigma_1)) \end{array}$$

$$\begin{array}{rcl} \mathrm{dom}(\sigma_1, \sigma_2) & = & (FV(t_2') \setminus (\mathrm{dom}(\Gamma) \cup \{y, y_1, \ldots, y_n\} \cup \mathrm{dom}(\sigma_1))) \\ & \cup & (FV(t_1') \setminus (\mathrm{dom}(\Gamma) \cup \{y_1, \ldots, y_n\}) \\ & = & (FV(t_1') \cup FV(t_2')) \setminus (\mathrm{dom}(\Gamma) \cup \{y, y_1, \ldots, y_n\}) \end{array}$$

CASE (Case U-SEC): Using the induction hypothesis on the first premise and Lemma 5 on the second premise. The argument for $\mathrm{dom}(\sigma_1, \sigma_2)$ is identical to (U-FUN). □

Lemma 7 (Unification under substitution)**.** *Given well-formed Ω and types t_1 and t_2 such that the following conditions are true:*

i. $\vec{b} = \vec{x}, ¡\vec{y} = FV(t_2) \setminus dom(\Omega.\Gamma)$
ii. $\Omega \vdash t_1 :: \kappa$*;* $\Omega[\vec{b}] \vdash t_2 :: \kappa$
iii. $\vec{b}; \Omega \vdash t_1 \leq t_2 : \sigma$
iv. σ^* *such that* $dom(\sigma^*) \cap \{\vec{x}, \vec{y}\} = \emptyset$.
v. $\sigma^*\Omega \vdash \sigma^* t_1 :: \kappa$*;* $\sigma^*\Omega[\vec{b}] \vdash \sigma^* t_2 :: \kappa$*;*

Then
$\vec{b}; \sigma^*\Omega \vdash \sigma^* t_1 \leq \sigma^* t_1 : \sigma^* \star \sigma$*, where* $\sigma^* \star \sigma = \{(x \mapsto \sigma^*(v)) \mid (x \mapsto v) \in \sigma\}$

Proof. By induction on the structure of (iii), and simultaneously on the structure of $\cdot; \vec{b}; \Omega \vdash e_1 \leq e_2 : \sigma$.

CASE (U-TID): Trivial.

CASE (U-UNI): Straightforward from use of the induction hypothesis.

CASE (U-LAB): Straighforward from use of the induction hypothesis on the expression unification judgment.

CASE (U-SEC): On the first premise, we can use the induction hypothesis to establish $\vec{b_2}; \sigma^*\Omega \vdash \sigma^* t_1 \leq \sigma^* t_2 : \sigma^* \star \sigma$. For the second premise, we must show that

$$\vec{b_2}; \sigma^*\Omega \vdash \sigma^* e_1 \leq (\sigma^* \star \sigma)\sigma^* e_2 : \sigma^* \star \sigma'$$

But, notice that we can easily apply the induction hypothesis on the expression unification judgment to show

$$\vec{b_2}; \sigma^*\Omega \vdash \sigma^* e_1 \leq \sigma^* \sigma e_2 : \sigma^* \star \sigma'$$

Thus, our goal is to show $\sigma^* \sigma e_2$ is equivalent to $(\sigma^* \star \sigma)\sigma^* e_2$. First, we use Lemma 6 assumption (iv) to show $\mathrm{dom}(\sigma) \cap \mathrm{dom}(\sigma') = \emptyset$ and the definition of $\star$ to show that $\mathrm{dom}(\sigma^* \star \sigma) \cap \mathrm{dom}(\sigma') = \emptyset$. Thus, the two substitutions commute and we can write

$$(\sigma^* \star \sigma)\sigma^* e_2 = \sigma^*(\sigma^* \star \sigma)e_2$$

Finally, from the definition of $\star$ we can show $\sigma^*(\sigma^* \star \sigma)e_2 \equiv \sigma^* \sigma e_2$ since $\sigma^* \star \sigma$ transforms σ by applying σ^* to the range. The same result is obtained by transforming the values from the range of σ that have been substituted into e_2 by σe_2.

CASE (U-FUN): To use the induction hypothesis on the first premise, we must show that $\sigma_0 \sigma^* t_1' = \sigma^* \sigma_0 t_1'$. However, we are always free to α-rename $\vec{b_x}$ and $\vec{b_y}$ to ensure that $\mathrm{dom}(\sigma_0) \cap$

$\text{dom}(\sigma^*) = \emptyset$; thus the substitutions commute and we can use the induction hypothesis. For the second premise, we follow the (U-SEC) case and use Lemma 6 and assumption (iv.) to show that $\text{dom}(\sigma_1) \cap \text{dom}(\sigma^*) = \emptyset$ and use commutativity to apply the induction hypothesis.

We now proceed to the cases of the expression unification judgment.

CASE (U-EID): Trivial; same as (U-TID).

CASE (U-VL): Inapplicable, since we assume that $\vec{b_1} = \cdot$.

CASE (U-VR1, U-VR2): By the first premises of each rule and assumption (iv) we have that $x \notin \text{dom}(\sigma^*)$. Thus, the conclusion of each of these rules produces $x \mapsto \sigma^* v$ which is precisely the definition of $\sigma^* \star (x \mapsto v)$.

CASE (U-CON, U-RCON): These are identical to (U-SEC) and (U-FUN); show that the induction hypothesis is applicable by using assumption (iv) and Lemma 6 to show that $\forall i.\text{dom}(\sigma_i) \cap \text{dom}(\sigma^*) = \emptyset$.

CASE (U-AS): A straightforward application of the induction hypothesis since, by definition, $e_1 \leq e_1' \in \Omega.A \Rightarrow \sigma^* e_1 \leq \sigma^* e_1' \in (\sigma'\Omega).A$. □

Lemma 8 (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:*

i. $\Omega.R = \Omega.A = \emptyset$
ii. $\Omega' = \Omega[\alpha :: \kappa]$
iii. $\Omega'[\Omega''] \vdash_{\varepsilon} e : t$
iv. $\Omega \vdash_{\varepsilon} t' :: \kappa$
v. $\sigma = kw'a \mapsto t'$

Then,

$$\Omega[\sigma\Omega''] \vdash_{\varepsilon} \sigma e : \sigma t$$

Proof. By induction on the structure of (iii.). Totally standard proof. The interesting cases are (T-SFX) and (T-PFX), where types can appear within terms.

□

Lemma 9 (Contraction of assumptions)**.** *For well-formed Ω and $\Omega[e_1 \leq e_2]$, if both of the following are true*

i. $\vec{b_2}; \Omega[e_1 \leq e_2] \vdash t_1 \leq t_2 : \sigma$
ii. $\cdot; \vec{b_2'}; \Omega \vdash e_1 \leq e_2 : \sigma'$ *with $\vec{b_2'}$ disjoint from $\vec{b_2}$*

Then, $\vec{b_2}; \Omega \vdash t_1 \leq t_2 : \sigma''$ where $\sigma'' = \sigma$ or $\sigma'' = \sigma' \circ \sigma$. (The composition of the substitutions)

Proof. Sketch: If (i) does not contain a sub-derivation with a non-trivial (U-AS) then the proof is straightforward using $\sigma'' = \sigma$.

