A Type System for Expressive Security Policies*

David Walker
Cornell University

Abstract

Certified code is a general mechanism for enforcing security properties. In this paradigm, untrusted agent code carries annotations that allow a host to verify its trustworthiness. Before running the agent, the host checks the annotations and proves that they imply the host's security policy. Despite the flexibility of this scheme, so far, compilers that generate proof-carrying code have focused on simple memory and control-flow safety rather than more general security properties.

Security automata can enforce an expressive collection of security policies including access control policies and resource bounds policies. In this paper, we show how to take specifications in the form of security automata and automatically transform them into signatures for a typed lambda calculus that will enforce the corresponding safety property. Moreover, we describe how to instrument typed source language programs with security checks and typing annotations so that the resulting programs are provably secure and can be mechanically checked. This work provides a foundation for the process of automatically generating secure certified code in a type-theoretic framework.

1 Introduction

Strong type systems such as those supported by Java or ML provide provable guarantees about the runtime behaviour of programs. If we type check programs before executing them, we know they "won't go wrong." Usually, the notion "won't go wrong" implies *memory safety* (programs only access memory that has been allocated for them), *control flow safety* (programs only jump to and execute valid code), and *abstraction preservation* (programs use abstract data types only as their interfaces allow). These properties are essential building blocks for any secure system such as a web browser, extensible operating system, or server that may download, check and execute untrusted programs. However, in order to build such systems, we must restrict program behaviour much more drastically; standard type safety properties are not sufficient to enforce realistic access control policies or to restrict the dissemination of secret information.

Certified code is a general framework for verifying security properties in untrusted code. To use this security architecture, a programmer or compiler must attach a collection of annotations to the code that they produce. These annotations can be proofs, types, or annotations from some other kind of formal system. Regardless, there must be some way of reconstructing a proof that the code obeys a certain security policy, for upon receiving annotated code, an untrusting web browser or operating system will use a mechanical checker to verify that the program is safe before executing it.

In theory, certified code is very general, but in practice, compilers that emit certified code have focused on a relatively limited set of properties. For example, Necula and Lee's proof-carrying code (PCC) implementation [14, 13] uses a first-order logic and they have shown that they can check many interesting properties of hand-coded assembly language programs including access control and resource bound policies [16]. However, the main focus of their proof-generating compiler Touchstone [15] is the generation of efficient code; the security policy they enforce is the standard type and memory safety. Other frameworks for producing certified code including Morrisett *et al.*'s [12, 9, 8] Typed Assembly Language (TAL) and Kozen's efficient code certification (ECC) [5] concentrate exclusively on standard type safety properties.

The main reason that certified code has been used in this restricted fashion is that automated theorem provers are not powerful enough to infer properties of arbitrary programs and, except in rare cases, constructing proofs by hand is prohibitively expensive. Researchers have been able produce proofs of type safety for their code because they place heavy restrictions on the programming languages that they compile and when they cannot prove type safety statically, they insert dynamic checks, completing the

*This material is based on work supported in part by the AFOSR grant F49620-97-1-0013. Any opinions, findings, and conclusions or recommendations expressed in this publication are those of the authors and do not reflect the views of this agency.

proof at run time. For example, using arrays safely requires that they prove each access is in bounds. The Touchstone compiler uses a theorem prover to attempt to complete this proof statically, but when it cannot do so, it inserts a run-time check.

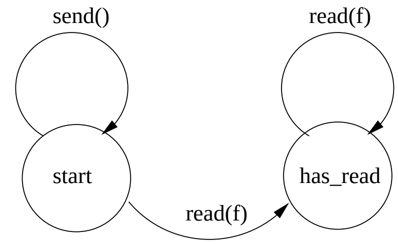
In order to construct a tractable system for secure certified code, we will follow the model developed for standard type safety proofs: Instrument the program with run-time security checks and then eliminate those checks that a theorem prover can verify statically. For an interesting class of security properties, this strategy will guarantee that any program can be automatically rewritten so that it is provably safe; the programmer need not be burdened by extensive proof obligations.

1.1 Security Automata and SASI

Unlike memory safety and control-flow safety, properties that must be enforced across all applications, security policies may vary from one application to the next. Schneider [19] has proposed *security automata* as a flexible mechanism for enforcing a much larger class of security policies than are possible in traditional type systems. Security automata are defined similarly to other automata [4]. In the model we will use in this paper, these machines possess either a finite or a countably infinite set of states, and rules for making transitions from one state to the next when they receive inputs. One of the states in the automaton is designated as the *bad* state. Security automata enforce safety properties by monitoring programs as they execute: Before an untrusted program is allowed to execute a security-sensitive operation, the security automaton checks to see if that operation will cause a transition to the *bad* state. If so, the automaton terminates the program. If not, the program is allowed to execute the operation and the security automaton makes a transition to a new state.

By monitoring programs in this way, security automata are sufficiently powerful that they can restrict access to sensitive files or operations, or bound the use of resources. They can also enforce the safety properties typically implied by type systems such as memory safety and control-flow safety. Security automata can only enforce safety policies. Hence, some interesting security policies including information flow and resource availability cannot be enforced by this mechanism. However, Schneider [19] points out that for some of these applications, we can still use a security automaton: The automaton must simply enforce a stronger property than is required. For example, we can ensure no secret information flows to the outside world by disallowing access to the network. This safety property can be enforced by a security automaton, but it reduces the number of legal programs that can be written.

