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 is 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 $l \subseteq h$ 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 choose 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...
with a particular choice of enforcement mechanism, to establish that a system \textit{implementation} complies with stated high-level security objectives.

### 1.1 Enforcing user-defined security policies

This paper presents \textsc{Fable}, a core formalism for a programming language that permits programmers to specify custom security policies and the semantics of their enforcement. \textsc{Fable} has two key elements. First, security policies are associated with program data and actions \textit{via security labels}. This association is expressed \textit{via} a dependent type \( i \cdot e \) where \( e \) is a term of type \texttt{lab} whose values \( i \) are essentially uninterpreted constructor applications of form \( C \langle \cdot \rangle \). For example, \( 0\)-ary constructors \texttt{LOW} and \texttt{HIGH} might be used as information flow labels, while the label on the type \texttt{int\{ACL\(\langle\cdot\rangle\)}} might represent an access control policy in which only \texttt{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 \textit{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 \texttt{send} having the type \texttt{socket \[\rightarrow\] bytes \[\rightarrow\] unit}. The following code snippet shows a simple enforcement policy that restricts what data might be sent on a socket via \texttt{send}.

\begin{verbatim}
letpol sock_send_dynamic
\lambda:l:lab.\lambda:s:socket\{\}.\lambda:m:lab.\lambda:msg:bytes\{m\}. if policy_start m then
send <socket>s <bytes>msg else ()
\end{verbatim}

Here, \texttt{sock_send_dynamic} wraps the \texttt{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 \( i \) where the next argument has a dependent type \texttt{socket\{\}}. This indicates that the socket is protected by policy label \( i \). Similarly, the third and fourth arguments are a label term \( m \) and the bytes of a message \texttt{msg} with the dependent type \texttt{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 \texttt{send} function takes an argument of type \texttt{socket\{\}}, not \texttt{socket\{\}}. There is likewise no way to send a protected message on an unprotected socket.

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

In the above example, the label \( i \) in the type \texttt{socket\{\}} is used by the program at run time. However, \textsc{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.

\begin{verbatim}
letpol sock_send_static
\phi:m.\lambda:s:socket\{\texttt{ALLOW}\{m\}\}.\lambda:msg:bytes\{m\}. send <socket>s <bytes>msg
\end{verbatim}

Here, the policy designer chooses the form of labels that protect sockets to be of the form \texttt{ALLOW\{\}} indicating that messages with security label \( i \) 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 \( \phi \) binds a \texttt{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 \texttt{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 \texttt{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.

\begin{verbatim}
letpol add \phi:l:lab.\lambda:x:int\{\}.\phi:m.\lambda:y:int\{m\}. \\
<\texttt{int}\langle\texttt{lab}\{l\}\rangle + \langle\texttt{int}\rangle> + \langle\texttt{int}\rangle>y)
\end{verbatim}

Here the label of the sum of the two integers is computed by calling the policy function \texttt{lab} (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 \( x : \texttt{int\{LOW\}} \) and \( y : \texttt{int\{HIGH\}} \) then a call to \texttt{add} with \( x \) and \( y \) as arguments will be given type \texttt{int\langle\texttt{lab\{LOW\{HIGH\}\}}\rangle}. The type checker can try to reduce the expression \texttt{lab\{LOW\{HIGH\}\} if necessary (in this case, to the value \texttt{HIGH\}). This makes type checking undecidable in general, but the added flexibility is quite useful. Furthermore, since \textsc{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 \textsc{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 \textsc{Fable}, examples of its use, and some of its formal properties, making the following contributions:

- \textsc{Fable} is a new dependently-typed core language for defining and enforcing user-defined security policies (Section 2). Among other novel features, \textsc{Fable} introduces the concept of an enforcement policy as the mechanism for implementing a traditional security policy, where \textsc{Fable} makes enforcement policies explicit in the program. We have proven that \textsc{Fable} is sound (Section 4.3).

- \textsc{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 automation-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.

- \textsc{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.
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 \( \bar{a} \) to stand for a list of elements of the form \( a_1, \ldots, a_n \), where the context is clear, we will also treat \( \bar{a} \) to be the set of elements \( \{a_1, \ldots, a_n\} \).

Programs \( P \) consist of a policy \( \pi \) and a program term \( e \). The policy consists of a list of definitions `letpol` \( x \mid \pi \), which bind variables \( \pi \) to values \( v \), which are a subcategory of expressions \( e \). Standard expression forms include unit \( () \), variables \( x \), application \( e_1 \, e_2 \), and fixpoint combinator \( \text{fix } \, x \, e \). The syntax of type expressions is similar, using the metavariables \( \Lambda, \alpha, \beta, \gamma, \... \).

Term abstraction \( \text{let } \, x \, = \, \pi \, \text{in } \, \lambda \, : \, \alpha \, \text{. } \, x \, = \, \gamma \) binds two kinds of variables: the \( \lambda \)-bound variable \( x \) is standard, while \( \varphi \)-prefixed list \( \gamma \) 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. Application \( e_1 \, e_2 \), 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 \( a_1 \rightarrow e_1 \) \( \mid \ldots \mid \) \( a_n \rightarrow e_n \)` evaluates to \( e \) if \( x \) matches the pattern `JOIN(a1,a2)`, where occurrences of \( a_1 \) and \( a_2 \) are substituted in \( e \) with the corresponding components in \( x \)'s run-time value. Pattern variables like \( a_1 \) and \( a_2 \) are explicitly bound in pattern clauses to simplify type checking.

Finally, as mentioned earlier, a type may also have an associated label, written \( \lambda \). Using these constructs we can define the type `\( \varphi \rightarrow \gamma \)`.

Figure 1. Syntax of FABLECORE.

Figure 2. Enforcing a simple access control policy.
access
lab
unit
member
Typing \( \phi \) tok \( \times \) \( \lambda \) lab label from its type.

login

k
\( \lambda \) function to see whether the \( \phi \) may only appear only in MEMBER with type. Checking the body \( \phi \) function the type \( \rightarrow \lambda \) string lab cap and \( \lambda \) we must show that unit as the arguments. To type check the \( \phi \) access unit tok USER \( \) with

Checking a single policy binding the context that results from checking all the policy expressions. As \( \Omega \) \( \epsilon \) \( \alpha \) \( \beta \) \( \gamma \) not overlap with labels from other policies in force in the same program. As

One assumption made here is that the labels returned by checkpkg 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.

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 \( \Omega \). The typing context \( \Omega \) consists of three components: \( \Gamma \), an environment that binds variables to types and type variables to kinds; a policy \( \pi \); 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 \( \Gamma \)—the form \( x : t \) records the type of a \( \lambda \)-bound variable; \( x : t \) records the type of a policy global variable; and \( \phi x : t \) records the type of a \( \phi \)-bound variable (in FABLECORE this is always lab). The semantics ensure that \( \phi \)-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 \( \Omega \) with the binding \( x : t \) added to its \( \Gamma \) component; similarly \( \Omega[\pi] \) is the context \( \Omega \) with the policy \( \pi \) added to its policy component. We will also use \( \Omega, \Gamma \) etc. to project components from \( \Omega \) and \( \Omega[\Gamma = \Gamma'] \) to denote the context \( \Omega \) with its \( \Gamma \) component replaced with \( \Gamma' \).

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 \( x \) \( \Rightarrow \) \( x \) v occurs in (T-LETPOL). It types the remainder of the policy in a context whose \( \Gamma \) component records the type of \( x \) and whose \( \pi \) component includes the definition itself.

The main judgment \( \Omega \vdash e : t \) types expressions. The index \( e \) 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 \( \beta \)-bindings) can safely use \( \phi \)-bound variables; term-level expressions cannot. Rule (T-VAR) allows \( \lambda \)-bound variables in either phase, while (T-LVAR) allows \( \phi \)-bound variables only in the \( \phi \) (type-level) phase. In the access function of the example, phantom variable \( k \) may only appear only in type-expressions (such as \( \text{lab} \rightarrow \text{USER}(k) \)) while \( u \) may appear both in type-level expressions (such as \( \text{unit}(u) \) in the second argument) and in the body of the function.

(T-LAB) gives a label term \( C(\bar{e}) \) a singleton label type \( \text{lab} \rightarrow C(\bar{e}) \) as long as each component \( e_i \in \bar{e} \) is \( \text{lab} \)-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 \( \lambda \)-bound variable \( x \) to be fresh. The second premise ensures that all \( \phi \)-bound variables \( \bar{y} \) appear in the argument’s type \( t \), and are not already mentioned in \( \Gamma \). The third premise checks \( t \) in a context with phantom variables \( \bar{y} \) each given lab type. Checking the body \( e \) extends the context further to include the \( \lambda \)-bound variable \( x \).

The first premise of (T-APP) requires \( e_1 \) to have an unlabaled function type. The third premise of (T-APP) checks that the type of \( \bar{e}_2 \) of argument \( e_2 \) can be unified with \( x \)’s type \( t_1 \), producing a substitution of the phantom variables \( \bar{y} \) (we explain the rules for unification below). For example, when typing the application access USER(Joe), the judgment \( k : \Omega.A \vdash \text{lab} \rightarrow \text{USER}(\text{Joe}) \leq \text{lab} \rightarrow \text{USER}(\text{Joe}) : \sigma \), produces \( \sigma = (k : \rightarrow \text{Joe}) \) as the result. We apply the substitution \( \sigma \), augmented with the substitution for the \( \lambda \)-bound variable, to the return type. In our example, the type of the application would be unit \( \text{USER}(\text{Joe}) \rightarrow \alpha \left( \text{lab} \right) \rightarrow \alpha \left( \text{acl} \right) \rightarrow \alpha \).

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 \( x \_\text{default} = \_\text{default} \rightarrow e_n \). 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:

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

We give the login function the type string → string → (\( (\text{lab} \times \text{unit}()) \)) where \( \text{lab} \times \text{unit}() \) is a dependent product type. \( \text{lab} \times \text{unit}() \) and a unit with labeled type unit() as the arguments. To type check the application access tok we must show that \( \text{lab} \rightarrow \text{tok} \), 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 \( \text{tok} \leq \text{USER}(j) \). 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 \( \phi \). 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 \( \pi \vdash t \rightarrow t' \) simply lifts the standard operational semantics on expressions \( \pi \vdash e \rightarrow e' \) to types. This reslation relies on the definition of the type evaluation context \( T \), which contains two kinds of hole \( \ast \) and \( \ast \), which can be filled with types and expressions respectively.