If (i) does contain an interesting application of (U-AS) then, it must be of the form $\cdot; \vec{b_2}; \Omega[e_1 \leq e_2] \vdash e_1 \leq e_2' : \sigma_0$ with $\cdot; \vec{b_2}; \Omega \vdash e_2 \leq e_2' : \sigma_0$ as a premise. However, by construction, from assumption (ii) we have $\sigma'(e_2) = e_1$, and by Lemma 5, $\text{dom}(\sigma') \cap FV(e_1) = \emptyset$. Thus, it is straightforward to establish $\cdot; \vec{b_2}; \Omega \vdash e \leq e_2' : \sigma' \circ \sigma_0$. Finally, since, from assumption (ii), $\vec{b_2'}$ is disjoint from $\vec{b_2}$ we have $\text{dom}(\sigma')$ disjoint from $\text{dom}(\sigma_0)$ and from $\text{dom}(\sigma)$. Proceed by induction on each (U-*) and use disjoint-ness to show that $(\sigma' \circ \sigma_0), \sigma_1 = \sigma' \circ (\sigma_0, \sigma_1)$. □

Lemma 10 (Soundness of reduction for type-level expressions)**.** *Given well-formed $\Omega = \Gamma; \pi; \cdot; \cdot$ and e and e' such that*

i. $\Omega[\psi = 0] \vdash_{\varepsilon} e : t$ *and* $\Omega[\psi = 0] \vdash_{\varepsilon} e' : t$
ii. *σ, a substitution of free variables in e such that* $\sigma(\Omega)[\psi = 0] \vdash_{\varepsilon} \sigma(e) : \sigma(t)$
iii. *range*$(\sigma) \subseteq 2^V$*, all values, where* $\forall x \in dom(\sigma).\Omega \vdash_{\varepsilon} x : t \Rightarrow \Omega \vdash_{\varepsilon} \sigma(x) : t$.
iv. $(\pi \vdash e \rightsquigarrow e')$

$$(\pi \vdash \sigma(e) \rightsquigarrow \sigma(e'))$$

Proof. We proceed first by induction on the structure of iv.(a) $\pi \vdash e \rightsquigarrow e'$.

CASE (E-ECTX): Trivial application of induction hypothesis on each of the syntactic forms of the evaluation contexts.

CASE (E-POL): $\mathtt{x} \notin \text{dom}(\sigma)$. Thus, evaluation can proceed via (E-POL).

CASE (E-TAP): Trivial.

CASE (E-LAB, E-LAB2): Trivial.

CASE (E-FIX): $f \notin \text{dom}(\sigma)$; trivial.

CASE (E-APP): $\sigma(v_1^{\sigma_0} v_2) = \sigma(v_1)^{\sigma(\sigma_0)} \sigma(v_2)$. If range$(\sigma)$ is all values, $\sigma(v_2) = v_2'$ ($\sigma(v_1)$ is always a value) and reduction of $\sigma(v_1^{\sigma_0} v_2) = \sigma(v_1)^{\sigma(\sigma_0)} \sigma(v_2)$ can proceed by E-APP. So, in one step of reduction we get $\sigma(\sigma_0, x \mapsto \sigma(v_2))e$, which is our objective.

CASE (E-MATCH): We can assume $\text{dom}(\sigma) \cap \vec{b_i} = \emptyset$. First, we must show that if

$$FV(v); FV(p_i); \cdot \not\vdash v \leq p_i : \sigma_i$$

then

$$FV(\sigma v); FV(\sigma p_i); \cdot \not\vdash \sigma v \leq \sigma p_i : \sigma_i$$

. This is easily shown, by contradiction. If σ does permit v to be unified with p_i, then σ is a unifier of v and p_i. However, the expression unification judgment computes most general unifiers and by assumption, no such unifier exists. Thus, σ cannot be a unifier.

Next, we consider

$$FV(v); FV(p_j); \cdot \vdash v \leq p_j : \sigma_j$$

We have $\text{dom}(\sigma_j) \subseteq \vec{b_j}$; and, by assumption, $\text{dom}(\sigma_j) \cap \text{dom}(\sigma) = \emptyset$. Thus, following Lemma 7, it is easy to show that $FV(\sigma v); FV(\sigma p_j); \cdot \vdash \sigma v \leq \sigma p_j : \sigma \star \sigma_j$. The result of the reduction is therefore $(\sigma \star \sigma_j)(\sigma e_j)$, which as argued in Lemma 7, is equivalent to $\sigma \sigma_j e_j$, as desired.

□

Lemma 11 (Soundness of type evaluation)**.** *Given well-formed $\Omega = \Gamma; \pi; \cdot; \cdot$ such that*

i. $\Omega \vdash t :: \kappa$

ii. *σ a substitution of label variables in t, such that $\sigma(\Omega) \vdash \sigma(t) :: \kappa$ and range(σ) all values.*
iii. $\pi \vdash t \rightsquigarrow^* t'$

$\pi \vdash \sigma(t) \rightsquigarrow^* \sigma(t')$.

Proof. Straightforward induction on the structure of $\pi \vdash t \longrightarrow^* t'$ and using Lemma 10 in the forward direction for (TE-CTXE) and backwards for (TE-CTXE2). □

Lemma 12 (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:*

i. $\Omega.R = \Omega.A = \emptyset$
ii. $\Omega' = \Omega[\varphi\vec{b}][x:t_1][R_x]$, *where*
  a. $\vec{b} = \vec{l}, ¡\vec{m}$
  b. $\vec{l} \cup \vec{m} = FV(t_1) \setminus dom(\Omega.\Gamma)$
  c. $R_x = \{x : t_1\}$ *if* $\Omega[\varphi\vec{b}] \vdash t : R$; $R_x = \emptyset$ *otherwise.*
iii. $\Omega'[\Omega''] \vdash_{\varepsilon} e : t$
iv. $\Omega[\psi = 0] \vdash_{\varepsilon} e' : t'_1$ *where* $\exists v. e' = v$
v. $\vec{b}; \Omega \vdash t'_1 \leq t_1 : \sigma$
vi. $\sigma' = \sigma, x \mapsto e'$

Then,

$$\Omega[\sigma'\Omega''] \vdash_{\varepsilon} \sigma'(e) : \sigma'(t_2)$$

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

Throughout, we are free to assume $\sigma'(\Omega) = \Omega$ from the Lemma 6 which gives us that dom$(\sigma') = \{\vec{l}, \vec{m}, x\}$, and from clause (iv) of well-formedness of Ω which gives us that dom$(\sigma') \cap FV(\Omega) = \emptyset$.

CASE (T-VAR): Here we have two sub-cases, depending on whether or not $x \in$ dom(σ').

Sub-case (a): Assumption (iii) is of the form $\Omega[\varphi\vec{b}][x:t_1][R_x = \emptyset][\Omega''] \vdash y : t_2$; $y \notin \{x, \vec{b}\}$. Using Lemma 6, we can conclude that dom$(\sigma') = \{x, \vec{b}\}$, and thus, $\sigma'(y) = y$.