The following diagram depicts a security automaton that enforces the policy that code must not perform a send on the network after reading a file:



The security automaton actually has three states: a *start* state, a *has_read* state, and the *bad* state, which is not shown in the diagram. For the purpose of this example, we will assume that there are only two security-sensitive or *protected* operations: the *send* operation and the *read* operation. Each of the arcs in the graph is labeled with one of these operations and indicates the state transition that occurs when the operation is invoked. If there is no outgoing arc from a certain state labeled with the appropriate operation, the automaton makes a transition to the *bad* state. For example, there is no *send* arc emanating from the *has_read* state. Consequently, if a program tries to use the network in the *has_read* state, it will be terminated.

We can enforce the policies specified by security automata by instrumenting programs with run-time security checks. For example, consider a program that executes the *send* operation at some point during computation. According to the security policy, we must be in the *start* state in order for the *send* to be safe. A program instrumentation tool could enforce this policy by wrapping the code invoking *send* with security checks:

```
let new_state = check_send(current_state) in
if new_state = bad then
    halt
else
    use()
```

The first statement invokes the security automaton to determine the next state given that a *send* operation is invoked in the current state. The second statement tests the next state to make sure it is not the *bad* one. If it is *bad*, then program execution is terminated.

After performing the initial transformation that sandboxes all protected operations, a program optimizer might attempt to eliminate redundant checks by performing standard program optimizations such as loop-invariant removal and common subexpression elimination. An optimizer might also use its knowledge of the special structure of a security automaton to eliminate more checks than would otherwise be possible.

An implementation developed by Erlingsson and Schneider [21] attests to the fact that security automata can enforce a broad, practical set of security policies. Their tool, SASI, automatically instruments untrusted code with checks dictated by a security automaton specification and optimizes the output code to eliminate checks that can be proven unnecessary. The tool is both flexible and efficient and they have implemented a variety of security policies from the literature. For example, using SASI for the Intel Pentium architecture, they have specified the memory and control-flow safety policy enforced by Software Fault Isolation (SFI) [22]. The SASI-instrumented code is only slightly slower than the code produced by the special-purpose, hand-coded MiSFIT tool for SFI [20]. As another example, using SASI for the Java Virtual Machine, they have been able to reimplement the security manager for Sun's Java 1.1. The SASI-instrumented code is equally as efficient as the Java security manager in some cases and more efficient in others. SASI is more flexible than the Java security model because individual systems or users can customize the set of protected operations rather than having that set dictated by the Java language specification.

1.2 An Overview

This paper presents a framework for automatically compiling programs into certified code that satisfies security automaton specifications. More specifically, we show how to instrument strongly-typed programs that do not necessarily follow the security policy with run-time security checks and typing annotations. The output from the instrumentation algorithm type checks in a new strongly typed language and the soundness of the language's type system ensures that the automaton security policy will never be violated. Furthermore, because the type system is defined independently of the instrumentation algorithm, untrusted parties can write their own instrumentation and optimization transformations. If the output program type checks then the code obeys the security policy. Thus, we also provide a new way to certify that untrusted code obeys expressive security policies.

In order to accomplish these goals, we have embedded predicates into the type system of the target language. Each protected operation is given a type that uses the predicates to specify a precondition restricting the application of the function. The precondition states that the function must not cause the program to enter the *bad* state. Now, each time the function is applied, the type checker must be able to prove that the calling context satisfies the precondition. If it can do so, the program satisfies the security policy.

In general, in order to complete the proof of safety, the program will have to contain run-time security checks. Intuitively, these security checks have types that express post-conditions containing information about the automaton transition function. The post-conditions can be used to help prove the preconditions on the protected operations.

Using these predicates (P), we can check that the pseudo-code from the previous section is safe:

$\texttt{let}\ \mathit{new\_state} = \mathit{check}_{\mathit{send}}(\mathit{current\_state})\ \texttt{in}$

$P_1: \quad \mathit{transition}_{\mathit{send}}(\mathit{current\_state}, \mathit{new\_state})$

$\texttt{if}\ \mathit{new\_state} = \mathit{bad}\ \texttt{then}$

    `halt`

`else`

    P_2: $\mathit{new\_state} \neq \mathit{bad}$

    $\mathit{send}()$

The predicates P_1 and P_2, together with the information that the automaton is actually in the state designated *current_state*, are sufficient to prove that the precondition on the *send* operation has been satisfied.

Our techniques offer a number of advantages over previous security architectures:

1. Given any well-typed source program, instrumentation is automatic and guaranteed to succeed. Programmers are not burdened with the obligation to produce proofs or to write down annotations that imply their programs obey the security policy.

2. The security policy is compiled into a signature that gives each security-sensitive function a type that restricts the use of that function. Because we compile into a typed language, a single mechanism, the type-checker, is responsible for verifying both the fundamental memory safety and type abstraction properties as well as the more sophisticated security properties. All checks can be performed during a single pass over the program.