The judgment \( \Omega \vdash t : \ast \) assigns a kind to a type. (K-SUB) allows a type of \( \text{U} \) kind to be given kind \( \text{M} \). (K-SEC) gives kind \( \text{M} \) a labeled type. Notice that in K-SEC the label is checked in \( \phi \)-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 \( \lambda \)-bound variable is in scope in \( t_2 \) only.

The judgment \( \Sigma : \lambda t_1 \leq t_2 : \sigma \) states that in a context where \( \Sigma \) 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 \( \sigma \) where \( \text{dom}(\sigma) = \Sigma \). This is a highly-restricted form of semi-unification [12]; the substitution \( \sigma \) 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
### Syntactic forms used in judgments

<table>
<thead>
<tr>
<th>Form</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>( \Gamma ::= x : t</td>
<td>x : t</td>
</tr>
<tr>
<td>( A ::= e_1 \leq e_2</td>
<td>A_1, A_2 )</td>
</tr>
<tr>
<td>( \Omega ::= x : \varphi</td>
<td>\varphi )</td>
</tr>
<tr>
<td>( E ::= \bullet</td>
<td>e</td>
</tr>
<tr>
<td>( E ::= \bullet</td>
<td>e</td>
</tr>
</tbody>
</table>

### Top-level judgments for typing programs and policies

<table>
<thead>
<tr>
<th>Form</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>( \Omega \vdash E : t )</td>
<td>Expression ( e ) has type ( t ) in environment ( \Omega )</td>
</tr>
</tbody>
</table>

### Type \( t \) has kind \( \kappa \) in environment \( \Omega \)

<table>
<thead>
<tr>
<th>Form</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>( \Omega ::= \bullet</td>
<td>e</td>
</tr>
</tbody>
</table>

### Small-step typed reduction rules

<table>
<thead>
<tr>
<th>Form</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>( \pi ::= e \rightarrow e' )</td>
<td>Evaluation relation on terms lifted to types</td>
</tr>
</tbody>
</table>

### Figure 3. Semantics of FABLECORE.
the bare types. (U-FUN) is similar to (U-UNI) in that the bound phantom variables are renamed (by σ₁) in t₁ and in t₂. Prior to unifying t₁ with t₂, the third premise requires all free variables in t₁ also in t₁ to be substituted using the substitution σ₁ produced by unifying t₁ and t₁ (U-SEC) is similar to (U-SEC).

The judgment x₁;x₂:A ⊢ e₁ ≤ e₂ : σ is more general than the corresponding judgment for types. Here, given x₁ the free variables in e₁ and x₂ the free variables in e₂ and a set of assumptions A, σ is a substitution where dom(σ) ⊆ x₁ ∪ x₂ such that σ(e₁) = σ(e₂). When attempting to unify types t₁ and t₂, this judgment is always used in the form where x₁ 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 if 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 tok ≤ USER(k) in a context that includes tok ≤ USER(r) as an assumption, it suffices to unify USER(r) ≤ USER(k): (k ↦ 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 p ⊢ e → e′ using evaluation contexts E, where the hole is written •. Context application is written with a dot, e.g., E • 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 p ⊢ t → 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 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 e is lab ⊑ LOW, then we can use this information to decide which branch to take. Additionally, we can use the context ΩA 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 ⊑ 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: τ₁ ≤ τ₂ ⇐⇒ τ₁ ⊑ τ₂. 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

\[
\begin{align*}
\text{letpol } \text{lub} & \triangleq \text{lub} \ L M \text{ lab.match}\ \ell, m \text{ with} \\
& 1, 1 \rightarrow \text{LOW, MED} \rightarrow \text{MED} \\
& \text{MED, HIGH} \rightarrow \text{HIGH} \\
\text{letpol } f & \triangleq \text{x:int}\{\}\rightarrow \text{int}\{(\text{lub} \ \text{LOW})\} \\
\text{One type of the function } f & \text{ is } t_1 = \text{x:int}\{\}\rightarrow \text{int}\{(\text{lub} \ \text{LOW})\}. \text{ We permit the type checker to attempt to reduce } \text{lub} \ \text{LOW}, \text{ even though it contains a free variable } l. \text{ If the type checker is able to reduce this using } \pi \vdash \text{ lub} \ \text{LOW} \rightarrow e, \text{ then it can also give } f \text{ the type } t_2 = \text{x:int}\{\}\rightarrow \text{int}\{(e)\}. \text{ Now consider a usage of this function } f \{(\text{int}(\text{MED}))\}. \text{ The type given to this expression should not depend on the choice of } t_1 \text{ or } t_2 \text{ as the type of } f. \text{ That is, given } \sigma = (l \mapsto \text{MED}) \text{ we require,}
\end{align*}
\]

\[
\pi \vdash \sigma(\text{lub} \ \text{LOW}) \rightarrow^* e' \iff \pi \vdash \sigma(e) \rightarrow^* e'
\]

To enforce this property, we must take care when matching label terms containing free variables. In our example, attempting to reduce \text{lub} \ \text{LOW} requires matching \text{l} against \text{LOW}, or against \text{MED} etc. Since the value of \text{l} varies depending on the context in which the function \( f \) is used (our example usage gives the \text{l} the value MED, but other usages might give other values to \text{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: \text{\varphi} \triangleq \text{x:int}\{\}\rightarrow \text{add}\ \text{x}. This function doubles its argument \text{x} using the \text{add} function defined in Section 1. We can type \text{add}\ \text{x} as int\{(\text{lub} \ 1)\}. However, since \text{lub} defines a lattice, we would like to be able to reduce this to just \text{l}. When attempting to reduce \text{lub} \ 1, it is easy to see that the first pattern in the definition of \text{lub} will always be matched, irrespective of the substitution chosen for \text{l}. Thus, we can permit the reduction \( \pi \vdash \text{lub} \ \text{LOW} \rightarrow^* l \).

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 \( l \) instantiates \text{l} to MED, then we can proceed with reduction on this assumption for \text{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 \text{l} is lab \rightarrow \text{LOW}, then we can use this information to decide which branch to take. Additionally, we can use the context \( ΩA \) 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.
letpol \lambda\cdot x:\text{lab.} \lambda\cdot y:\text{lab.} \text{match} x, y \text{ with } \\
\text{LOW}, \_ \rightarrow y \\
\text{MED}, \_ \rightarrow x \\
\text{MED}, y \rightarrow \_ \\
\_ \rightarrow \_ \\

letpol \lambda a:x:a,.<\alpha\{\text{LOW}\} > x \\
letpol \lambda a:\text{pl},m:x:a\{\_\}\{\_\},.<\alpha\{\text{lub}\ lamb\}m > x \\
letpol \lambda a:\text{pl},l:x:a\{\_\},.<\alpha\{\text{lub]\ lamb\}m > x \\
letpol \lambda a:pl,\lambda\cdot x:a\{\_\},m:<\alpha\{\text{lub}\ lamb\}m > x \\
letpol \lambda a:pl,\lambda\cdot x:a\{\_\},m:<\alpha\{\text{lub}\ lamb\}m > x \\

let tmp = app \alpha [a] \rightarrow b \rightarrow x \rightarrow \text{in} \\
app \alpha [a] \rightarrow \text{tmp} y \\

\text{Figure 4. Enforcing a static information flow policy.} \\

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. 

\text{Theorem} (Noninterference). Given \pi, the policy of Figure 4, and an expression e that does not contain any relabeling operations, and an empty context \Omega (\Omega \Gamma = \Omega = \pi = \Omega \Lambda = \cdot) such that \Omega [x: t \{\text{HIGH}\}] \vdash \pi ; e \vdash t \{\text{LOW}\}, where \pi \text{ is of } U\text{-kind; and values } v_1 \text{ and } v_2: \\
(\forall i \in \{1, 2\}, \Omega \vdash \pi ; v_i: t \{\text{HIGH}\} \land \pi \vdash (x \rightarrow v_i) e \rightarrow \_ v_i) \Rightarrow v'_1 = v'_2 \\

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

\begin{align*} \\
\Omega \vdash e_1 : t \{\text{HIGH}\} & \quad \Omega \vdash e_2 : t \{\text{HIGH}\} \\
\Omega \vdash e_1 \parallel e_2 : t \{\text{HIGH}\} & \quad (\text{T-BRACKET}) \\
\end{align*} \\

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 proceeds by a straightforward since reduction within each side of the bracket obeys the normal typing and evaluation rules of \text{FABLE\text{CORE}}. This allows us to use the subject reduction property of \text{FABLE\text{CORE}} 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 \text{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 straightforward from Core-ML typing derivations to \text{FABLE\text{CORE}} programs. The main typing judgment in Core-ML is of the form \mathcal{L}; \Gamma \vdash t : \text{ML}; e : t. The judgment states that the Core-ML expression e has type t in a context where \mathcal{L} is the lattice of security labels, and \Gamma is the usual context associating variables with their types. We define a family of translations, \{\mathcal{T}_{\mathcal{L}}\}, that relate Core-ML types, contexts and derivations to \text{FABLE\text{CORE}} 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 \lambda\cdot terms in Core-ML by inserting the appropriate calls to sub; this allows \text{FABLE\text{CORE}} function types to be, as usual, contravariant in the labels of their arguments and covariant in their return type. 

\text{Theorem} (Expressiveness of static information flow). Given the policy \pi of Figure 4, a lattice \mathcal{L} equivalent to the function lub of \pi, and a Core-ML derivation \mathcal{D} such that, \mathcal{D} = (\mathcal{L}; \Gamma \vdash t : \text{ML}; e : t); then \mathcal{T}_{\mathcal{L}}[\mathcal{D}] : \vdash \pi; [\mathcal{T}_{\mathcal{L}}]: [\mathbf{t}]. 

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

\[
\text{flow} : \lambda \text{src} . \lambda \text{dest} . \text{unit} \times \{\text{FLOW}(\text{src}, \text{dest})\}.
\]

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 \(\alpha(\text{src})\) may be coerced to type \(\alpha(\text{dest})\). 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 ly for x and y 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 \(\text{JOIN}(\text{b}, \text{y})\).