We have two further sub-cases:

Sub-case (a.i): $y : t_2 \in \Omega''.\Gamma$. In this case, $FV(t_2) \cap$ 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): Assumption (iii) is of the form $\Omega[\varphi\vec{b}][x:t_1][R_x][\Omega''] \vdash x : t_1$. But, $\sigma'(x) = e'$ and, so, from assumption (iv), $\Omega \vdash \sigma'(x) : t'_1$ is trivial. However, from clause (iv) of Lemma 6, since by assumption $\Omega.A = \emptyset$, we have $\sigma'(t_1) = t'_1$, so we have $\Omega \vdash_{\varepsilon} \sigma'(x) : \sigma'(t_1)$. Finally, we must show that $\Omega[\sigma'\Omega''] \vdash \sigma'(x) : \sigma'(t_1)$; 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.

CASE (T-RVAR): Again we have two sub-cases, depending on whether or not $x \in$ dom(σ').

Sub-case (a): Assumption (iii) is of the form $\Omega[\varphi\vec{b}][x:t_1][R_x][\Omega''] \vdash y : t_2$, where $y \notin \{x, \vec{b}\}$ and $R_x = \emptyset$. 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): Assumption (iii) is of the form $\Omega[\varphi\vec{b}][x:t_1][R_x = x : t_1][\Omega''] \vdash x : t_1$. Again, we proceed similar to (T-VAR) sub-case (b), using assumption (iv) and Lemma 6 to arrive at $\Omega \vdash \sigma'(x) : \sigma'(t_1)$. 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 of $\Omega''.\Gamma$ we reach our conclusion.

CASE (T-VARφ): Identical to (T-VAR) sub-case (a), since by assumption, this lemma does not apply directly to phantom variable substitutions. Note that this rule places no constraints on the form of $\Omega.R$; thus removing a relevant assumption in the conclusion poses no problems.

CASE (T-PVAR): Again, identical to (T-VAR) sub-case (a); this lemma does not apply to policy variable substitutions.

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 $\Omega[\sigma'\Omega''][R = \sigma' R_i] \vdash_{\varepsilon} \sigma'(e_i) : ¡\mathbf{lab}$ for the ith premise. For the conclusion, we note that $\sigma'(C(\vec{e})) = C(\sigma'(\vec{e}))$ and obtain $\Omega[\sigma'\Omega''] \vdash_{\varepsilon} \sigma'(C(\vec{e})) : \sigma'(\mathbf{lab} \sim C(\vec{e}))$, the desired result by noting that all of the R_i's are present in Ω'' and can therefore be partitioned according to the needs to the induction hypothesis. Similarly, for $¡C(\vec{e})$.

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

CASE (T-L3): Straightforward use of induction hypothesis on the premise.

CASE (T-L4): Induction hypothesis on both premises.

CASE (T-L5): Induction hypothesis on the premise, and similar to (T-L1) in the conclusion.

CASE (T-FIX): We have $f \notin$ dom(σ'). Thus, $\sigma'(\mathbf{fix}\ \mathsf{f}.v) = \mathbf{fix}\ \mathsf{f}.\sigma(v)$. Now, we can use the induction hypothesis on the second premise to establish the conclusion.

CASE (T-TAB): Since $\alpha \notin$ dom(σ') we can use the induction hypothesis on the second premise. However, we must be careful in the conclusion with R. If, $R_x \neq \emptyset$ in assumption (iii); and $\Omega''.R = \emptyset$, then, where initially we have $\Omega'[\Omega''] \vdash_{\varepsilon} e : ¡t_2$, in the conclusion we have $\Omega[\sigma'\Omega''] \vdash_{\varepsilon} \sigma' e : \sigma' t_2$. To restore the type to $¡\sigma'(t_2)$, we can use (T-SUBR), if necessary.

CASE (T-ABS): Our goal is to show, via (T-ABS), $\Omega[\sigma'\Omega''] \vdash_{\varepsilon} \varphi\vec{b_y}.\lambda y : \sigma'(s_1).\sigma'(e) : \sigma'(t)$, since dom$(\sigma')$ cannot mention $\vec{b_y}, y$. From assumption (iii), we have $\Omega'[\Omega''] \vdash_{\varepsilon} \varphi\vec{b_y}.\lambda y : s_1.e : t$ with the premise, $\Omega'[\Omega''][\varphi\vec{b_y}][y : s_1][R_y] \vdash_{\varepsilon} e : s_2$. However, this context is of the form $\Omega'[\Omega'']$, and is well-formed by Lemma 4. Thus, the induction hypothesis is applicable and we obtain

$$\Omega[\sigma'\Omega''][\varphi\vec{b_y}][y : \sigma' s_1][\sigma' R_y] \vdash_{\varepsilon} \sigma' e : \sigma' s_2$$

Thus, to reach our goal, we use this last judgment 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 13 (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 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 to establish

$$\Omega[\sigma'\Omega''][R = R_1 \setminus R_x] \vdash_{\varepsilon} \sigma' e_1 : ¡(\vec{b}.x : \sigma' t_1) \xrightarrow{fx} \sigma' t_2$$

Similarly, for the second premise we obtain

$$\Omega[\sigma'\Omega''][R = R_2 \setminus R_x] \vdash_{\varepsilon} \sigma' e_2 : \sigma t_1'$$

Now, by, Lemma 7 we obtain that $\vec{b};\Omega[\sigma'\Omega''] \vdash \sigma' t_1' \leq \sigma' t_1 : \sigma' \star \sigma$. This suffices to establish our conclusion.

CASE (T-ASN): Straightforward application of induction hypothesis to the first two premises.

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 five premises are trivial to re-establish since $\mathrm{dom}(\sigma')$ doesnt include any of $\vec{b_i}$. For the sixth premise, we use the induction hypothesis and restore $\sigma' e :$ ¡**lab**. Similarly, for the ninth premise. For the final premise, we have $\sigma' e_i : \sigma' t$; i.e. uniform type for each case of the pattern. The conclusion is straightforward.

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

CASE (T-PFX): We apply the induction hypothesis to the second premise of (ESC).

CASE (T-SFX): Similar to (T-PFX) on each premise.

CASE (T-EVT): Applying the induction hypothesis to the first premise, we obtain $\Omega[\sigma'\Omega''] \vdash_{\varepsilon} \sigma' e : \sigma' t$. Now, given $\pi \vdash t \rightsquigarrow^* t'$, we must show that $\pi \vdash \sigma' t \rightsquigarrow^* \sigma' t'$. This is exactly the statement of Lemma 11. □

Lemma 13 (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:*

i. $\Omega.R = \Omega.A = \emptyset$
ii. $\Omega' = \Omega[\varphi\vec{b}][x : t_1][R_x]$, *where*
  a. $\vec{b} = \vec{l}, ¡\vec{m}$
  b. $\vec{l} \cup \vec{m} = FV(t_1) \setminus dom(\Omega.\Gamma)$
  c. $R_x = \{x : t_1\}$ *if* $\Omega[\varphi\vec{b}] \vdash t : R$; $R_x = \emptyset$ *otherwise.*
iii. $\Omega'[\Omega''] \vdash_{\varepsilon} t : \kappa$
iv. $\Omega \vdash_{\varepsilon} v : t_1'$
v. $\vec{b};\Omega \vdash t_1' \leq t_1 : \sigma$
vi. $\sigma' = \sigma, x \mapsto e'$

Then,

$$\Omega[\sigma'\Omega''] \vdash_{\varepsilon} \sigma'(t) : \kappa$$

Proof. By simultaneous induction with cases of Lemma 12 on the structure of assumption (iii).