3. The language of types specifies the interface to programs allowing separate compilation, independent checking, and safe linking of client programs. Therefore, program instrumentation and optimization can be performed offline and the results can be checked when the program is downloaded. The program instrumentation and optimization tools are not part of the trusted computing base.

4. Because security constraints are compiled into a typed language, we can leverage extensive research in type-directed compilation to compile instrumented programs into low-level typed languages. We believe we can extend the techniques developed by Morrisett *et al.* [12] and produce secure Typed Assembly Language.

The remaining sections of this paper describe the approach in more detail. We begin by describing a source language for program instrumentation (Section 2). Next, in Section 3, we present a formal model for security automata based on work by Alpern and Schneider [1, 19]. At this point, we define a typed language, $\lambda_{\mathcal{A}}$, for encoding security automata and specify how to construct a signature that will specialize the language so that it enforces the policy specified by a particular automaton (Section 4). In Section 5, we show how to instrument insecure programs and discuss possible optimizations to the procedure. The last section discusses related and future work.

2 The Insecure Source Language

Before we can describe security automata formally, we must describe the language that we wish to make secure. For the purposes of this paper, we use a simply-typed lambda calculus augmented with a finite set of base types and some constants. The syntax of the language appears in Figure 1.

There are two classes of constants, a countably infinite set of objects a of base type and finite set of function constants f. The latter denotes potentially insecure operations, for example, operations

base types	b	$\in$	BaseType
types	τ	$::=$	$b \mid (\tau_1, \ldots, \tau_n) \to 0$
constants	a	$\in$	A
protected ops	f	$\in$	F
value vars	g, x	$\in$	ValueVar
values	v	$::=$	$a \mid f \mid x \mid$ $\texttt{fix}\, g(x_1{:}\tau_1, \ldots, x_n{:}\tau_n).e$
expressions	e	$::=$	$v_0(v_1, \ldots, v_n) \mid \texttt{halt}$

Figure 1: Source Language Syntax

Q	a finite or countably infinite set of states
q_0	a distinguished initial state
bad	the single "bad" state
F	a finite set of function symbols (f)
A	a countable set of constants (a)
δ	a computable (deterministic) total function: $\mathrm{F} \to (Q \times \vec{\mathrm{A}}) \to Q$

Figure 2: Elements of a Security Automaton

that send bits on the network or read files. We will refer to these functions as the "protected functions" or "protected operations" and in the next section we will show how to use security automata to restrict access to them. Ordinary (unrestricted) functions may be constructed using the notation:

$$\texttt{fix}\, g(x_1{:}\tau_1, \ldots, x_n{:}\tau_n).e$$

where the arguments x_1 to x_n have types τ_1 to τ_n and g, the name of the function itself, may appear recursively in e. When an argument is unused or a function is not recursive, we will often use an underscore in place of the name of the argument or function.

The only distinctive element of our lambda calculus is that it is presented in *continuation-passing style* (CPS) [18]. Functions written in a CPS style never "return" to their caller; instead, they are passed an auxiliary function, or *continuation*. When the function has completed its computation, it calls this continuation. We signify the fact that our functions never return using the notation "$\to 0$" in their type. The symbol `halt` terminates computation. Although CPS is not strictly necessary, this decision will simplify the presentation of the target language. For the remainder of the paper, we will assume familiarity with CPS.

We will assume a static semantics for the language given by a typing judgements $\Gamma \vdash v : \tau$ and $\Gamma \vdash e$ where Γ is a finite map from value variables to types. The former judgement concludes value v is well-formed and has type τ while the latter judgement says simply that e is well-formed. CPS expressions do not return values and consequently, their judgements need not specify a type.

The types for constants are given by a signature $\mathcal{C}_{source}$ where $\mathcal{C}_{source}(a)$ has the form b and $\mathcal{C}_{source}(f)$ has the form $(b_1, \ldots, b_n, (b) \to 0) \to 0$. For the purposes of this paper, the protected operations f operate on objects of base type and accept a single continuation. This restriction is not a fundamental limitation of the work, but allowing higher-order protected operations does cause some complications.[1] Otherwise, the typing rules for the language are completely standard and have been omitted.

We also assume a standard small-step operational semantics for our language denoted by $e_1 \longmapsto e_2$. Function constants must obey their signature and are assumed to be total. Hence:

$$\begin{array}{l} \text{if } \mathcal{C}_{source}(f) = (b_1, \ldots, b_n, (b) \to 0) \to 0 \\ \text{then } f(a_1, \ldots, a_n, v_{cont}) \longmapsto v_{cont}(a) \\ \qquad\qquad \text{(for some } a \text{ with type } b) \end{array}$$

when a_i has type b_i. As we mentioned above, `halt` terminates computation. In other words, there does not exist an expression e such that $\texttt{halt} \longmapsto e$. We will use this instruction to terminate programs that misbehave. Once again, the remaining operational rules are standard and have been omitted.