The first three lines of the main expression use the flow function to attempt to obtain capabilities that witness the flow from b to x and ly to \(\text{JOIN}(\text{b}, \text{l})\). The match inspects the labels that are returned by \text{flow} and in case where they are actually \text{FLOW}(\_\_\_\_), the final premise of (T-MATCH) permits the type of \(\text{fx}\) to be refined from \(\text{lab} \sim \text{fl} \to \text{lab} \sim \text{FLOW}(\text{b}, \text{l})\) and the type of \text{capx} to be refined to \(\text{unit}(\text{FLOW}(\text{b}, \text{l}))\), and similarly for \text{capy}. The remainder of the program is similar to the static case, but requires more uses of

\[
\text{letpol flow A src lab A dest lab \ldots}
\]

\[
\text{let f = if oracle src dest then FLOW(src, dest) else NOFLOW in}
\]

\[
(f, \text{<unit}(f)>)(\ldots)
\]
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 FABLECORE we might try to write a policy function that permits assignment through a labeled reference. It might look something like the following:

```plaintext
letpol update $\alpha \cdot \phi \cdot \lambda \cdot \alpha \cdot \text{ref}(!) \cdot \lambda \cdot \alpha$. 
\begin{align*}
\text{<unit} & \langle \text{CLOW} \cdot \text{EFFECT}(t) \rangle \triangleright (\langle \alpha \cdot \text{ref} \rangle \cdot x) := y \\
\end{align*}
```

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:

```plaintext
\phi \cdot \lambda \cdot x \cdot \text{int} \cdot \text{ref}(!). \text{let} \_ = \text{update} [\text{int}] \times 0 \text{ in } 0
```

This function has the type $(\lambda \cdot x \cdot \text{int} \cdot \text{ref}(!) \rightarrow \text{int})$. 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 $C(e)$ and, as previously, only the policy may associate such labels with program terms. A relevant label can be given the type $\lambda \cdot \text{lab}$, 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 $\text{unit} \cdot \langle \text{CLOW} \cdot \text{EFFECT}(t) \rangle$) 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 $C(e)$. Next, abstractions in FABLE now allow abstraction over both relevant and normal phantom labels. For example, in $\phi \cdot \lambda \cdot x \cdot \text{ref}(!) \cdot \text{e}$, $l$ 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 $\text{t}([\text{HIGH}])$, or $\text{t}([\text{HIGH}])$. In contrast, it would be illegal to pass a term with type $\text{t}([\text{HIGH}])$ to the function $\text{q} \cdot \text{m} \cdot \lambda \cdot \alpha \cdot \text{e} \cdot \text{e}$. 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 $s_x$ as values, and the standard dereferencing and assignment constructs. Note that for simplicity we do not model dynamic allocation; accounting for allocation effects poses an additional technical challenge.

The type language of FABLE now includes relevant types and reference types. For any type $t$, $\alpha_t$ is a relevant type. In addition, a type such as $\alpha_t(e)$ 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 unlabelled types, and $M$ for types that may be labeled. Kind $R$ includes both types such as $\text{lab}$ as well as $\alpha_t([H])$, i.e., both unlabelled and labeled types. This induces a convenient sub-kinding relation $U < M < R$. Finally, the typing environment $\Omega$ includes a $R$ component that records the set of relevant assumptions in $\Omega \cdot \Gamma$ that must be discharged by an expression.

The remainder of Figure 7 is divided into three sections showing selected judgments from the semantics of FABLECORE. 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 $\Gamma$. However, in our case this can be problematic due to type-level dependencies. For instance, given $\Gamma = \gamma \cdot \text{lab} \cdot x : t(y)$, if we split this as $\Gamma = \{ \Gamma_1 = x : t(y) \} \oplus \{ \Gamma_2 = y : \text{lab} \}$, then we are left with an ill-formed environment $\Gamma_1$ since the label name $y$ appears in the type of $x$, but $y$ is free in $\Gamma_1$. To avoid such complications, we maintain a separate context $R$ of relevant assumptions in $\Gamma$ and never split $\Gamma$ itself.

The judgment $\Omega \vdash e : t$ expressions similarly to the corresponding judgment in FABLECORE, but must track relevant assumptions. The first rule (T-SUBR) states that any type $\tau$ may be treated as a relevant type $\alpha_t$. The rule (T-VAR) is only applicable when $(\Omega \cdot R = \emptyset)$; 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-VARp) that allows a $\phi$-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 $\phi$-bound variables. However, here, the $\phi$-bound variables may include some relevant variables $\bar{z}$. When checking the ascribed type $t_1$ and the body of the abstraction, we must record the relevant variables $\bar{z}$ as having type $\lambda \cdot \text{lab}$ 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 $(\Omega \cdot R \neq \emptyset)$ 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 $\bar{z} ; \Omega \vdash t_1 \leq t_2 : \alpha$ 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
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 destroyed by the policy. The first premise of (T-PFX) uses the judgment $\Omega, x : t \vdash e : t$ defined by the rule (ESC). In the first premise of (ESC), relevant($\Omega, t$) stands for all assumptions $x : t$ in $\Omega$ where $t$ 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 kind 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 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-VAR$\phi$) 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 completely 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 dom($\Omega, t$) = dom($M$); for each location in the store $M$ to be given a reference type in $\Omega$; and for all values in the store to be well-typed closed terms. This is the content of the judgment $\Omega \vdash M$. 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 $\Omega, t : \cdots : \alpha$ and memory $M$ such that $\Omega \vdash M$, and $\pi; e$ such that $\Omega, t : \pi; e$, then either $\pi \vdash (M, e) \leadsto (M', e')$ for some $M'$, $e'$ or $e$ is a value.

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

---

**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.)

---

**Figure 8.** Tracking effects using FABLE.
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 \( C(i, l) \), where \( i \) describes the standard (confidentiality) label of the value, and \( l \) 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 \( \text{unit}(\text{C(Low,Effect}(i))) \) so that the program cannot ignore the returned value. Figure 8 shows the full policy.

To apply 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 \( \alpha \) and \( \beta \) 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 i 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 \( \forall t \) 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 \( \text{state} \rightarrow e \). Since \( \text{state} \rightarrow e \) 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 \( \text{state} \rightarrow e \rightarrow \alpha \rightarrow \beta \); such a function requires the current state as its first argument; furthermore, the type expresses a precondition that the current state be described by the expression \( e \). The FABLE type checker can ensure that a function with such a type is only called in states that satisfy the state precondition.

Figure 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 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` 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 \( f \) (such as `Read` or `Send`) followed by the function \( f \) itself. The last two arguments are \( x \) and \( s \), 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 \( \Omega \cdot A \) 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-

---

\( \text{abbrv} \text{state} \rightarrow e = \forall t. \text{lab} \rightarrow e \times \text{unit}(\{t\}) \)

\( \text{let polym delta } \phi, \lambda s. \text{state} \rightarrow \lambda f. \text{lab}. \lambda x. \text{lab}. \)

\( \text{let } t = \text{match } (\text{fst } s, f) \times \text{with} \)

\( \text{State}(\text{Start}, \ldots), \text{Read}, \text{High} \rightarrow \text{State}(\text{Has_read}, f, x) \)

\( \text{State}(\text{Start}, \ldots) \rightarrow \text{State}(\text{Start}, f, x) \)

\( \text{State}(\text{Has_read}, \ldots), \text{Send}, \ldots \rightarrow \text{BAD} \)

\( \text{State}(\text{Has_read}, \ldots), \ldots \rightarrow \text{State}(\text{Has_read}, f, x) \)

\( \ldots \rightarrow \text{BAD} \)

\( \ldots \rightarrow \text{BAD} \)

\( t \mapsto t \mapsto t; \text{ i.e., side effects cannot occur in expressions that appear in types.} \)
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 are 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.

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

$\Gamma = \Gamma_1, x : t, \Gamma_2$ 
$\exists \kappa. \Omega[\Gamma = \Gamma_1, \Gamma_2] \vdash t : \kappa$

$\Gamma = \Gamma_1, \forall x : t, \Gamma_2$ 
$\exists \Gamma. \Gamma[\Gamma = \Gamma_1, \Gamma_2] \vdash \forall x : t, \Gamma_2$

$\Gamma = \Gamma_1, \exists x : t, \Gamma_2$ 
$\exists \kappa, \Omega[\Gamma = \Gamma_1, \Gamma_2] \vdash \exists x : t, \Gamma_2$

$\Gamma = \Gamma_1, \forall x : t, \Gamma_2$ 
$\exists \kappa, \Omega[\Gamma = \Gamma_1, \Gamma_2] \vdash \forall x : t, \Gamma_2$

$\Gamma = \Gamma_1, \exists x : t, \Gamma_2$ 
$\exists \kappa, \Omega[\Gamma = \Gamma_1, \Gamma_2] \vdash \exists x : t, \Gamma_2$

$\Gamma = \Gamma_1, \forall x : t, \Gamma_2$ 
$\exists \kappa, \Omega[\Gamma = \Gamma_1, \Gamma_2] \vdash \forall x : t, \Gamma_2$

$\Gamma = \Gamma_1, \exists x : t, \Gamma_2$ 
$\exists \kappa, \Omega[\Gamma = \Gamma_1, \Gamma_2] \vdash \exists x : t, \Gamma_2$

$\Gamma = \Gamma_1, \forall x : t, \Gamma_2$ 
$\exists \kappa, \Omega[\Gamma = \Gamma_1, \Gamma_2] \vdash \forall x : t, \Gamma_2$
Given \( \Gamma = \{ \ldots \} \) where \( \Gamma \vdash t : M \) and memory \( M \) such that \( \Omega \vdash M \); then either \( \exists \tau. \pi \vdash e \rightsquigarrow e' \) or \( \exists \omega. e = v \).

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

---

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

1. \( \text{dom}(\Omega, \Gamma) = \text{dom}(M) \)
2. \( \Omega \vdash t : M \) and \( \Gamma \vdash t : M \) for each \( t \) in the store.
3. \( \text{range}(M) = \{ \psi, \ldots \} \)
4. \( \Omega(x) \in \{ t \} \rightarrow \Omega(x) \subseteq M \)

**Theorem (Progress).** Given \( \Omega = \{ \ldots \} \), \( \Omega \vdash \pi.e : t \), and memory \( M \) such that \( \Omega \vdash M \); then either \( \exists \tau. \pi \vdash e \rightsquigarrow e' \) or \( \exists \omega. e = v \).