The only non-trivial cases are (K-SEC), (K-RSEC) and (K-ULAB). Each of these are satisfied by using the induction hypothesis for Lemma 12. □

Theorem 14 (Preservation)**.** *Given $\Omega \equiv \Gamma;\cdot;\cdot;\cdot$ and memory M such that $\Omega \models M$, and $\pi;e$ such that $(A1) \quad \Omega \vdash \pi;e : t$, then $(A2) \quad \pi \vdash (M,e) \rightsquigarrow (M',e')$ implies $\Omega \models M'$ and $\Omega \vdash \pi;e' : t$.*

Proof. By induction on the structure of the derivation (A2). Unless explicitly stated, $t' \equiv t$. We examine all the side-effect free cases first, (i.e., reductions via (E-ECTX) where $M = M'$) where the obligation of $\Omega \models M'$ is satisfied by assumption.

CASE (E-POL): By the syntactic form of $\mathtt{x}$, we can conclude that the derivation (A1) contains a sub-derivation with an application of (T-PVAR) $\Omega' \vdash_{\varepsilon} \mathtt{x} : t$, with $\mathtt{x} : t \in \Omega.\Gamma$. From the assumption of well-formedness of Ω, and Lemma 4, we can conclude that Ω' is well-formed too. Now, by clause (iii) of the definition of well-formedness (Definition 1), we can conclude $\Omega' \vdash_{\varepsilon} v : t$. To establish the conclusion, we use (A1) replacing the application of (T-PVAR) with $\Omega' \vdash_{\varepsilon} v : t$, which is permissible since we may apply weakening to the assumption $\mathtt{x} : t \in \Omega'.\Gamma$.

CASE (E-TAP): By assumption, the second premise of (T-PROG) concludes with an application of (T-TAP), $\Omega' \vdash_{\varepsilon} v\ [t_1] : (\alpha \mapsto t_1)t_2$.

Sub-case (a): If $v = \Lambda\alpha{::}\mathsf{k}.\mathsf{e}$ (i.e. no top-level relabeling operator), then the second premise of (T-TAP) must be an application (A3) of (T-TAB). In order to re-establish that the conclusion of (A2) is well-typed, we must show that $\Omega \vdash \pi;(\alpha \mapsto t_1)e : t$. However, this is a straightforward application of the type-substitution lemma, Lemma 8 using the second premise of the (T-TAB) sub-derivation (A3).

Sub-case (b): If $v = \langle t \rangle u$, then the second premise of (T-TAP) must be an application of (T-PFX) or (T-SFX), in either case with $\Omega' \vdash_{\cdot}^{ESC} \Lambda\alpha{::}\mathsf{k}.\mathsf{e} : \forall\alpha{::}\mathsf{k}.t'$, since u is a pre-value. However, by assumption, $\Omega.R = \ relevant(\Omega.\Gamma) = \emptyset$, and the second premise of (T-PROG) is of the form $\Omega' \vdash_{\cdot} v\ [t_1] : t$, where $\ relevant(\Omega'.\Gamma) = \emptyset$. Thus, $\Omega' \vdash_{\cdot}^{ESC} u : \forall\alpha{::}\mathsf{k}.t'$ is equivalent to its second premise, $\Omega' \vdash_{\cdot} u : \forall\alpha{::}\mathsf{k}.t'$, which is an application (A3) of (T-TAB). Again, by use of Lemma 8 on the second premise of (A3), the result is immediate.

CASE (E-FIX): The second premise of (T-PROG) concludes with (T-FIX). From the substitution Lemma 12 and noting that $\mathrm{dom}(\sigma') = \{f\} \not\subseteq \mathrm{dom}(\Omega.\Gamma)$ and $\mathrm{range}(\sigma') \subseteq 2^V$ satisfying the assumptions of the lemma.

CASE (E-LAB): From the structure of $e = \langle t \rangle \langle t' \rangle e'$, the second premise of (A1) contains a sub-derivation (A1.2) of the form $\Omega' \vdash_{\cdot} \langle t \rangle \langle t' \rangle u : t$ by an application of (T-PFX) or (T-SFX). Our goal is to show that $\Omega' \vdash_{\varepsilon} \langle t \rangle u : t$.

Sub-case (a): (A1.2) is an application of (T-PFX). Then (A4) $\Omega' \vdash_{\cdot}^{ESC} \langle t' \rangle u : t'$. However, just as in sub-case (b) of (E-TAP), we have that $relevant\Omega'.\Gamma = \emptyset$ and we can treat (A4) directly as an application of (T-PFX) or (T-SFX).

Sub-case (a.i): (A4) is (T-PFX). So, we have $\Omega' \vdash_{\varepsilon} u : t'\{\vec{e}\}$ and $t' = t\{\vec{e'}\}\{\vec{e}\}$. So, to re-establish the induction hypothesis, we use (T-PFX) with $\Omega' \vdash_{\varepsilon} u : t\{\vec{e'}\}\{\vec{e}\}$ in the premise via (ESC).

Sub-case (a.ii): (A4) is (T-SFX). So, we have $\Omega' \vdash_{\varepsilon} u : t''$ and $t' = t''\{\vec{e}_l\} = t\{\vec{e'_l}\}$. If the latter is a suffix of the former, we apply (T-SFX) else (T-PFX) to re-establish the induction hypothesis.

Sub-case (b): Similar.

CASE (E-APP): : By assumption, we have as the second premise of (A1), the sub-derivation (A1.2) $\Omega' \vdash v_1^{\sigma} v_2 : t$ with (T-APP), (or (T-SUBR), (T-EVT) etc.) as the top-most judgment. As previously, with sub-case (b) of (E-TAP) and (a) of (E-LAB), we can conclude that $\Omega'.R = \emptyset$.

If the top-level judgment of (A1.2) is (T-APP) then we immediately have (A1.2.2) $\Omega' \vdash v_1 : \vec{b}.x : t_1 \to t_2$ as the second premise, and (A1.2.3) $\Omega' \vdash v_2 : t_1'$ as the third premise. Now, as with (T-TAP), we must proceed on cases where either $\exists u.v_1 = u$ or $\exists u.v_1 = \langle t \rangle u$. Given that $\Omega'.R = \emptyset$, the distinction is not significant since, in the latter case, $\Omega' \vdash^{ESC} v_1 : tt$ can be replaced with $\Omega' \vdash v_1 : tt$. Thus, we can assume that $u = \varphi\vec{b}.\lambda x : t_1.e$, and that (A1.2.2) is an application of the (T-ABS) rule.

From the last premise of (T-ABS) in (A1.2.2), we have (A1.2.2.1), $\Omega'[\vec{b}][R = R'][x : t_1] \vdash e : t_2$, and from the fourth premise of (A1.2) we have, $\vec{b};\Omega' \vdash t_1' \leq t_1 : \sigma$, and, from clause (iii) of Lemma 6, we have that range(σ) is limited to values, and dom$(\sigma) = \vec{b}$, where dom$(\sigma) \cap FV(\Omega.\Gamma) = \emptyset$. Note that $R' \subseteq \{x : t_1\}$.