2.1 An Example: The Taxation Applet

Using our simple source language, we can write a "taxation applet." When invoked, this applet sends a request out for tax forms. After sending the request, the applet reads a private file containing the customer salary before computing the taxes owed. We will assume files and integers (*int*) are available as base types; *send* (across the network for the tax forms) and *read* (file) are the two protected operations.[2]

$$\begin{array}{lll} \multicolumn{3}{l}{\texttt{fix}\,\_(secret{:}file, x_{cont}{:}(int) \to 0).} \\ \quad \texttt{let}\,\_ \quad = send()\ \texttt{in} & & \texttt{\% send for forms} \\ \quad \texttt{let}\, salary = read(secret)\ \texttt{in} & & \texttt{\% read salary} \\ \quad \texttt{let}\, taxes \ = salary\ \texttt{in} & & \texttt{\% compute taxes!!} \\ \quad x_{cont}(taxes) & & \end{array}$$

The customer would like to ensure information about his or her salary is not leaked on the network. In the following sections, we will satisfy the customer by showing how to specify and enforce the "no send after read" policy that was discussed informally in the introduction.

3 Security Automata

Our definition of security automata is derived from the work of Alpern and Schneider [1] who use similar automata to define and reason about safety and liveness properties. Schneider [19] extended this work by defining the class of security enforcement mechanisms that monitor program execution. He proved that these enforcement mechanisms are specified by security automata and that they can enforce any safety property.

[1] The needed extensions complicate the semantics of the target language, but offer little additional insight into the solution of the problem. See Section 4.4 for more explanation.

[2] In this example and others, we use the notation $\texttt{let}\, x_1, \ldots, x_n = v(v_1, \ldots, v_n)\ \texttt{in}\, e$ as an abbreviation for the function application:

$$v(v_1, \ldots, v_n, \texttt{fix}\,\_(x_1{:}\tau_1, \ldots, x_n{:}\tau_n).e)$$

For our purposes, a security automaton ($\mathcal{A}$) is a 6-tuple containing the fields summarized in Figure 2. Like an ordinary finite automaton, a security automaton has a finite or a countably infinite set of states Q, and a distinguished initial state q_0. The automaton also has a single bad state (*bad*). Entrance into the bad state indicates that the security policy has been violated. All other states are considered "good" or "accepting" states. The automaton's inputs correspond to the application of a function symbol f to arguments $a_1, \ldots, a_n$ where f is taken from the set F (dictated by the insecure language) and $a_1, \ldots, a_n$ are taken from the set A.

A security automaton defines allowable program behaviours by specifying a transition function δ. Formally, δ is a deterministic, total function with a signature $\mathrm{F} \to (Q \times \vec{\mathrm{A}}) \to Q$ where $\vec{\mathrm{A}}$ denotes the set of lists $a_1, \ldots, a_n$. Upon receiving an input $f(a_1, \ldots, a_n)$, an automaton makes a transition from its current state to the next state as dictated by this transition function. If the security policy permits the operation $f(a_1, \ldots, a_n, v_{cont}) \longmapsto v_{cont}(a)$ in the current state then the next state will be one of the "good" ones. On the other hand, if the security policy disallows the action $f(a_1, \ldots, a_n, v_{cont}) \longmapsto v_{cont}(a)$ then $\delta(f)(q, a_1, \ldots, a_n)$ will equal *bad*. All programs must respect the basic typing rules so at minimum, for all f, q, a_1, …, a_n there exists a state q' such that q' is not *bad* and $\delta(f)(q, a_1, \ldots, a_n) = q'$ only if $\mathcal{C}(f) = (b_1, \ldots, b_n, (b) \to \mathtt{0}) \to \mathtt{0}$ and $\mathcal{C}(a_i) = b_i$. Furthermore, once the automaton enters the *bad* state, it stays there. Formally, for all f and $a_1, \ldots, a_n$, $\delta(f)(bad, a_1, \ldots, a_n) = bad$. Finally, the transition function δ must be computable. Moreover, the implementor of the security policy must supply code for a family of functions δ_f such that $\delta_f(q_1, a_1, \ldots, a_n)$ equals $\delta(f)(q_1, a_1, \ldots, a_n)$. In the following sections, we will use these functions to instrument untrusted code with security checks.

The language accepted by the automaton $\mathcal{A}$, written $\mathcal{L}(\mathcal{A})$, is a set of strings where a string is a finite sequence of symbols $s_1, \ldots, s_n$ and each symbol s_i is of the form $f(a_1, \ldots, a_m)$. The string $s_1, \ldots, s_n$ belongs to $\mathcal{L}(\mathcal{A})$ if $Accept(q_0, s_1, \ldots, s_n)$ where *Accept* is the least predicate such that:

Definition 1 (String Acceptance) *For all states q and (possibly empty) sequences of symbols $s_1, \ldots, s_n$, $Accept(q, s_1, \ldots, s_n)$ if $q \neq bad$ and:*

1. *$s_1, \ldots, s_n$ is the empty sequence or*
2. *$s_1 = f(a_1, \ldots, a_m)$ and $\delta(f)(q, a_1, \ldots, a_m) = q'$ and $Accept(q', s_2, \ldots, s_n)$*

We are now in a position to define the security policy for our taxation applet. Informally, the policy we desire is "no network send after any file has been read." Furthermore, for the sake of future examples, access to some files will be restricted; in order to determine whether or not the applet has been granted access to the file f, we will have to invoke the function *read*?(f), which has been provided by the implementor of the file system.