---

**Figure 11.** Static semantics of FABLE (Part 1).
Figure 12. Static semantics of FABLE (Part 2).

Additional syntactic forms used in judgments

| $M$ | ::= | $(\ell,v) | M_1, M_2$ |
| $E$ | ::= | $\cdot | e | v | \ell | \bar{c} | \bar{e} | \text{match } \text{with } x.p_i \rightarrow e | \langle \ell \rangle | | C(\bar{v}, \bar{e}) | | \cdot | \cdot ::= e | v ::= \cdot |

| $\pi \vdash e \rightsquigarrow e'$ | \text{Evaluation contexts} |

Small-step typed reduction rules

\[
\begin{align*}
\pi \vdash e & \rightsquigarrow e' \\
\pi \vdash (M,e) & \rightsquigarrow (M',e') \quad \text{(E-CTX)} \\
\pi \vdash (M,E,e) & \rightsquigarrow (M,E,e') \quad \text{(E-CTX)} \\
\pi \vdash \Lambda \alpha : \kappa.e \ [t] & \rightsquigarrow (\alpha \rightsquigarrow t)e \quad \text{(E-TAP)} \\
\pi \vdash \langle \ell \rangle(u) & \rightsquigarrow (\ell)u \quad \text{(E-LAB)} \\
\pi \vdash \text{match } v \text{ with } x.p_i & \rightarrow e_i \quad \text{(E-MATCH)} \\
\pi \vdash t & \rightsquigarrow t' \\
\end{align*}
\]

Evaluation relation on terms lifted to types

\[
\begin{align*}
\pi \vdash e & \rightsquigarrow e' \\
\pi \vdash T.e & \rightsquigarrow T'.e' \quad \text{(TE-CTXE1)} \\
\pi \vdash T & \rightsquigarrow T' \\
\end{align*}
\]
CASE (T-UNIT): {} is a value.

CASE (T-L1): e is either C(\vec{v}, e, \vec{e}) or \{C(\vec{v}, e, \vec{e})\}, both of which are legal evaluation contexts. By the induction hypothesis \pi \vdash e \leadsto 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 = e' [t] or v[t]. In the first case, we use the induction hypothesis and apply (E-CTX) to show that \pi \vdash (M, e [t]) \leadsto (M', e' [t]). In the second case, by second premise of (T-TAP) we have \Omega \vdash v : \forall \alpha : \kappa.e. By enumeration of the possible syntactic forms of \nu we find that this premise must be an either an application of the (T-TAB) rule and \nu = \lambda \alpha : \kappa.e and (T-TAB) is applicable. (The possibility of \nu being a labeled pre-value \langle i \rangle u is ruled out, since we syntactically require \nu to be a labeled type; \nu : \alpha : \kappa.e is not a labeled type.)

CASE (T-APP): If \nu is either e_1 \nu_2 or v \sigma \cdot \nu_2, then, by applying the induction hypothesis to the second and third premises respectively, we get our result using (E-CTX). If, \nu = \nu_2 \nu_3 then, we can conclude that the second premise of (T-APP) must be an application of the (T-ABS) rule and \nu_1 = \forall \vec{x} : t.e. Following an argument similar to (T-TAP), we have that (E-APP) is applicable and \pi \vdash \nu_2 \nu_3 \leadsto (x \mapsto \nu_2, \sigma)e.

CASE (T-ASN): If we have e = e_1 := e_2 or \nu_1 := e_2, then by the induction hypothesis on the first and second premise respectively, we have our result via (E-CTX). If, however, e = \nu_1 \nu_2, then by assumption, dom(\Omega, \Gamma, \Pi) = dom(M) \cup \{x | \text{letpol } x \in \pi\}. Then, by \Omega \vdash M, M = M_1, (\cdot, v), M_2, satisfying the premise of (E-UPD) and enabling a reduction.

CASE (T-MATCH): If e = \text{match } \nu \cdot \text{with } \nu_1 : p_1 \rightarrow e_1, \ldots, \nu_n : p_n \rightarrow e_n \text{ then we must show that reduction via (E-MATCH) is applicable.} To establish this, note that the third premise of (T-MATCH) requires \langle \bar{b}_2, p_2 \rangle = \langle \bar{x}_{\text{def}}, x_{\text{def}} \rangle. Thus, it suffices to show that \nu \vdash x_{\text{def}} : \nu \mapsto x_{\text{def}} \mapsto v, for all closed labeled values \nu. As \Omega \vdash \nu : \text{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 = \nu' \text{ then, by the induction hypothesis on the premise } e \text{ can take a step via (E-CTX). If } e = \nu', \text{ then by an argument identical to (T-ASN), a step by (E-DRF) is possible.}

CASE (T-PFX), (T-SFX): If e = \langle i \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 i \rangle u, and t is a labeled type, then, e is a value.

If, \nu is an unlabeled type, then reduction can proceed via (E-LAB). What remains is e = \langle i \rangle \langle i \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 \Omega is well-formed, and \Omega \vdash \Pi : e : t contains a sub-derivation of the form \Omega' \vdash e : t \text{ or } \Omega' \vdash \nu : k \text{ then } \Omega' \text{ is well-formed.}

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

Lemma 5 (Expression unification). Given \langle \bar{b}_2 : \Omega \vdash e_1 \leq e_2 : \sigma \rangle such that following conditions hold:

(A1) \text{FV}(e_1) \setminus \text{dom}((\Omega, \Gamma)) = \emptyset \text{ and } \Omega \vdash e_1 : \text{lab}

(A2) \text{FV}(e_2) \setminus \text{dom}(\Omega, \Gamma) \subseteq \{x | x, jx \notin \bar{b}_2 \neq \emptyset\}

(A3) \{x | x, jx \notin \bar{b}_2 \neq \emptyset \} \cap \text{dom}(\Omega, \Gamma) = \emptyset

Then, the following are true:

i. \nu \vdash \text{dom}((\sigma)) \subseteq \{x | x, jx \notin \bar{b}_2 \neq \emptyset\}

ii. range(\sigma) = \{v_1, \ldots, v_n\}

iii. \nu \in \text{range}(\sigma) \Rightarrow \Omega \vdash \nu : \text{lab}

iv. (\text{FV}(e_2) \setminus \text{dom}(\Omega, \Gamma)) = \text{dom}(\sigma)

\nu. \Omega.A = \emptyset \Rightarrow \sigma(e_2) = e_1 \text{ (subject to } \alpha-\text{renaming of bound variables)}

Proof. Case (U-EID): By assumption (A1) we have \text{FV}(e) \setminus \text{dom}(\Gamma) = \emptyset = \text{dom}(\sigma) \subseteq \bar{b}_2. Proposition (v) is trivial.

Case (U-CON, U-RCON): We use the induction hypothesis for the first i − 1 premises and the assumption that the induction hypothesis is applicable to the ith premise. From assumption (A1) and proposition (i) we can conclude that \text{dom}(\sigma_i') \cap \text{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 \text{FV}(\sigma_i'(e_i')) \cap \text{dom}(\Omega, \Gamma) \subseteq \text{FV}(e_2) \setminus \text{dom}(\Omega, \Gamma), thus re-establishing (A2). (A3) is trivial, since \bar{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 \not\equiv j \Rightarrow \text{dom}(\sigma_i') \setminus \text{dom}(\sigma_j) = \emptyset and \text{dom}(\bar{\sigma}) = \bigcup \text{dom}(\bar{\sigma}). 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 \bar{b}_1 = \emptyset.

Case (U-VR1, U-VR2): For proposition (i), we use the first premise to confirm that x \in \bar{b}_2; proposition (ii) is immediate since \nu 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 \Omega (Definition 1), we have that \Omega \vdash e_i' : \text{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 = \emptyset.
Lemma 6 (Type unification). Given \( \vec{b}_2; \Omega \vdash t_1 \leq t_2 : \sigma \), such that the following conditions hold:

A1 \( \mathsf{FV}(t_1) \setminus \mathsf{dom}(\Omega, \Gamma) = \emptyset \)
A2 \( \mathsf{FV}(t_2) \setminus \mathsf{dom}(\Omega, \Gamma) \subseteq \{x \mid x \in \mathsf{FV}(t_1) \} \)
A3 \( \{x \mid x \in \mathsf{FV}(t_1) \} \cap \mathsf{dom}(\Omega, \Gamma) = \emptyset \)

Then, the following are true:

i. \( \mathsf{dom}(\sigma) \subseteq \{x \mid x \in \mathsf{FV}(t_1) \} \)
ii. \( \mathsf{FV}(t_2) \setminus \mathsf{dom}(\Gamma) = \mathsf{dom}(\sigma) \)
iii. \( \mathsf{range}(\sigma) = \{y_1, \ldots, y_n\} \land \forall \vec{b}_2; \Omega \vdash \mathsf{lab} \)
iv. \( \mathsf{FV}(t_1) = \emptyset \Rightarrow \sigma(t_2) = t_1 \) (subject to \( \alpha \)-renaming of bound variables)

Proof. Case (Case U-ID): By (A1) \( \mathsf{FV}(t) \setminus \mathsf{dom}(\Gamma) = \emptyset = \mathsf{dom}(\sigma) \). 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 \( \mathsf{FV}(t_1) \setminus \mathsf{dom}(\Omega, \Gamma) \) is the same as \( \mathsf{FV}(\vec{b}_2; \Omega \vdash \mathsf{lab}) \). By the induction hypothesis, we have that \( t_1 = (\sigma, \beta \rightarrow \alpha) t_2 \). Thus, by \( \alpha \)-renaming of the bound variable in \( \forall \vec{b}_2; \Omega \vdash \mathsf{lab} \), 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) \( \mathsf{FV}(\vec{b}_2; x : t_1 \rightarrow t_2) \setminus \mathsf{dom}(\Omega, \Gamma) = \emptyset \) and, by definition, \( \mathsf{FV}(t_1) \setminus \mathsf{dom}(\Omega, \Gamma) \subseteq \mathsf{FV}(\vec{b}_2; x : t_1 \rightarrow t_2) \); thus \( \mathsf{FV}(t_1) \setminus \mathsf{dom}(\Omega, \Gamma) \setminus \mathsf{dom}(\vec{b}_2) = \emptyset \). (A2) is established using a similar argument for \( \sigma \) and noting that \( \vec{b}_2 \) are all fresh (by \( \alpha \)-conversion, if necessary). (A3) also follows from the freshness of \( \vec{b}_2 \).

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 2 to conclude that \( \mathsf{dom}(\sigma_1) \subseteq \mathsf{dom}(\sigma_2) \); thus \( \mathsf{FV}(\sigma_1 \mathsf{dom}(\Omega, \Gamma) \subseteq \mathsf{FV}(\sigma_2 \mathsf{dom}(\Omega, \Gamma)) \). (A3) follows from freshness as previously.