Our goal, then, is to show that $\Omega' \vdash (\sigma, x \mapsto v_2)e : (\sigma, x \mapsto v_2)t_2$. Using (A1.2.2.1), (A1.2.3), we can apply the subtitution lemma (Lemma 12) to obtain the result.

The other cases ((T-SUBR), (T-EVT) etc.) for the top-level judgment are handled separately.

CASE (E-MATCH): : The second premise of (A1) is an application (A1.2) of (T-MATCH), $\Omega' \vdash$ **match** v **with**... : t. Since, $\Omega \models M$ and dom$(\Omega') =$ dom$(\Omega) \cup \vec{x}$, we can conclude that $FV(v) = \emptyset$.

The third premise of (A2) (E-MATCH) gives us (A2.1) $\cdot;\vec{b}_j;\cdot \vdash v \leq p_j : \sigma_j$ and from the premises of (A1.2) $\Omega' \vdash_{\varepsilon} v :$ ¡**lab**. In order to establish the applicability of Lemma 5, we show that its condition (A1) is applicable since $FV(v) = \emptyset$; condition (A2) and (A1.2) from the fifth premise of (T-MATCH). From clause (iv) of Lemma 5, $FV(p_j) = \vec{b} = $ dom(σ) since, $\Omega.\Gamma$ is empty.

Furthermore, since by assumption v has type ¡**lab**, by (T-LAB) all sub-terms of v must be values of type ¡**lab**. By the last premise of (A1.2) (T-MATCH) we have (A1.2.1) $\Omega'[\vec{b}_j][v \leq p_j] \vdash_{\varepsilon} e_j : t$.

First, to establish $\Omega[\vec{b}_j] \vdash_{\varepsilon} e_j : t$, we rely on Lemma 9 in conjunction with (A1.2.1) and (A2.1). The assumption $v \leq p_j$ is only used in the second-to-last premise of (T-APP) in the form of (A1.2.1.x) $\vec{b};\Omega''[v \leq p_j] \vdash t_1 \leq t_2 : \sigma'$ with $\Gamma(\vec{b})$ *undefined*. Thus $\vec{b}_j$ in (A2.1) is disjoint from $\vec{b}$ in (A1.2.1.x) and Lemma 9 is applicable. We obtain that $\vec{b};\Omega'' \vdash t_1 \leq t_2 : \sigma_j \circ \sigma'$; from which, we can conclude $\Omega'[\vec{b}_j] \vdash_{\varepsilon} e_j : \sigma_j(t)$.

However, by noticing that each $\vec{b}_i$ is distinct, $FV(t) \cap ($dom$(\sigma_j) = \vec{b}_j) = \emptyset$, since otherwise the last premise of (T-MATCH) in (A1.2) requiring each e_i to have the same type t, could not be satisfiable. Thus $\sigma_j \circ \sigma' \equiv \sigma'$ and we have $\Omega'[\vec{b}_j] \vdash_{\varepsilon} e_j : t$.

Finally, by repeated application of the substitution lemma (Lemma 12) we have $\Omega' \vdash_{\varepsilon} \sigma_j(e_j) : \sigma_j(t)$. As noted previously, $\sigma_j(t) = t$, which is our goal.

CASE (E-DRF): By assumption we have in the second premise of (A1), (A1.2), $\Omega' \vdash_{\varepsilon} !l : t$ with $\Omega' \vdash_{\varepsilon} l : t$**ref** in the premise. By well-formedness of Ω' and $\Omega \models M$ we have that $\Omega' \vdash v : t$, since weakening of $\Omega.\Gamma$ is permissible.

CASE (E-UPD): To establish the well-formedness of M' we must establish that v_2 is not a name. But, by (A1) we have $\Omega' \vdash_{\varepsilon} v_2 : t$, where dom$(\Omega') =$ dom$(M) \cup$ dom(π) and dom(π) are not values. Thus, the only free names in v_2 are locations in M, which are permissible values.

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

Sub-case (T-SUBR): Using the induction hypothesis in (T-SUBR) is trivial; then, in order to establish that $\pi \vdash t \rightsquigarrow t' \Rightarrow \pi \vdash$ ¡$t \longrightarrow$¡t', we use (TE-CTXT) with the ¡● context.

Sub-case (T-EVT): We use the induction hypothesis for the first premise and the determinism of the reduction $\pi \vdash t \rightsquigarrow t'$ for the second premise.

Sub-case (T-L3): We use the induction hypothesis on the premise. Now, we must establish that $\pi \vdash$ **lab**$\sim e \rightsquigarrow t' \Rightarrow t' =$ **lab**$\sim e'$. However, this is is immediate from the definition of the type evaluation context, case **lab**$\sim\bullet$.

Sub-case (T-L4): We use the induction hypothesis on the first premise. Now, we must establish that $(\pi \vdash$ **lab**$\sim e' \rightsquigarrow$ **lab**$\sim e'') \Rightarrow \Omega \vdash_{\varepsilon} e''$: **lab**;0. But, from the definition of the type evaluation context, case **lab**$\sim\bullet$, we have that $\pi \vdash e' \rightsquigarrow e''$. Thus, using the second premise of (T-L4) and the induction hypothesis, together with the irreducibility of the type **lab** to establish our conclusion.

Sub-case (T-L5): We use the induction hypothesis to establish $\Omega \vdash_{\varepsilon} e'$: ¡**lab**. Now, we can use (T-L5) to establish that $\Omega \vdash_{\varepsilon} e'$: **lab**$\sim e'$. However, by assumption (A2), and the premise of (A1) that establishes that e has no side-effects, we have that $\pi \vdash e \rightsquigarrow e'$. Thus, using the the (T-EVT) with (TE-CTXE2) we re-establish $\Omega \vdash_{\varepsilon} e'$: **lab**$\sim e$, as desired.

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

Sub-case $(E = C(\vec{v}, \cdot e_j, \vec{e}))$: The judgment (A1) has the form $\Omega' \vdash_{\varepsilon} E : t$, using (T-L*). By the induction hypothesis, if (A1) is not (T-L1) or (T-L5), we can easily establish that $\Omega' \vdash_{\varepsilon} e'_j$: **lab** or $\Omega' \vdash_{\varepsilon} e'_j$: ¡**lab**, using (T-L2), (T-L3) or (T-L4) as necessary, and re-establish $\Omega' \vdash_{\varepsilon} E' : t$. However, if (A1) is (T-L1) or (T-L5), we have $t =$ **lab** $C(\vec{v}, e_j, \vec{e})$, while in the conclusion we have $t' =$ **lab** $C(\vec{v}, e'_j, \vec{e})$.

But, by the premises of (T-L1) and also in (T-L5), we have that e_j is effect-free. Thus, if $\pi \vdash (M, e_j) \rightsquigarrow (M, e'_j)$ then $\pi \vdash e_j \rightsquigarrow e'_j$; thus via (T-CTXE) for the label context, and using (TE-CTXE2) we have that $\pi \vdash t' \rightsquigarrow t$ and we can re-establish $\Omega \vdash_{\varepsilon} e$: **lab** $C(\vec{v}, e_j, \vec{e})$, using (T-L1) or (T-L5) as the premise of (T-EVT).