The corresponding security automaton has three states: *start*, the initial state; *has_read*, the state we enter after any file read; and, of course, the *bad* state. The protected operations, F, are *send* and *read* and the constants, A, include all files and the integers. The transition function δ, written in pseudo-code, is the following:

$$
\begin{array}{l}
\delta_{send}(q) \;=\\
\quad \mathtt{if}\; q = start\; \mathtt{then}\\
\quad\quad start\\
\quad \mathtt{else}\\
\quad\quad bad
\end{array}
$$

$$
\begin{array}{l}
\delta_{read}(q, f) \;=\\
\quad \mathtt{if}\; q = start \wedge read?(f)\; \mathtt{then}\\
\quad\quad has\_read\\
\quad \mathtt{else\ if}\; q = has\_read \wedge read?(f)\; \mathtt{then}\\
\quad\quad has\_read\\
\quad \mathtt{else}\\
\quad\quad bad
\end{array}
$$

4 The Secure Target Language

So far, we have defined two languages, a lambda calculus for writing applications and a specification language for describing security properties. This section presents a third language, $\lambda_{\mathcal{A}}$, that serves as a target for the compilation of the other two. The security automaton specifications are compiled into an interface that gives types to the protected functions and the corresponding automaton functions (*i.e.* δ_{send}). The types on protected functions specify sufficient preconditions that the type system for the new language can enforce the security policy. Application programs, such as the taxation applet, are compiled into $\lambda_{\mathcal{A}}$ expressions by inserting typing annotations and run-time checks. These instrumented programs type check against the interface, implying they obey the security policy.

In the following section, we will describe the main constructs in the language $\lambda_{\mathcal{A}}$ independently of any security policy. In section 4.2, we will show how to specialize the language by constructing the typed interface for a particular security automaton. Section 5 describes an algorithm for instrumenting application programs.

4.1 The Syntax and Semantics of $\lambda_{\mathcal{A}}$

The secure language contains three main parts: the predicates P, the types τ, and the term level constructs. We will explain each of these parts in succession. Figure 3 presents the syntax of the entire language.

Predicates Predicates, P, may be variables ρ or ϱ, indices (constant predicates) ι, or functions of a number of arguments $\iota(P_1, \ldots, P_n)$. There are three distinct kinds of predicates:

- Predicates that correspond to values of base type (kind `Val`).
- Predicates that correspond to security automaton states (kind `State`).
- Predicates that describe relations between values and/or states (kind $(\kappa_1, \ldots, \kappa_n) \to \mathcal{B}$ where $\mathcal{B}$ is the boolean kind).

kinds	κ	$::=$	$\texttt{Val} \mid \texttt{State} \mid \mathcal{B} \mid (\kappa, \ldots, \kappa) \to \mathcal{B}$
indices	$\iota, \hat{a}, \hat{q}$		
index sig	$\mathcal{I}$	$:$	*indices* $\to$ *kinds*
pred. vars	ρ, ϱ		
predicates	P	$::=$	$\rho \mid \iota \mid \iota(P_1, \ldots, P_n)$
predicate ctxt	Δ	$::=$	$\cdot \mid \Delta, \rho{:}\kappa \mid \Delta, P$
base types	b	$\in$	$\mathrm{BaseType} + \texttt{S}$
types	τ	$::=$	$b(P) \mid \forall[\Delta].(P, \tau_1, \ldots, \tau_n) \to \texttt{0} \mid \exists\rho{:}\kappa.\tau$
constants	a, q, f		
constant sig	$\mathcal{C}$	$:$	*constants* $\to$ *types*
value vars	g, x		
values	v	$::=$	$x \mid a \mid \texttt{fix}\, g[\Delta].(P, x_1{:}\tau_1, \ldots, x_n{:}\tau_n).e \mid v[P] \mid v[\cdot] \mid \texttt{pack}[P, v]\ \mathit{as}\ \tau$
expressions	e	$::=$	$v_0(v_1, \ldots, v_n) \mid \texttt{halt} \mid \texttt{let}\, \rho, x = \texttt{unpack}\, v\, \texttt{in}\, e \mid \texttt{if}\, v\, (q \to e_1 \mid \_ \to e_2)$

Figure 3: Syntax of $\lambda_{\mathcal{A}}$

Most security policies depend upon the properties of particular values, and consequently, we must track these values in the type system. This is the role of the predicates with kind `Val`. For each value a of base type, there is a corresponding index that we will write using the notation $\hat{a}$. For example, the file foo has an associated index $\widehat{foo}$. Using the index $\widehat{foo}$, we can specify precise properties of foo including "foo is readable" or "foo is writeable." In our examples, we will use the meta-variable ρ for predicate variables that range over indices of kind `Val`.

We are careful to distinguish between indices and values in order to separate computation performed at compile time (type checking) and computation performed at run time. Indices, and more generally, predicates are used exclusively at compile time; we erase these annotations before running the program. If we want to use the information contained in a predicate at run time, we must use the corresponding value instead. This design has the advantage that we need only pass run-time data around when we actually need it rather than because we required it to type check the program. Furthermore, a type-erasure semantics is a practical necessity if the type system is to be used in low-level certified code [12, 2].