To establish proposition (i), we simply use \( \mathsf{dom}(\sigma_1, \sigma_2) = \mathsf{dom}(\sigma_1) \cup \mathsf{dom}(\sigma_2) \) and from the induction hypothesis we have \( \forall \vec{b}_2; \Omega \vdash \mathsf{lab} \). Similarly, (iii) follows.

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

To establish proposition (ii), our goal is

\[
\begin{align*}
\mathsf{dom}(\sigma_1, \sigma_2) &= \mathsf{FV}(\sigma_1) \setminus \mathsf{dom}(\Omega, \Gamma) \\
&= \mathsf{FV}(t_1) \setminus \mathsf{dom}(\Omega, \Gamma) \\
&= \mathsf{FV}(t_2) \setminus \mathsf{dom}(\Omega, \Gamma)
\end{align*}
\]

To show this, note that \( \mathsf{dom}(\sigma_0) = \{y, \vec{b}_2\} \) and \( \mathsf{range}(\sigma_0) = x, \vec{b}_2 \). By the induction hypothesis on the second premise, we can conclude that

\[
\begin{align*}
\mathsf{dom}(\sigma_1) &= \mathsf{FV}(\sigma_1) \setminus \mathsf{dom}(\Omega, \Gamma) \\
&= \mathsf{FV}(t_1) \setminus \mathsf{dom}(\Omega, \Gamma) \\
&= \mathsf{FV}(t_2) \setminus \mathsf{dom}(\Omega, \Gamma)
\end{align*}
\]

since \( y \) does not appear in \( t_1' \). By proposition (iii) of Lemma 5 \( \mathsf{FV}(\mathsf{range}(\sigma_1)) \subseteq \mathsf{dom}(\Gamma) \).

So, applying the induction hypothesis to the third premise, we have

\[
\begin{align*}
\mathsf{dom}(\sigma_2) &= \mathsf{FV}(\sigma_2) \setminus \mathsf{dom}(\Omega, \Gamma) \\
&= \mathsf{FV}(t_2) \setminus \mathsf{dom}(\Omega, \Gamma) \\
&= \mathsf{FV}(t_2) \setminus \mathsf{dom}(\Omega, \Gamma)
\end{align*}
\]

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

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

i. \( \vec{b} = \vec{x}, \vec{y} = \mathsf{FV}(t_2) \setminus \mathsf{dom}(\Omega, \Gamma) \)
ii. \( \Omega \vdash t_1 \vdash \kappa; \Omega[\vec{b}] = t_2 \vdash \kappa \)
iii. \( \vec{b}; \Omega \vdash t_1 \leq t_2 : \sigma \)
iv. \( \sigma' \) such that \( \mathsf{dom}(\sigma') \cap \{x \mid x \in \mathsf{FV}(\vec{b}) \} = \emptyset \)

Then

\[ \vec{b}; \sigma; \Omega \vdash t_1 \leq t_2 : \sigma' \]

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

Case (U-TID): Trivial.

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

Case (U-LAB): Straightforward 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 \( \vec{b}_2; \sigma; \Omega \vdash t_1 \leq t_2 : \sigma' \). For the second premise, we must show that

\[ \vec{b}_2; \sigma; \Omega \vdash e_1 \leq (\sigma' \star \sigma) e_2 \]

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

\[ \vec{b}_2; \sigma; \Omega \vdash e_1 \leq (\sigma' \star \sigma) e_2 \]

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

\[ (\sigma' \star \sigma) e_2 \]

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

Case (U-FUN): To use the induction hypothesis on the first premise, we must show that \( \sigma \sigma_1 t_1 = \sigma' \sigma_0 t_1 \). However, we are always free to \( \alpha \)-rename \( \vec{b} \) and \( \vec{b}_2 \) to ensure that \( \mathsf{dom}(\sigma_0) \cap \mathsf{dom}(\sigma_1) \subset \mathsf{dom}(\Gamma) \).
dom(σ′) = ∅; 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 dom(σ1) ∩ dom(σ′) = ∅ 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 $b_1 \equiv \cdot$.

CASE (U-VR1, U-VR2): By the first premises of each rule and assumption (iv) we have that $x \not\in$ dom(σ+). Thus, the conclusion of each of these rules produces $x \mapsto \sigma^*v$ which is precisely the definition of $\sigma^*x \equiv (x \mapsto v)$.

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 $\forall i, dom(\sigma_i) \cap dom(\sigma^+) = \emptyset$.

CASE (U-AS): A straightforward application of the induction hypothesis since, by definition, $e_1 \leq e_1' \in \sigma^+ \sigma_1 \leq \sigma^* e_1' \in (\sigma^+ \Omega) \cdot A$.

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

i. $\Omega \cdot R = \Omega \cdot A = \emptyset$
ii. $\Omega'' = \Omega[\cdot:\kappa]$
iii. $\Omega''[\Omega'] \vdash e_1 \vdash e_2 : t$
iv. $\Omega \vdash t' : \kappa$
v. $\sigma = kw' \cdot \mapsto t'$

Then, $\Omega''[\Omega'] \vdash \sigma e : \sigma t$

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 $\Omega$ and $\Omega[e_1 \leq e_2]$, if both of the following are true

i. $b_2 : \Omega[e_1 \leq e_2] \vdash t_1 \vdash t_2 : \sigma$
ii. $b_2 : \Omega \vdash e_1 \leq e_2 : \sigma'$ with $b_2 \sigma$ disjoint from $b_2'$

Then, $b_2 : \Omega \vdash t_1 \vdash t_2 : \sigma''$ 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 $b_2 : \Omega[e_1 \leq e_2] \vdash e_1 \leq e_2' : \sigma_0$ with $b_2 \sigma$ disjoint from $b_2'$. However, by construction, from assumption (ii) we have $\sigma(e_2) = e_1'$, and by Lemma 5, dom(σ') ∩ dom(σ) = ∅. Thus, it is straightforward to establish $b_2 : \Omega[t] \vdash e \leq e_2' \in \sigma' \circ \sigma_0$. Finally, since, from assumption (ii), $b_2'$ is disjoint from $b_2$ we have dom(σ') disjoint from dom(σ)0 and from dom(σ). Proceed by induction on each (U-τ) and use disjointness 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 $\Omega = \Gamma; \pi; \cdot; \cdot$ such that

i. $\Omega[\psi] = \emptyset \vdash e : t$ and $\Omega[\psi] = \emptyset \vdash e' : t$
ii. $\sigma$, a substitution of free variables in $e$ such that $\sigma(\Omega[\psi] \| \psi = \emptyset \| \pi \vdash e \vdash \sigma(t)$
iii. $\text{range}(\sigma) \subseteq 2^\kappa$, all values, where $\forall x \in \text{dom}(\sigma) \cdot \Omega[\psi] x : t \Rightarrow \Omega[\psi] x : t$
iv. $(\pi \vdash e \vdash e')$

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

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

CASE (E-POL): $x \not\in$ dom(σ). Thus, evaluation can proceed via (E-POL).

CASE (E-TAP): Trivial.

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

CASE (E-FIX): $f \not\in$ dom(σ); trivial.

CASE (E-APP): $\sigma(v^0_2) = \sigma(v^1_2) = \sigma(v^0_1). \sigma(v^0_2)$. If range(σ) is all values, $\sigma(v^1_2) = v^1_2$ (σ(v1) is always a value) and reduction of $\sigma(v^0_1) = \sigma(v^1_2) = \sigma(v^1_1). \sigma(v^0_2)$ can proceed by E-APP. So, in one step of reduction we get $\sigma(\sigma_0, v \mapsto \sigma(v^1_2))$, which is our objective.

CASE (E-MATCH): We can assume dom(σ) ∩ $b_i = \emptyset$. First, we must show that if

$FV(v); FV(p_i) : \| v \leq p_i : \sigma_i$

then

$FV(\sigma v); FV(p_i) : \| \sigma v \leq p_i : \sigma_i$

This is easily shown by contradiction. If $\sigma$ does permit $v$ to be unified with $p_i$, then $\sigma$ 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, $\sigma$ cannot be a unifier.

Next, we consider

$FV(v); FV(p_j) : \| v \leq p_j : \sigma_j$

We have dom(σ) ∩ $b_j = \emptyset$; and, by assumption, dom(σ) ∩ dom(σ) = ∅. Thus, following Lemma 7, it is easy to show that $FV(\sigma v); FV(p_j) : \| \sigma v \leq p_j : \sigma_j \circ \sigma_j$. The result of the reduction is therefore $(\sigma \circ \sigma_j)(\sigma e_j)$, 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. $\sigma$ a substitution of label variables in $t$, such that $\sigma(\Omega) \vdash \sigma(t) : \kappa$ and range($\sigma$) all values.

iii. $\pi \vdash t \mapsto^* t'$

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

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

Lemma 12 (Substitution). Given well-formed $\Omega$, $\Omega'$ both well-formed, and $\Omega''$ such that $\Omega''[\Omega''']$ is well-formed. If all of the following conditions are true:

i. $\Omega.R = \Omega.A = \emptyset$

ii. $\Omega' = \Omega'\{\bar{b} / t\} | R_e | (\forall \bar{x} \cdot \text{dom}(\Omega,G))$

a. $\bar{b} = \Gamma, \text{m}$

b. $\bar{m} = \text{FV}(t) \setminus \text{dom}(\Omega,G)$

c. $R_e = \{x : t_1\} \text{ if } \Omega'\{\bar{b} / t\} \vdash t : R; R_e = \emptyset$ otherwise.

iii. $\Omega''[\Omega'''] \vdash e : t$

iv. $\Omega''[\gamma : \emptyset] \vdash e' : t_1'$ where $\exists v.e' = v$

v. $\bar{b}; \Omega''[\gamma : \emptyset] \vdash t_1 \leq t_1$ \text{ and } $\sigma$

vi. $\sigma' = \sigma, x \mapsto e'$

Then,

$\Omega[\sigma' \Omega'''] \vdash \sigma'(e) : \sigma'(t_2)$


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

Case (T-VAR): Here we have two sub-cases, depending on whether or not $x \in \text{dom}(\sigma')$.

Sub-case (a): Assumption (iii) is of the form $\Omega'[\bar{b} / t_1] | R_e | \Omega''[\Omega''''] \vdash \gamma : y ; x \notin \{x, \bar{b}\}$, and thus, $\sigma'[\gamma] = y$.