Sub-case $(E =$ ¡$C(\vec{v}, \cdot e_j, \vec{e}))$: Identical.

```
letpol lub λx:lab.λy:lab. match x,y with
          x, x → x
          LOW, _ → y
          MED, LOW → x
          MED, _ → y
          _ , _ → x

letpol low Λα.λx:α.<α{LOW}>^1 x

letpol join Λα.φl,m.λx:α{l}{m}.<α{lub l m}>^2 x

letpol sub Λα.φl.λx:α{l}.λm:lab.<α{lub l m}>^3 x

letpol def Λα.λl:lab.λx:α. sub [α] (low [α] x) l

letpol app Λα::U.Λβ.φl.λf:(α → β){l}.φm.λx:α{m}.
          <β{lub l m}>^4 (<α → β>^5f) (<α>^6 x)

letpol app2 Λα.Λβ.φl.λf:(α → β){l}.λx:α.
          <β{l}>^7 (<α → β>^8f) x

abbrv bool{l} = ∀α.(α → α→ α){l}
Λα.λb: bool{HIGH}.λx:α{LOW}.λy: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 14. Enforcing a static information flow policy.

Sub-case ($E = \cdot e_1 e_2$): The top-level typing judgment (A1) is (T-APP). Let the second premise of this judgment be $\Omega' \vdash_{\overline{\varepsilon}} e_1 : t_1$. To re-establish the hypothesis, we will apply (T-APP) again. By the induction hypothesis we have that $\Omega' \vdash_{\overline{\varepsilon}} e_1' : t_1$. 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 ($E = \cdot e\langle t \rangle$): The top-level typing judgment is either (T-SFX) or (T-PFX). We use the induction hypothesis and note that type-evaluation does not change the structure of a type; thus (T-SFX) or (T-PFX) respectively are still applicable.

Sub-case ($E =$ **match** $\cdot e$ **with**$\ldots$): The top-level typing judgment is (T-MATCH). We use the induction hypothesis on the first premise noting that $t' = t$ since the type **lab** is irreducible.

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

□

B. Correctness of the static information-flow policy

Figure 14 reproduces the policy of Figure 4. 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 B gives the semantics of FABLECORE^2, an extension of FABLECORE in the spirit of Core-ML^2 [21], 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 15 (Progress for FABLECORE^2)**.** *Given well-formed $\Omega = \cdot;\cdot;\cdot,\pi$ the policy of Figure 14, and a* FABLECORE^2 *program e such that $\Omega \vdash \pi; e : t$ and all relabeling operations in e are indexed by one of the indices that appear in text of π. Then, $\pi \vdash e \rightsquigarrow e'$, or e is a value.*

Proof. By induction on the structure of $\Omega \vdash \pi; e : t$, relying on Theorem 3 for most cases.

CASE (T-UNIT, T-VAR, T-LVAR, T-LAB, T-ABS, T-TAB, T-SFX, T-SUB, T-SUB2, T-EVT): Trivial, by using Theorem 3 and appealing to evaluation context $\{\!\{\bullet \parallel e\}\!\}$ or $\{\!\{e \parallel \bullet\}\!\}$.

CASE (T-PFX): If $e \neq \{\!\{v_1 \parallel v_2\}\!\}$ then we follow Theorem 3, appealing to evaluation context $\{\!\{\bullet \parallel e\}\!\}$ or $\{\!\{e \parallel \bullet\}\!\}$, if necessary. Otherwise, if $e = \{\!\{v_1 \parallel v_2\}\!\}$, then if $t :: \mathsf{U}$, by definition $\langle t\rangle e$ is a value.

CASE (T-APP): If e_1 and e_2 are not bracketed values, then we follow Theorem 3. If e_1 is bracketed, then e_1 must be $\langle t\rangle \{\!\{v_1 \parallel v_2\}\!\}$, since, by (T-BRACKET) $\Omega \vdash_{\overline{\varepsilon}} \{\!\{v_1 \parallel v_2\}\!\} : t'e$, 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 relabeling indices $i \in \{1 \ldots 8\}$. By inspection, it is straightforward to establish that $5, 6, 8$ are the only unlabeling operations in the program. In the first case, we have an evaluation context

$$E \cdot (\langle t\rangle^5 \{\!\{v_f \parallel v_f'\}\!\}) v_x$$

Again, by inspection of the relabeling indices, it is clear that this evaluation context can be expanded of

$$E \cdot \langle t\rangle^4 (\langle t\rangle^5 \{\!\{v_f \parallel v_f'\}\!\}) v_x$$

Similarly, for

$$E \cdot (\langle t\rangle^8 \{\!\{v_f \parallel v_f'\}\!\}) v_x$$

and

$$E \cdot \langle t\rangle^7 (\langle t\rangle^8 \{\!\{v_f \parallel v_f'\}\!\}) v_x$$

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

Finally, we observe that it is impossible for $e_1 = \langle t\rangle^6 \{\!\{v_1 \parallel v_2\}\!\}$. From the source program, we note that such a term is introduced by $e' = e_1' e_2'$ where $e_2' = \langle t\rangle^6 \{\!\{v_1 \parallel v_2\}\!\}$. 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 $e_2' = v_x = \langle t\rangle^6 \{\!\{v_1 \parallel v_2\}\!\}$, then, in one step of reduction, using the definition of $\lfloor v_x \rfloor$, the $\langle t\rangle^6 \{\!\{v_1 \parallel v_2\}\!\}$ term is immediately destructed. Similarly for (E-BAPP2). Thus, it is impossible for $\langle t\rangle^6 \{\!\{v \parallel v'\}\!\}$ to appear in the left-side of an application.

By the above argument, it is also straightforward to establish that if $e_2 = \langle t\rangle^6 \{\!\{v \parallel v'\}\!\}$, 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 $\langle t\rangle \{\!\{v_1 \parallel v_2\}\!\}$. 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

Additional syntactic forms

e	$::=$	$\ldots \mid \{\!\{ e_1 \parallel e_2 \}\!\}$	Bracketed expressions represent multiple executions
u	$::=$	$\ldots \mid \{\!\{ v_1 \parallel v_2 \}\!\}$	Pre-values
v	$::=$	$u \mid \langle t \rangle u$	Labeled pre-values; where t is of U only if $v = \{\!\{ v_1 \parallel v_2 \}\!\}$
E	$::=$	$\ldots \mid \{\!\{ \bullet \parallel e \}\!\} \mid \{\!\{ e \parallel \bullet \}\!\}$	Evaluation contexts

$\Omega \vdash_{\varepsilon} e : t$ Additional type rules

$$\frac{t = t'\{\vec{e_l}\} \quad \Omega \vdash t' :: \mathsf{U} \quad \pi \vdash \mathsf{lub}\ \vec{e_l} \rightsquigarrow^* \mathsf{HIGH} \quad \Omega \vdash_{\varepsilon} e_1 : t \quad \Omega \vdash_{\varepsilon} e_2 : t \quad \forall i. e_i \neq \{\!\{ e'_i \parallel e''_i \}\!\}}{\Omega \vdash_{\varepsilon} \{\!\{ e_1 \parallel e_2 \}\!\} : t} \text{(T-BRACKET)}$$

$\pi \vdash e \rightsquigarrow e'$ Additional reduction rules

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

$$\frac{u \neq \{\!\{ v_1 \parallel v_2 \}\!\} \quad t \neq t'\{e\}}{\pi \vdash \langle t \rangle u \rightsquigarrow u} \text{(E-LAB2)} \qquad \frac{v_1 = \varphi\vec{x}.\lambda x : t.e \quad v_2 \neq \langle t \rangle^6 \{\!\{ v'_2 \parallel v''_2 \}\!\}}{\pi \vdash v_1^{\sigma} v_2 \rightsquigarrow (\sigma, x \mapsto v_2)e} \text{(E-APP)}$$