The second kind of predicate allows us to specify information about automaton states. Again, for every automaton state q, there is an associated index $\hat{q}$. We will usually use the predicate $\hat{q}$ to indicate the program is currently executing in state automaton q. Sometimes in our examples (and in particular for the states *start* and *has_read*), we will omit the hat notation; context is enough to discriminate between states and their associated indices. The meta-variable ϱ will range over variables of kind `State`.

Finally, the language of predicates contains a set of relations. These relations will serve two purposes. The first purpose is to describe the transition function of the security automaton. For example, the transition function for the automaton of Section 3 can be described by two predicates, $\delta_{send}(P_{q_1}, P_{q_2})$ and $\delta_{read}(P_{q_1}, P_{q_2}, P_{file})$. The former predicate may be read "in state P_{q_1}, executing the *send* operation causes a transition to state P_{q_2}." The latter predicate is similar. The predicate $\delta_{send}(start, start)$ states that performing a *send* operation in the *start* state causes a transition to the *start* state. These predicates will be discussed in more detail in the next section when we describe how to instantiate the language signature for a particular automaton.

The second relation we will need is the special predicate $\neq(\cdot,\cdot)$ of kind $(\texttt{State}, \texttt{State}) \to \mathcal{B}$. We will use this predicate to denote the fact that some state q is not equal to the *bad* state and consequently that it is safe to execute an operation that causes a transition into q.

We specify the well-formedness of predicates using the judgement $\Phi \vdash P : \kappa$ where Φ is a type-checking context containing three components: a predicate context Δ, a finite map Γ from value variables to types, and another predicate P' indicating the current state of the automaton. The latter two components are not used for specifying the well-formedness of predicates (they will be used for type-checking terms). The signature $\mathcal{I}$ assigns kinds to the indices. Kinds for variables are determined by the predicate context Δ. The kind of a function symbol must agree with the kinds of the predicates to which it is applied. The formal rules are uninteresting so we have removed them to Appendix A.

Figure 4 gives the rules for provability of predicates. The judgement $\Phi \vdash P$ indicates that the boolean-valued predicate P is true, and the judgement $\Phi \vdash P'$ in_state indicates that the program is currently executing in the automaton state P'. We will elaborate on how these judgements are used when we describe the static semantics of functions.

Aside from the special predicate $\neq(\cdot,\cdot)$, our predicates are completely uninterpreted. This decision makes it trivial to show the decidability of the type system. However, some optimizations may not be possible without a stronger logic. To remedy this situation, implementers are free to add axioms to the type system provided they also supply a decision procedure for the richer logic. In Section 5.2, we show how to add security policy-specific axioms that allow many unnecessary security checks to be eliminated.

Types As mentioned above, security policies often depend upon the properties of particular values. In order to reflect values into the type structure, we use singleton types $b(P)$ where P is either a index or

$\boxed{\Phi \vdash P \qquad \Phi \vdash P \text{ in\_state}}$

$$\frac{}{\Delta_1, P, \Delta_2; \Gamma; P' \vdash P} \tag{1}$$

$$\frac{}{\Phi \vdash \neq (\hat{q}, \hat{q}')} \; (\hat{q} \neq \hat{q}') \tag{2}$$

$$\frac{}{\Delta; \Gamma; P \vdash P \text{ in\_state}} \tag{3}$$

Figure 4: Static Semantics: Provability

a variable of kind `Val` and b is one of base types. For example, our file foo will be assigned the type $file(\widehat{foo})$.

In many situations, we will not be able to infer the state of the security automaton statically. As a result, we will have to pass representations of automaton states around at run time and check them dynamically to determine their values. These state representations will have the singleton type $\mathtt{S}(P)$ where P has kind `State` and `S` is the new base type for automaton states.

In many circumstances, we may not know or even care which state or value we are manipulating. For instance, the math library may not contain any security-sensitive operations. We would simply like these functions to manipulate integers without being specific about which ones. In this case, we will use the existential type $\exists\rho{:}\mathtt{Val}.int(\rho)$ to indicate we have an integer, but we do not know which one. As we will see in Section 5, this existential encodes the generic (non-singleton) base types from the source language.

The final type constructor is a modified function type $\forall[\Delta].(P, \tau_1, \ldots, \tau_n) \rightarrow \mathtt{0}$. The predicate context Δ abstracts a series of predicate variables for unknown values or states and requires that a sequence of boolean-valued predicates be satisfied before the function can be invoked. The predicate P in the first argument position is not an argument to the function. Rather, it is another precondition requiring that the function be called in the state associated with P. The actual arguments to the function must have types τ_1 through τ_n.

We specify the well-formedness of types using the judgement $\Phi \vdash \tau$. A type is well-formed if Δ contains the free predicate variables of the type. Function types $\forall[\Delta].(P, \tau_1, \ldots, \tau_n) \rightarrow \mathtt{0}$ also require that P has kind `State` and that the predicates occuring in Δ have kind $\mathcal{B}$. Once again, the formal rules may be found in Appendix A. Because predicates are uninterpreted, we can use standard syntactic equality of types up to alpha conversion of bound variables.