We have two further sub-cases:

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

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

Sub-case (b): Assumption (iii) is of the form $\Omega'[\bar{b} / t_1] | R_e | \Omega''[\Omega''''] \vdash x : t_1$, but, $\sigma'(x) = e'$ and, so, from assumption (iv), $\Omega'[\sigma'(x)] \vdash t_1$ 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 $\Omega'[\sigma'(x)]$. Finally, we must show that $\Omega'[\sigma' \Omega'''][\gamma] \vdash \sigma'(t_1)$; however, from the premise of (T-VAR), we have that $\Omega.R = \Omega''[\emptyset] = \emptyset$. Thus, we establish the conclusion since weakening of $\Omega'.G$ is permissible.

Case (T-RVAR): Again we have two sub-cases, depending on whether or not $x \in \text{dom}(\sigma')$.

Sub-case (a) Assumption (iii) is of the form $\Omega'[\bar{b} / t_1] | R_e | \Omega''[\Omega''''] \vdash y : t_2$, where $y \notin \{x, \bar{b}\}$ and $R_e = \emptyset$. Again, we proceed by cases on whether the $y : t$ assumption is present in $\Omega$ or in $\Omega''$, in a manner identical to (T-VAR), sub-case (a).

Sub-case (b): Assumption (iii) is of the form $\Omega'[\bar{b} / t_1] | R_e | \Omega''[\Omega''''] \vdash t_1 | R_i = x : t_1 | \Omega''[\Omega''''] \vdash x : t_1$. 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 $\Omega''[\emptyset]R_x = \emptyset$ since $R_x \neq \emptyset$ and (T-RVAR) requires only a single relevant assumption in the context. Thus, using weakening of $\Omega''[\emptyset]G$ 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-UNIT): Trivial.

Case (T-L1): We use the induction hypothesis on each of the n-premises, obtaining $\Omega[\sigma' \Omega'''][\gamma] \vdash \sigma'(t_i)$ for the $i$th premise. For the conclusion, we note that $\Omega' \vdash C(\bar{c}) = C'(\bar{c})$ and obtain $\Omega[\sigma' \Omega'''][\gamma] \vdash \sigma'(\bar{c}) = \sigma'(\bar{c}) \vdash C(\bar{c}) = C'(\bar{c})$, the desired result by noting that all of the $R_i$’s are present in $\Omega''[\emptyset]G$ and can therefore be partitioned according to the needs to the induction hypothesis. Similarly, for $\vdash C(\bar{c})$.

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


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 $f \notin \text{dom}(\sigma')$. Thus, $\sigma'(\text{fix} f.v) = \text{fix} f.\sigma(v)$. Now, we can use the induction hypothesis on the second premise to establish the conclusion.

Case (T-TAB): Since $\alpha \notin \text{dom}(\sigma')$ we can use the induction hypothesis on the second premise. However, we must be careful in the conclusion with $R$. If $R \neq \emptyset$ in assumption (iii) and $\Omega''[\emptyset]R = \emptyset$, then, where initially we have $\Omega'[\emptyset]R_e = e : t_1$, in the conclusion we have $\Omega[\sigma' \Omega'''][\gamma] \vdash e : \sigma'(t_2)$. To restore the type to $\vdash \sigma'(t_2)$, we can use (T-SUBR), if necessary.

Case (T-ABS): Our goal is to show, via (T-ABS), $\Omega[\sigma' \Omega'''][\gamma] \vdash \sigma(\bar{b})\lambda y : \sigma'(s_1) \vdash \alpha : \sigma(t)$, since $\text{dom}(\sigma')$ cannot mention $\bar{b}, y$.

From assumption (iii), we have $\Omega'[\emptyset]R_e = e : s_1$, with the premise, $\Omega'[\emptyset]R_e = e : s_2$. However, this context is of the form of $\Omega''[\emptyset]G$, and is well-formed by Lemma 4. Thus, the induction hypothesis is applicable and we obtain

$$\Omega[\sigma' \Omega'''][\gamma] \vdash \sigma'(s_1) \vdash \sigma'(\bar{b})\lambda y : \sigma'(s_1) \vdash \alpha : \sigma'(t)$$
Thus, to reach our goal, we use this last judgment in the premise of (T-ABS). Finally, if $R_e = \Omega', R = \emptyset$ and $R_e \neq \emptyset$, 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 $\Omega[s'\Omega', t'] : \kappa$. 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 $R_1 \sqcup R_2 = [R, \Omega \sqcup \Omega', R]$. We can use the induction hypothesis on the first premise to establish

$$\Omega[s'\Omega'][R = R_1 \setminus R_2] \vdash s'_1 : s_1 t_1 \vdash \sigma' t_2$$

Similarly, for the second premise we obtain

$$\Omega[s'\Omega'][R = R_2 \setminus R_1] \vdash s'_2 : \sigma_1' t_1$$

Now, by, Lemma 7 we obtain that $\bar{b} : \Omega[s'\Omega', t] : \vdash s'_1 t_1 \leq \sigma' t_1 : \sigma' * \sigma$. 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) $R_1 \sqcup R_2 = R, \Omega \sqcup \Omega', R$. The first five premises are trivial to re-establish since $\text{dom}(\sigma)$ doesn’t include any of $\bar{b}$. For the sixth premise, we use the induction hypothesis and restore $\sigma' e : \text{lab}$. Similarly, for the ninth premise, we have $\sigma' e t : \sigma' t : \text{i.e. uniform type for each case of the pattern. The conclusion is straightforward.}$

**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 to the induction hypothesis to the first premise, we obtain $\Omega[\sigma'\Omega'][R = \emptyset] \vdash \sigma' e : \sigma t$. Now, given $\pi \vdash t \rightarrow^* t'$, we must show that $\pi \vdash \sigma' t \rightarrow^* \sigma' t'$. This is exactly the statement of Lemma 11.

**Lemma 13** (Substitution for kinding judgment). Given well-formed $\Omega, \Omega'$ both well-formed, and $\Omega''$ such that $\Omega''[\Omega']$ is well-formed. If all of the following conditions are true:

i. $\Omega, R = \emptyset$, $\Lambda \subseteq \emptyset$

ii. $\Omega' = \emptyset$, $\bar{b}$, where $\bar{b} = \bar{f}, \bar{m}$

iii. $\Omega'' = \emptyset$; $\bar{t} : \kappa$

iv. $\Omega''[\Omega'] \vdash t : \kappa$

v. $\bar{b} : \Omega \vdash t_1 \leq t_1 : \sigma$

Then, $\Omega[\sigma'\Omega', t'] : \kappa$

**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 $\Omega \equiv \Gamma; \vdash t : \kappa$ and memory $\tilde{M}$ such that $\Omega \vdash \tilde{M}$ and $\pi, e$ such that (A1) $\Omega \vdash \pi, e : t$ then (A2) $\pi \vdash (M, e) \rightarrow (M', e')$ implies $\Omega \vdash \tilde{M}'$ 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 $M = \tilde{M}'$) where the obligation of $\Omega \vdash \tilde{M}'$ is satisfied by assumption.

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

**CASE (E-TAP):** By assumption, the second premise of (T-PROG) concludes with an application of (T-TAP), $\Omega'' \vdash v [t_1] : (\alpha \rightarrow t_2)$. There are two cases (A2) is a well-typed, we must show that $\Omega \vdash \pi, \alpha : \tilde{M}$. However, to reach our goal, we use this last judgment in the premise of (T-TAB). In order to re-establish that the conclusion of (A2) is valid, we redefine $\Omega''[\Omega']$ with $\Omega''[\tilde{M}]$ and $\Omega''[\tilde{M}]$ 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 $\text{dom}(\sigma) = \{f \} \subseteq \text{dom}(\Omega, \Gamma)$ and $\text{range}(\sigma') \subseteq 2^\Gamma$ satisfying the assumptions of the lemma.

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

**Sub-case (a):** (A1.2) is an application of (T-PFX). Then (A4) $\Omega'' \vdash (t') u : t'$. 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 $\Omega' \vdash u : t'(\vec{e})$ and $t' = t'(\vec{e})[\vec{e}]$. So, to re-establish the induction hypothesis, we use (T-PFX) with $\Omega' \vdash u : t'(\vec{e})[\vec{e}]$ in the premise via (ESC).

**Sub-case (a.ii):** (A4) is (T-SFX). So, we have $\Omega' \vdash u : t''$ and $t' = t''[\vec{e}] = t'(\vec{e})$. 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) $\Omega' \vdash v' : t'$ with (T-APP), (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) $\Omega' \vdash v_1 \cdot b. x : t_1 \Rightarrow t_2$ as the second premise, and (A1.2.3) $\Omega' \vdash t': t''$ as the third premise. Now, as with (T-TAP), we must proceed on cases where either $\exists y v_1 = u$ or $\exists v_1 = \emptyset\cdot u$. Given that $\Omega', R = \emptyset$ the distinction is not significant, in the latter case, $\Omega' \vdash \emptyset. \cdot v_1 : t$ can be replaced with $\Omega' \vdash v_1 : t$. Thus, we can assume that $u = \emptyset \cdot b. x : t_1, e$, 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), $\Omega' \vdash [\vec{e}] = \cdot : e \Rightarrow t_2$, and from the fourth premise of (A1.2) we have, $\vec{h}. \Omega' \vdash t'; \vec{t}_1 : \sigma$, and, from clause (iii) of Lemma 6, we have that range$\sigma$ is limited to values, and dom$\sigma = \vec{h}$, where dom$\sigma \cap \bigcup V(\Omega, \Gamma) = \emptyset$. Note that $\vec{R}' \subseteq \{ t_1 \}$.

Our goal, then, is to show that $\Omega' \vdash (\vec{e}, x \mapsto v_2) e_j : (\sigma, x \mapsto v_2) t_2$. Using (A1.2.2.1), (A1.2.3), we can apply the substitution 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), $\Omega' \vdash \mathbf{v} \mathbf{w} \ldots : t$. Since, $\Omega = M$ and dom$\Omega = \emptyset$, we can conclude that $FV(\mathbf{v}) = \emptyset$.

The third premise of (A2) (E-MATCH) gives us (A2.1) $\vec{h}. \Omega' \vdash v \leq p_j \vdash \vec{t}_1 : \sigma_j$ and from the premises of (A1.2) $\Omega' \vdash \mathbf{v} \vdash \vec{p}_j : \mathbf{t}$; lab. In order to establish the applicability of Lemma 5, we show that its condition (A1) is applicable since $FV(\mathbf{v}) = \emptyset$; condition (A2) and (A1.2) from the fifth premise of (T-MATCH). From clause (iv) of Lemma 5, $FV(p_j) = \vec{h} = \emptyset$ since, $\Omega' = \emptyset$.