$$\pi \vdash \langle t \rangle^{4,7}(\langle t_f \rangle^{5,8} \{\!\{ v_f \parallel v'_f \}\!\})^{\sigma} v_x \rightsquigarrow \{\!\{ \langle t \rangle^{4,7}(\langle t_f \rangle^{5,8} v_f)^{\sigma} \lfloor v_x \rfloor_1 \parallel \langle t \rangle^{4,7}(\langle t_f \rangle^{5,8} v'_f)^{\sigma} \lfloor v_x \rfloor_2 \}\!\} \text{(E-BAPP1)}$$

$$\frac{v_f \neq \langle t' \rangle \{\!\{ v_1 \parallel v_2 \}\!\}}{\pi \vdash \langle t \rangle^4 (v_f^{\sigma}(\langle t_x \rangle^6 \{\!\{ v_x \parallel v'_x \}\!\})) \rightsquigarrow \{\!\{ \langle t \rangle^4 v_f^{\sigma}(\langle t_x \rangle^6 v_x) \parallel \langle t \rangle^4 v_f^{\sigma}(\langle t_x \rangle^6 v'_x) \}\!\}} \text{(E-BAPP2)}$$

Figure 15. Semantics of FableCore^2.

in (T-APP) has shown, all these unlabeled bracketed terms are destructed immediately. Thus, the case of $v = \langle t \rangle \{\!\{ v_1 \parallel v_2 \}\!\}$ is ruled out and we can rely on Theorem 3. □

Lemma 16 (Subject reduction for FableCore^2). *Given well-formed $\Omega = \cdot;\cdot;\cdot;\cdot$, and π the policy of Figure 14, and a FableCore^2 program e such that all relabeling operators in e are annotated with an index from π, such that $\Omega \vdash \pi; e : t$. Then,*

$$\pi \vdash e \rightsquigarrow e' \Rightarrow \Omega \vdash \pi; e' : t$$

Proof. By induction on the structure of $\pi \vdash e \rightsquigarrow e'$. We must consider on the additional rules in Figure B, relying on Theorem 14 for the other cases.

Case (E-LAB2): This only applies to non-bracketed values, so there is no change from Theorem 14.

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 12 on the third and fourth premises, relying on the last premise to establish that the e_i are not bracketed values.

Case (E-BAPP1): It is straightforward to show that $\langle t_f \rangle v_f$ and $\langle t_f \rangle v'_f$ have the same type as $\langle t \rangle \{\!\{ v_f \parallel v'_f \}\!\}$, using the third and fourth premises of (T-BRACKET). Similarly for v_x and $\lfloor v_x \rfloor_i$, and finally that in $\pi \vdash e \rightsquigarrow \{\!\{ e_1 \parallel e_2 \}\!\}$ 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, $\langle t \rangle^4 = \langle t \mathsf{lub}\ \mathsf{l}\ \mathsf{m} \rangle^4$ and t_f is an unlabeled type, where $t_f\{l\}$ is the type of $\{\!\{ v_f \parallel v'_f \}\!\}$. Thus, by (T-BRACKET), it must be the case that $\mathsf{l} = \mathsf{HIGH}$. 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 $\langle t_x \rangle^6$ appears, we can conclude that $t_x :: \mathsf{U}$ and thus has no label. However, as previously $\langle 4 \rangle t = \langle 4 \rangle t'\{\mathsf{lub}\ \mathsf{l}\ \mathsf{m}\}$ where $t_x\{\mathsf{m}\}$ is the type of $\{\!\{ v_x \parallel v'_x \}\!\}$. From, (T-BRACKET) we can conclude that $t_x\{\mathsf{m}\}$ 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 17 (Noninterference). *Given π, the policy of Figure 4, and an expression e that does not contain any relabeling operations, and an empty context Ω ($\Omega.\Gamma = \Omega.\pi = \Omega.A = \cdot$) such that $\Omega[x : t\{\mathsf{HIGH}\}] \vdash \pi; e : t'\{\mathsf{LOW}\}$, where t' is of U-kind; and values v_1 and v_2:*

$$(\forall i \in \{1,2\}. \Omega \vdash \pi; v_i : t\{\mathsf{HIGH}\}1 \wedge \pi \vdash (x \mapsto v_i)e \rightsquigarrow^* v'_i) \Rightarrow v'_1 = v'_2$$

Proof. Straightforward from the substitution lemma, Lemma 12; Lemma 16 and from construction, $\Omega \vdash_{\varepsilon} v : t$ where t not guarded at HIGH, implies $\lfloor v \rfloor_1 = \lfloor v \rfloor_2$. □

C. Completeness of the static information-flow policy

In this section, we show that the information flow policy of Appendix B, Figure 14 is complete with respect to to the purely functional fragment of Pottier and Simonet's Core-ML [21]. Figure 16 reproduces the syntax and the static semantics of a minimal functional fragment of Core-ML.

Definition 18 (Non-degeneracy of Core-ML typing). *A Core-ML type $(t_1 \rightarrow t_2)^l$ is non-degenerate if, and only if, $l \triangleleft t_2$ and both t_1 and t_2 are non-degenerate; unit is non-degenerate. A typing derivation $\mathscr{D} = \Gamma \vdash_{ML} \mathtt{e} : \mathtt{t}$ is non-degenerate if, and only if, for every sub-derivation $\mathscr{D}'$ with conclusion $\mathtt{t}'$, $\mathtt{t}'$ 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 C shows a translation from Core-ML typing derivations $\mathscr{D}$ to FABLECORE programs e.

Theorem 19 (Completeness of static information flow)**.** *Given* e *such that,* $\mathscr{D} = \Gamma \vdash_{ML} \mathsf{e} : \mathsf{t}$ *is non-degenerate; then* $\llbracket\Gamma\rrbracket \vdash \pi; \llbracket\mathscr{D}\rrbracket : \llbracket\mathsf{t}\rrbracket$*, where* π *is the policy of Figure 14.*

Proof. By induction on the structure of the translation $\llbracket\mathscr{D}\rrbracket$.

CASE (X-U): Trivial.

CASE (X-V): Trivial.

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

$$\llbracket\Gamma, f : (\mathsf{t}_1 \to \mathsf{t}_2)^l, x : \mathsf{t}_1\rrbracket \vdash_{\varepsilon} \pi; \llbracket\mathscr{D}\rrbracket : \llbracket\mathsf{t}_2\rrbracket$$

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

$$\begin{array}{ll} i.\llbracket\Gamma\rrbracket \vdash \pi; \llbracket\mathscr{D}_1\rrbracket : \llbracket(t_1 \to t_2)^l\rrbracket, & \text{and,} \\ ii.\llbracket\Gamma\rrbracket \vdash \pi; \llbracket\mathscr{D}_2\rrbracket : \llbracket t_1\rrbracket & \end{array}$$

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

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

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

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 $\llbracket\Gamma\rrbracket, x : t_1' \vdash \llbracket\mathscr{D}_1\rrbracket : t_1$. Now, using the type for app given in (Case X-APP1), we conclude that app$\ldots\llbracket\mathscr{D}_1\rrbracket$ has type $t_2\{e_l\}$. After the application of join we conclude that e' has type $t\{\mathsf{lub}\ e_{l_2}\ e_l\}$. However, from the non-degeneracy assumption, we have $l \lhd \mathsf{t}_2$ or $l_2 \sqsubseteq l$; thus, the type of e' is $t\{e_{l_2}\} = \llbracket\mathsf{t}_2\rrbracket$. To type $\lambda x : t_1'.\llbracket\mathscr{D}_2\rrbracket$ we use the induction hypothesis to establish that $\llbracket\mathscr{D}_2\rrbracket$ 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 $\forall\alpha{::}\mathsf{M}.\mathsf{l}{:}\mathbf{lab} \to \alpha\to\alpha\{\mathsf{l}\}$, which is sufficient to establish the type of $(t_1' \to t_2')\{e_{l'}\} \equiv \llbracket(\mathsf{t}_1' \to \mathsf{t}_2')^{l'}\rrbracket$ for the translation, which is our goal. □