Values and Expressions The typing rules for values have the form $\Phi \vdash v : \tau$ and state that the value v has type τ in the given context. The judgement $\Phi \vdash e$ states that the expression e is well-formed. Recall that CPS expressions do not return values, and hence the latter judgement is not annotated with a return type. Figure 5 presents the formal rules. In these judgements, we use the notation $\Phi, \rho{:}\kappa$ to denote a new context in which the binding $\rho{:}\kappa$ has been appended to the list of assumptions in Φ. The operation is undefined if ρ appears in Φ. The notations Φ, P and $\Phi, x{:}\tau$ and the extension to Φ, Δ are similar, although P may already appear in Φ.

The values include variables and constants. The treatment of values is standard, and as in the insecure language, a signature $\mathcal{C}$ gives types to the constants.

The value $v[P]$ is the instantiation of the polymorphic value v with the predicate P. We consider this instantiation a value because predicates are used only for type-checking purposes; they have no run-time significance. The value $v[\cdot]$ is somewhat similar: if v has type $\forall[P, \Delta].(\cdots) \rightarrow \mathtt{0}$ and we can prove the predicate P is valid in the current context, we give $v[\cdot]$ the type $\forall[\Delta].(\cdots) \rightarrow \mathtt{0}$. Again, the notation $[\cdot]$ is used only to specify that the type-checker should attempt to prove the precondition; it will not influence the execution of programs. In a system with a more sophisticated logic than the one presented in this paper, we might not want to trust the correctness of complex decision procedures for the logic. In this case, we would replace $[\cdot]$ with a proof of the precondition and replace the type checker's decision procedure with a much simpler proof-checker.

Target language function values differ from source language functions in that they specify a list of preconditions using the predicate context Δ. Every function also expresses a state precondition P. The function can only be called in the state denoted by P. Static semantics rule (10) contains the the judgement $\Phi \vdash P$ in_state, which ensures this invariant is maintained. This rule also enforces the standard constraints that argument types must match the types of the formal parameters. Finally, because the predicate context is empty, any preconditions the function might have specified must have already been proven valid.

Rule (6) states that we type check the body of a function assuming its preconditions hold. In this rule, we use the notation $\Phi \leftarrow P$ to denote a context Φ' in which the state component of Φ has been *replaced* by P. For example, suppose a function g is defined in the context $\Delta'; \Gamma'; P'$. The type checker can use any of the predicates in Δ' to help prove g is well-formed but it cannot assume that g will be called in the state P'. The function g is defined here, but may not be used until much later in the computation when the state is different (P'' perhaps).

It is tempting to define a predicate "*in_state*(P)" and to include this predicate in the list of function preconditions Δ. Using this mechanism, it may appear as though we could eliminate the special-purpose state component of the type-checking context. Unfortunately, this simplification is unsound. Consider the following informal example:

`% Assume current state =` $start$
`let` $g{:}\forall[in\_state(start)].(\tau_1, \ldots, \tau_n) \rightarrow \mathtt{0} = \cdots$ `in`
`% Prove precondition:`
`let` $g'{:}\forall[\,].(\tau_1, \ldots, \tau_n) \rightarrow \mathtt{0} = g[\cdot]$ `in`
`% Change the state to` q' `where` $q' \neq start$:
`let` $\_ = op()$ `in`
`%` g `is not called in the` $start$ `state!`
$g'(v_1, \ldots, v_n)$

In the last line, the function g is invoked in a state q' when the function definition assumed it would be in-

$\boxed{\Phi \vdash v : \tau}$

$$\frac{}{\Phi \vdash x : \tau}\ (\Phi(x) = \tau) \tag{4}$$

$$\frac{}{\Phi \vdash a : \tau}\ (\mathcal{C}(a) = \tau) \tag{5}$$

$$\frac{\begin{array}{c}\Phi \vdash \tau_g \\ (\Phi, \Delta, g{:}\tau_g, x_1{:}\tau_1, \ldots, x_n{:}\tau_n) \leftarrow P \vdash e\end{array}}{\Phi \vdash \mathtt{fix}\, g[\Delta].(P, x_1{:}\tau_1, \ldots, x_n{:}\tau_n).e : \tau_g} \tag{6}$$

$$(\text{where } \tau_g = \forall[\Delta](P, \tau_1, \ldots, \tau_n) \rightarrow \mathsf{0})$$

$$\frac{\Phi \vdash v : \forall[\rho{:}\kappa, \Delta].(P', \tau_1, \ldots, \tau_n) \rightarrow \mathsf{0} \quad \Phi \vdash P : \kappa}{\Phi \vdash v[P] : (\forall[\Delta].(P', \tau_1, \ldots, \tau_n) \rightarrow \mathsf{0})[P/\rho]} \tag{7}$$

$$\frac{\Phi \vdash v : \forall[P, \Delta].(P', \tau_1, \ldots, \tau_n) \rightarrow \mathsf{0} \quad \Phi \vdash P}{\Phi \vdash v[\cdot] : \forall[\Delta].(P', \tau_1, \ldots, \tau_n) \rightarrow \mathsf{0}} \tag{8}$$

$$\frac{\Phi \vdash P : \kappa \quad \Phi \vdash v : \tau[P/\rho]}{\Phi \vdash \mathtt{pack}[P, v]\ \mathit{as}\ \exists\rho{:}\kappa.\tau : \exists\rho{:}\kappa.\tau} \tag{9}$$