Furthermore, since by assumption $\mathbf{v}$ has type $\mathbf{p}$, lab, by the last premise of (A1.2) (T-MATCH) we have (A1.2.1) $\Omega'[\vec{b}] [v \leq p_j] \vdash e_j : t$. First, to establish $\Omega'[\vec{b}] [v \leq p_j] \vdash e_j : t$, we rely on Lemma 9 in conjunction with (A1.2.1) and (A2.1). The assumption $v \leq p_j$ is used only in the second-to-last premise of (T-APP) in the form of (A1.2.1.1,$x \vec{h}. \Omega'[\vec{b}][v \leq p_j] \vdash t_1 \leq t_2 : \sigma'$ with $\Gamma(\vec{b})$ undefined. Thus $\vec{b}$ in (A2.1) is disjoint from $\vec{b}$ in (A1.2.1.1.x) and Lemma 9 is applicable. We obtain $\vec{h}. \Omega'[\vec{b}][v \leq p_j] \vdash t_1 \leq t_2 : \sigma_j \circ \sigma'$ from which, we can conclude $\Omega'[\vec{b})] [v \leq p_j] \vdash e_j : \sigma_j(t)$.

However, by noticing that each $\vec{b}$ is distinct, $FV(t) \cap \text{dom}(\sigma_j_1) = \vec{b}$, since otherwise the last premise of (T-MATCH) in (A1.2) requiring each $e_j$ to have the same type $t$, could not be satisfiable. Thus $\sigma_j \circ \sigma' \equiv \sigma'$ and we have $\Omega'[\vec{b})] [v \leq p_j] \vdash e_j : t$.

Finally, by repeated application of the substitution lemma (Lemma 12) we have $\Omega'[\vec{b})] [\sigma_j(e) : \sigma_j(t)]$. 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), $\Omega' \vdash l : t$ with $\Omega' \vdash l : \text{ref}$ in the premise. By well-formedness of $\Omega'$ and $\Omega'$ 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 $\Omega' \vdash v_2 : t$, where dom$(\Omega') = \text{dom}(M) \cup \text{dom}(\pi)$ and dom$(\pi)$ 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 $\pi \vdash t \Rightarrow t' \Rightarrow \mathbf{lab} \Rightarrow \mathbf{e}'$. However, this is immediate from the definition of the type evaluation context, case lab ~ *.

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

**Sub-case (T-L3):** We use the induction hypothesis on the premise. Now, we must establish that $\pi \vdash \mathbf{lab} \Rightarrow \mathbf{e} \Rightarrow \mathbf{e}'. \mathbf{lab}$. But, from the definition of the type evaluation context, case lab ~ *; we have that $\mathbf{e} \Rightarrow \mathbf{e}'$. 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 $\Omega \vdash \mathbf{e} : \mathbf{lab}$. Now, we can use (T-L5) to establish $\Omega \vdash \mathbf{e}' : \mathbf{lab} \Rightarrow \mathbf{e}'$. However, by assumption (A2), and the premise of (A1) that establishes that $\mathbf{e}$ has no side-effects, we have that $\mathbf{e} \Rightarrow \mathbf{e}'$. Thus, using the (T-L5) with (TECTXE2) we re-establish $\Omega \vdash \mathbf{e} : \mathbf{lab} \Rightarrow \mathbf{e}'$ as desired.

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

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

But, by the premises of (T-L1) and also in (T-L5), we have that $e_j$ is effect-free. Thus, if $\pi \vdash (M, e_j) \Rightarrow (M, e_j')$ then $\pi \vdash e_j \Rightarrow e_j'$; thus via (T-CTXE) for the label context, and using (TECTXE2) we have that $\pi \vdash t \Rightarrow t$ and we can re-establish $\Omega \vdash e : \mathbf{lab} C(\vec{v}, e_j, \vec{e})$, using (T-L1) or (T-L5) as the premise of (T-EVT).

**Sub-case (E = \vec{C}(\vec{v}, e_j, \vec{e})):** Identical.
letpol \(\lambda x.\text{lab}.\lambda y.\text{lab}.\text{match } x, y\) with
\(x, x 
\rightarrow x\)
LOW, \(x 
\rightarrow y\)
MED, \(\text{LOW} \rightarrow x\)
MED, \(\text{LOW} \rightarrow y\)

letpol low \(\lambda x.\lambda x.\alpha.\text{lab}.\alpha\{\text{LOW}\}\) \(\rightarrow x\)

letpol join \(\lambda x.\phi l m.\alpha x\{\{\alpha\}\} m.\alpha l m\) \(\langle\alpha l m\rangle \rightarrow x\)

letpol sub \(\lambda x.\phi l m.\alpha x\{\{\alpha\}\} m.\alpha l m\) \(\langle\alpha l m\rangle \rightarrow x\)

letpol def \(\lambda x.\phi l m.\alpha x\{\{\alpha\}\} m.\alpha l m\) \(\langle\alpha l m\rangle \rightarrow x\)

letpol app \(\lambda x.\phi l m.\alpha x\{\{\alpha\}\} m.\alpha l m\) \(\langle\alpha l m\rangle \rightarrow x\)

letpol app \(\lambda x.\phi l m.\alpha x\{\{\alpha\}\} m.\alpha l m\) \(\langle\alpha l m\rangle \rightarrow x\)

letpol app2 \(\lambda x.\phi l m.\alpha x\{\{\alpha\}\} m.\alpha l m\) \(\langle\alpha l m\rangle \rightarrow x\)

let pol app \(\lambda x.\phi l m.\alpha x\{\{\alpha\}\} m.\alpha l m\) \(\langle\alpha l m\rangle \rightarrow x\)


\[\textbf{Figure 14.} \text{ Enforcing a static information-flow policy.}\]

**Sub-case \((E = e_1 e_2):\)** The top-level typing judgment (A1) is (T-APP). Let the second premise of this judgment be \(\Omega \vdash e_1 : t_1\). To re-establish the hypothesis, we will apply (T-APP) again. By the induction hypothesis we have that \(\Omega \vdash e_1 : t_1\). By assumption, since \(e_2\) is unchanged, the remaining premises are unchanged.

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

**Sub-case \((E = e (\tau)):\)** 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 \((E = \text{match } e \text{ with } \ldots):\)** The top-level typing judgment is (T-MATCH). We use the induction hypothesis on the first premise noting that \(t = t\) since the type \(\text{lab}\) is irreducible.

**Sub-case \((E = 1 \cdot 1\):** 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 analyzers.

Figure B gives the semantics of \(\text{FABLECORE}^2\), an extension of \(\text{FABLECORE}\) in the spirit of Core-ML\(^2\) [21], Potter and Simonet’s technique for representing multiple program execution within the syntax of a single program.

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

**Lemma 15** (Progress for \(\text{FABLECORE}^2\)). Given well-formed \(\Omega = \vdash e : \pi\) the policy of Figure 4, and a \(\text{FABLECORE}^2\) program \(e\) such that \(\Omega \vdash e : t\) and all relabeling operations in \(e\) are indexed by one of the indices that appear in text of \(\pi\). Then, \(\Omega \vdash e \rightarrow e'\), or \(e\) is a value.

**Proof.** By induction on the structure of \(\Omega \vdash e : t\), 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 \(\{e\} \vdash e'\) or \(\{e\} \vdash e\).

**CASE (T-PFX):** If \(e \neq v_1 \parallel v_2\) then we follow Theorem 3, appealing to evaluation context \(\{e\} \vdash e'\) or \(\{e\} \vdash e\), if necessary. Otherwise, if \(e = v_1 \parallel v_2\), then if \(\tau : U\), by definition \(\langle\tau\rangle e\) 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 \(\langle\tau\rangle v_1 \parallel v_2\), since, by (T-BRACKET) \(\Omega \vdash v_1 \parallel v_2 : \tau\), and from the first premise of (T-APP) we must have \(e_1\) of unlabeled type, and so \(\tau\) 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 unlabeled operations in the program. In the first case, we have an evaluation context

\[E \cdot \langle\tau\rangle^5 (v_f \parallel v'_f)\]

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

\[E \cdot (\langle\tau\rangle^5 (v_f \parallel v'_f)\]

Similarly, for

\[E \cdot (\langle\tau\rangle^8 (v_f \parallel v'_f)\]

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

Finally, we observe that it is impossible for \(e_1 = \langle\tau\rangle^6 v_1 \parallel v_2\). From the source program, we note that such a term is introduced by \(e' = e'_1 e'_2\) where \(e'_2 = \langle\tau\rangle^6 v_1 \parallel v_2\). 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 \(e'_2 = v_4 = \langle\tau\rangle^6 v_1 \parallel v_2\), then, in one step of reduction, using the definition of \(v_4\), the \(\langle\tau\rangle^6 v_1 \parallel v_2\) term is immediately destructed. Similarly for (E-BAPP2). Thus, it is impossible for \(\langle\tau\rangle^6 v \parallel v'\) to appear in the left-side of an application.

By the above argument, it is also straightforward to establish that if \(e_1 = \langle\tau\rangle^6 v \parallel v'\), then, although (E-BAPP1) 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 \(\langle\tau\rangle v_1 \parallel v_2\). Note that the bracketed term must be prefixed by an unlabeled 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 unlabeled operators in the program are labeled 5, 6, 8; but, as our examination
in (T-APPL) has shown, all these unlabeled bracketed terms are destructed immediately. Thus, the case of \( \nu = \langle t \rangle \{ v_1 \parallel v_2 \} \) is ruled out and we can rely on Theorem 3.

**Lemma 16** (Subject reduction for FABLECORE\(^2\)). *Given well-formed \( \Omega \vdash \cdot : \cdot \), and \( \pi \) the policy of Figure 14, and a FABLECORE\(^2\) program \( e \) such that all relabeling operators in \( e \) are annotated with an index from \( \pi \), 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 the additional rules in Figure B, relying on Theorem 14 for the other cases.