$$\begin{array}{lcll} \mathsf{e} & ::= & () \mid x \mid fix f.\lambda x.\mathsf{e} \mid \mathsf{e}_1\mathsf{e}_2 & \textit{expressions} \\ \mathsf{t} & ::= & \text{unit} \mid (\mathsf{t}_1 \to \mathsf{t}_2)^l & \textit{types} \end{array}$$

$$\text{Subtyping} \qquad (\ominus \to \oplus)^{\oplus}$$

$$\text{Guards} \qquad l \lhd \text{unit} \qquad \frac{\mathscr{L} \vdash l \sqsubseteq l'}{l \lhd (\mathsf{t} \to \mathsf{t}')^{l'}}$$

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

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

$$\frac{\Gamma \vdash_{ML} \mathsf{e}_1 : (\mathsf{t} \to \mathsf{t}')^l \qquad \Gamma \vdash_{ML} \mathsf{e}_2 : \mathsf{t} \qquad l \lhd \mathsf{t}'}{\Gamma \vdash_{ML} \mathsf{e}_1\mathsf{e}_2 : \mathsf{t}'} \text{ (ML-APP)}$$

$$\frac{\Gamma \vdash_{ML} \mathsf{e} : \mathsf{t}' \qquad \mathsf{t}' \le \mathsf{t}}{\Gamma \vdash_{ML} \mathsf{e} : \mathsf{t}} \text{ (ML-SUB)}$$

$$t \le t \text{ (SUB-ID)} \qquad \frac{t_1' \le t_1 \qquad t_2 \le t_2' \qquad l \sqsubseteq l'}{(t_1 \to t_2)^l \le (t_1' \to t_2')^{l'}} \text{ (SUB-FN)}$$

Figure 16. Core-ML syntax and typing (functional fragment).

References

[1] C. Andreae, J. Noble, S. Markstrum, and T. Millstein. A framework for implementing pluggable type systems. In *OOPSLA '06*. ACM Press.

[2] L. Augustsson. Cayenne–a language with dependent types. In *ICFP '98*. ACM Press.

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

[4] D. Brumley and D. Boneh. Remote timing attacks are practical. In *USENIX SSYM'03*.

[5] B. Chin, S. Markstrum, and T. Millstein. Semantic type qualifiers. In *PLDI '05*. ACM Press.

[6] 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.

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

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

[9] T. Fraser. LOMAC: Low Water-Mark Integrity Protection for COTS Environments. In *IEEE S&P*, 2000.

[10] T. Fraser, J. Nick L. Petroni, and W. A. Arbaugh. Applying flow-sensitive CQUAL to verify MINIX authorization check placement. In *PLAS '06*. ACM Press.

[11] R. Harper and G. Morrisett. Compiling polymorphism using intensional type analysis. In *POPL '95*. ACM Press.

[12] F. Henglein. Type inference and semi-unification. In *LFP '88*. ACM Press.

[13] B. Hicks, D. King, P. McDaniel, and M. Hicks. Trusted declassification:: high-level policy for a security-typed language. In *PLAS '06*. ACM Press.

[14] B. Hicks, S. Rueda, T. Jaeger, and P. McDaniel. From trusted to secure: Building and executing applications that enforce system security. In *USENIX*, 2007.

[15] P. Kocher, J. Jaffe, and B. Jun. Differential power analysis. *LNCS*, 1666:388–397, 1999.

[16] P. Li and S. Zdancewic. Encoding information flow in Haskell. In *CSFW '06*. IEEE Computer Society.

[17] D. Marino, B. Chin, T. Millstein, G. Tan, R. J. Simmons, and D. Walker. Mechanized metatheory for user-defined type extensions. In *WMM '06*.

[18] A. C. Myers and B. Liskov. Protecting privacy using the decentralized label model. *ACM TOSEM*, 9(4):410–442, 2000.

[19] OWASP. Top-10 web application vulnerabilities. `http://www.owasp.org/index.php/Top_10_2007`.

[20] M. Pistoia, A. Banerjee, and D. A. Naumann. Beyond stack inspection: A unified access-control and information-flow security model. In *S&P '07*. IEEE Computer Society.

[21] F. Pottier and V. Simonet. Information flow inference for ML. *ACM*

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

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

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

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

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

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

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

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

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

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

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

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

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

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

Figure 17. Translation from a Core-ML derivation $\mathscr{D}$ to FABLECORE.

(TOPLAS), 25(1), 2003.

[22] A. Sabelfeld and A. C. Myers. Language-based information-flow security. *IEEE JSAC*, 21(1):5–19, 2003.

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

[24] P. Sewell, G. Stoyle, M. Hicks, G. Bierman, and K. Wansbrough. Dynamic rebinding for marshalling and update, via redex-time and destruct-time reduction. *JFP*, 2007.

[25] N. Swamy and M. Hicks. Fable: A language for enforcing user-defined security policies (extended version). University of Maryland, Technical Report, CS-TR-4876; `http://www.cs.umd.edu/projects/PL/fable/TR.pdf`.

[26] N. Swamy, M. Hicks, S. Tse, and S. Zdancewic. Managing policy updates in security-typed languages. In *CSFW '06*.

[27] D. Walker. A type system for expressive security policies. In *POPL '00*. ACM Press.

[28] D. Walker. *Advanced Topics in Types and Programming Languages; B. Pierce ed.*, chapter Substructural Type Systems. MIT Press, 2004.

[29] G. Wassermann and Z. Su. Sound and precise analysis of web applications for injection vulnerabilities. *PLDI '07*.

[30] H. Xi. Applied Type System (extended abstract). In *TYPES 2003*. Springer-Verlag LNCS 3085, 2004.

[31] H. Xi and F. Pfenning. Dependent types in practical programming. In *POPL'99*. ACM Press.

[32] X. Zhang, A. Edwards, and T. Jaeger. Using CQUAL for static analysis of authorization hook placement. In *USENIX Security '02*.

[33] L. Zheng and A. C. Myers. Dynamic security labels and noninterference. In *FAST '04*.