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

**CASE (APPL):** 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-APPL) 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 (APPL1):** It is straightforward to show that \( \langle t \rangle v' \) and \( \langle t \rangle v' \) have the same type as \( \langle t \rangle \{ v_1 \parallel v_2 \} \), using the third and fourth premises of (T-APPL). Similarly for \( v_1 \) and \( \{ v_1 \} \), and finally that in \( \pi \vdash \cdot \rightsquigarrow \{ e_1 \parallel e_2 \} \) that \( e_i \) 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, \( \langle t \rangle v' \equiv \langle t \rangle \{ v_1 \parallel v_2 \} \). Thus, by (T-APPL), it must be the case that \( \pi \equiv \{ v_1 \parallel v_2 \} \). 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-APPL) to satisfy its first two premises.

**CASE (APPL2):** Similar to the previous case to show that each \( e_i \) is well-typed. Now, must still provide the evidence for the first two premises of (T-APPL) in order to type \( \{ e_1 \parallel e_2 \} \). However, by inspection of the policy function app, in which \( \langle v \rangle \) appears, we can conclude that \( t_i \equiv \pi \) and thus has no label. However, as previously \( \langle t \rangle = \langle t \rangle \{ \text{lab} \} \) where \( t_i \{ m \} \) is the type of \{ v_1 \parallel v_2 \}. From, (T-APPL) we can conclude that \( t_i \{ m \} \) is guarded at level HIGH and thus \( t \) is also guarded at HIGH, which is of the type of \( e_1 \) and \( e_2 \). This is sufficient for (T-APPL).

**Theorem 17** (Noninterference). *Given \( \pi \), the policy of Figure 4, and an expression \( e \) that does not contain any relabeling operations, and an empty context \( \Omega \{ \pi : \pi \} \) such that \( \Omega \{ \pi : \pi \} \) is \( \pi \vdash : t \{ \text{HIGH} \} \), \( \pi \equiv , \pi : e : t \{ \text{LOW} \} \), \( \pi \equiv , \pi : e : t \{ \text{LOW} \} \) and \( \pi \equiv , \pi : e : t \{ \text{LOW} \} \).

**Proof.** Straightforward from the substitution lemma, Lemma 12; Lemma 16 and from construction, \( \Omega \vdash e : t \) where \( t \) not guarded at HIGH, implies \( \{ v_1 \} \equiv \{ v_2 \} \).

**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 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 \( \{ t_1 \rightarrow t_2 \} \) is non-degenerate if, and only if, \( t_1 \) and \( t_2 \) are non-degenerate; unit is non-degenerate. A typing derivation \( \Gamma \vdash e : t \) is non-degenerate if, and only if, for every sub-derivation \( \Gamma' \) with conclusion \( t' \), \( t' \) is non-degenerate.

The non-degeneracy condition above assures that all function-typed expressions \( e \) are given types that permit the application of \( e \) and the third premise of (ML-APPL) that requires the type of the function to be non-degenerate. So, while in programs such
as \((λx.0)e_1, e_2\) 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 \(\mathcal{D}\) to FABLECORE programs \(e\).

**Theorem 19** (Completeness of static information flow). Given \(e\) such that, \(\mathcal{D} = Γ \vDash_{\text{ML}} e : x : t\) is non-degenerate; then \([Γ] : \pi; [\mathcal{D}] : [t]\), where \(\pi\) is the policy of Figure 14.

**Proof.** By induction on the structure of the translation \([\mathcal{D}]\).

**Case (X-U):** Trivial.

**Case (X-V):** By the induction hypothesis we have
\[
[Γ, f : (t_1 \to t_2)^′, x : t_1] ≺ [t_1, [t_2]]
\]
To establish the conclusion we use, (T-FIX) with (T-SFX) followed by (T-ABS), with the induction hypothesis applicable in the third and fourth premises of (T-ABS), and \(x = 0\).

**Case (X-APP1, X-APP2):** By the induction hypothesis we have both
\[
i.[Γ] : \pi; [\mathcal{D}_1] : [(t_1 \to t_2)^′], \text{ and,}
\]
\[
i.[Γ] : \pi; [\mathcal{D}_2] : [t_1]
\]
The type of \(\text{app}2\) is \(∀α : \text{M}, [Γ, f : \text{M} ; \lambda(α → β)(1) → x : α → β]l\). Thus, we apply the type \(\text{app}2\ldots [\mathcal{D}_2]\) to be \([t_1']\). In case (X-APP2) this is specifically \(([t_1']\ldots [t_2])\) which is an acceptable translation of the Core-ML type unit. In case (X-APP1), this type is specifically \([l]\ldots [l]\) which is not yet an acceptable translation of \(t_2\). Thus, we apply \(\text{join}\ldots\) which has type \(∀α, φ, m : \text{M} ; α(1) [m] → α \{\text{lab} [l] m\}\) to obtain \([l] \{\text{lub} [l] m\} [t_2] = [\text{lub} [l] m] [t_2]\}. To conclude, we use the final premise of (ML-APP) which asserts that \(l \subset t_2\), which requires \(l \subset l'\). Thus, \(\text{lub} [l] m [t_2] = [t_2]'\).

**Case (Case X-SUB):** We proceed by induction on the structure of the subtypings derivation using the induction hypothesis to establish that \([Γ] : \pi; [\mathcal{D}] : [t]\). That is, we wish to establish that given
\[
[Γ] : \pi; e : [t]\]
then
\[
[Γ] \vdash _{\pi} [t] \vdash [t]'\]
The (SUB-FN) case is trivial. We examine first the type of \(e'\) in (SUB-FN). By assumption we have that the type of \(e\) is \((t_1 \to t_2)(e_1)\). We have that \(x : t_1\) by ascription in the lambda binding. Thus, by the inductive hypothesis we have that \([Γ, x : t_1] : [\mathcal{D}_1] : t_1\). Now, using the type for \(\text{app}\) given in (Case X-APP1), we conclude that \(\text{app}\ldots [\mathcal{D}_1]\) has type \(t_2(e_2)\). After the application of \(\text{join}\) we conclude that \(e'\) has type \(\text{lub} e_1 e_2\). However, from the non-degeneracy assumption, we have \(l \subset t_2\) or \(l \subset l'\); thus, the type of \(e'\) is \((t_1 \to t_2)\). To type \(λx : \text{M} : \text{lub} \to α \to α\{l\}\) we use the induction hypothesis to establish that \([\mathcal{D}_2]\) has type \(t_2\) to arrive at the type \(t_1' \to t_2'\) using (T-ABS) with \(x = 0\). Finally, the type of \(\text{def}\) is \(∀α : \text{M} : \text{lub} \to α \to α\{l\}\), which is sufficient to establish the type of \((t_1' \to t_2')\) for the translation, which is our goal. \(\square\)

**References**


Translation of Core-ML types and environment

\[ [\text{unit}] \equiv \text{unit} \]
\[ e_1 \in \{\text{LOW}, \text{MED}, \text{HIGH}\} \]
\[ [e_1] \equiv [\text{unit}]. [e_1] \]
\[ [(t_1 \rightarrow t_2)]' \equiv ([(t_1)] \rightarrow [(t_2)]) \{[\text{unit}]\} \]
\[ [\text{unit}] \equiv \text{unit} \]

Translation of Core-ML labels to FABLECORE terms

\[ [\text{LOW}]' \equiv \text{LOW} \]
\[ [\text{MED}]' \equiv \text{MED} \]
\[ [\text{HIGH}]' \equiv \text{HIGH} \]

Translation of derivations \( \mathcal{D} \) to FABLECORE expressions.

\[ [\Gamma \vdash \text{ML} \cdot \text{unit}] \equiv () \] (X-U)
\[ [\Gamma \vdash \text{ML} \cdot x : \Gamma(x)] \equiv x \] (X-V)
\[ \begin{array}{c}
\frac{\mathcal{D} \vdash f : (([(t_1 \rightarrow t_2)]') \Rightarrow \tau)}{\Gamma \vdash f : \text{ML} \cdot \text{fix}_f \cdot \lambda x. e} \quad (X-ABS) \\
\frac{\mathcal{D}_1 = \Gamma \vdash \text{ML} \cdot e_1 : (t_1 \rightarrow t_2)'}{\mathcal{D}_2 = \Gamma \vdash \text{ML} \cdot e_1 \cdot e_2 : t_2}
\end{array}
\]
\[ \frac{[\mathcal{D}_1 \cdot \mathcal{D}_2 \cdot l \vdash t = t']}{[(t_2 = t')] \equiv \text{join} \cdot [\text{app2} \cdot [(t_1)] \cdot [(t_2)] \cdot [[\mathcal{D}_1]] \cdot [[\mathcal{D}_2]]} \] (X-APP1)
\[ \frac{\mathcal{D}_1 = \Gamma \vdash \text{ML} \cdot e_1 : (t_1 \rightarrow t_2)'}{\mathcal{D}_2 = \Gamma \vdash \text{ML} \cdot e_1 \cdot e_2 : t_2}
\]
\[ \frac{[\mathcal{D}_1 \cdot \mathcal{D}_2 \cdot l \vdash t = t']}{[(t_2 = \text{unit})] \equiv \text{app2} \cdot [(t_1)] \cdot [(t_2)] \cdot [[\mathcal{D}_1]] \cdot [[\mathcal{D}_2]]} \] (X-APP2)

Translation of the subtyping judgment given as a translation of a \([t]\)-typed FABLECORE expression, e.

\[ [t \leq t'] \equiv e' \] (X-SUB)

\[ \frac{\mathcal{D}_1 = \Gamma \vdash \text{ML} \cdot e_1 : (t_1 \rightarrow t_2)'}{\mathcal{D}_2 = \Gamma \vdash \text{ML} \cdot e_2 : t_2}
\]
\[ \frac{[\mathcal{D}_1 \cdot \mathcal{D}_2 \cdot l \vdash t \leq t']}{[(t_1 \leq t_2) \Rightarrow (t_1' \leq t_2')] \equiv \text{def} \cdot [t_1' \rightarrow t_2'] \cdot e' \cdot (\lambda x : t_1', \mathcal{D}_2') \} \] (SUB-FN)

where
\[ t_1 = [(\text{unit})], \quad t_1' = [(\text{unit})]
\[ t_2 = [t_2], \quad e_2 = [t_2]
\[ t_2 = [t_2] = \tau(e_1), \quad t_2' = [t_2']
\[ e_1 = [\mathcal{D}_1], \quad e_2 = [\mathcal{D}_2], \quad \text{and,}
\[ e' = \text{join} \cdot [\text{app2} \cdot [(\text{unit})] \cdot [t_2] \cdot e \cdot [\mathcal{D}_1]]
\]

Figure 17. Translation from a Core-ML derivation \( \mathcal{D} \) to FABLECORE.