$\boxed{\Phi \vdash e}$

$$\frac{\begin{array}{c}\Phi \vdash v_0 : \forall[].(P, \tau_1, \ldots, \tau_n) \rightarrow \mathsf{0} \\ \Phi \vdash v_1 : \tau_1 \quad \cdots \quad \Phi \vdash v_n : \tau_n \\ \Phi \vdash P \text{ in\_state}\end{array}}{\Phi \vdash v_0(v_1, \ldots, v_n)} \tag{10}$$

$$\frac{}{\Phi \vdash \mathtt{halt}} \tag{11}$$

$$\frac{\Phi \vdash v : \exists\rho{:}\kappa.\tau \quad \Phi, \rho{:}\kappa, x{:}\tau \vdash e}{\Phi \vdash \mathtt{let}\, \rho, x = \mathtt{unpack}\, v\ \mathtt{in}\, e} \tag{12}$$

$$\frac{\begin{array}{cc}\Phi \vdash v : \mathtt{S}(\rho) & \Phi \vdash q : \mathtt{S}(\hat{q}) \\ \Phi' \vdash e_1[\hat{q}/\rho] & \Phi, \neq(\rho, \hat{q}) \vdash e_2\end{array}}{\Phi \vdash \mathtt{if}\, v\, (q \rightarrow e_1 \mid \_ \rightarrow e_2)} \tag{13}$$

$$\begin{array}{l}(\text{where } \Phi = \Delta, \rho{:}\mathtt{State}, \Delta'; \Gamma; P \\ \text{and } \Phi' = \Delta, (\Delta'[\hat{q}/\rho]); \Gamma[\hat{q}/\rho]; P[\hat{q}/\rho])\end{array}$$

$$\frac{\begin{array}{c}\Phi \vdash v : \mathtt{S}(\hat{q}) \quad \Phi \vdash q : \mathtt{S}(\hat{q}) \\ \Phi \vdash e_1\end{array}}{\Phi \vdash \mathtt{if}\, v\, (q \rightarrow e_1 \mid \_ \rightarrow e_2)} \tag{14}$$

$$\frac{\begin{array}{c}\Phi \vdash v : \mathtt{S}(P) \quad \Phi \vdash q : \mathtt{S}(\hat{q}) \\ \Phi, \neq(P, \hat{q}) \vdash e_2\end{array}}{\Phi \vdash \mathtt{if}\, v\, (q \rightarrow e_1 \mid \_ \rightarrow e_2)} \begin{pmatrix} P \neq \rho \\ P \neq \hat{q}\end{pmatrix} \tag{15}$$

Figure 5: Static Semantics: Values and Expressions

voked in the *start* state. The example highlights the main difference between the state predicates and the others: The validity of the predicates in Δ is invariant throughout the execution of the program whereas the validity of a state predicate varies during execution because it depends implicitly on the current state of the machine.

Existential values are handled in standard fashion. The value $\mathtt{pack}[P, v]\ \mathit{as}\ \exists\rho{:}\kappa.\tau$ creates an existential package that hides P in τ using ρ. The corresponding elimination form, $\mathtt{let}\, \rho, x = \mathtt{unpack}\, v'\ \mathtt{in}\, e$ unpacks the existential v', substituting v for x and P for ρ into the remaining expression e. As with polymorphic types, we assume a type-erasure interpretation of existentials.

Finally, the conditional $\mathtt{if}\, v\, (q \rightarrow e_1 \mid \_ \rightarrow e_2)$ tests an automaton state v to determine whether v is the state q or not. If so, the program executes e_1 (see rule (13)) and if not, the program executes e_2. A variant of Harper and Morrisett's typecase [3] operator, $\mathtt{if}$ also performs type refinement. If v has type $\mathtt{S}(\rho)$ then it refines the type-checking context with the information that $\rho = \hat{q}$ by substituting $\hat{q}$ for ρ. On the other hand, if v is not q, the second branch is taken and the context is refined with the information $\neq (\rho, \hat{q})$. Programs can use this mechanism to dynamically check whether or not they are about to enter the bad state and prevent it.

There is no need to use the $\mathtt{if}\, v$ construct if we know which state a value v represents. For example, we know the expression $\mathtt{if}\, q\, (q \rightarrow e_1 \mid \_ \rightarrow e_2)$ will reduce to e_1 and therefore e_2 is dead code and the test is wasted computation. However, during the proof of soundness of the type system, such configurations arise and cause difficulties. To avoid these difficulties, we follow the strategy of Crary *et al.* [2] and add the *trivialization* rules (14) and (15) which deal with these redundant cases. Each rule type checks only the branch of the if statement that will be taken.

Operational Semantics The operational semantics for the language is given by the relation $e \longmapsto_s e'$ (see Figure 6). The symbol s is either empty $(\cdot)$ or it is a protected function symbol applied to some number of arguments $(f(a_1, \ldots, a_n))$. Most operations, and, in fact, all of the operations shown in Figure 6, emit the empty symbol. However, this figure does not show the operation of the protected functions. In the next section, we will explain the operational semantics of the protected functions f in the context of a signature for a particular security automaton.

We have given a typed operational semantics to facilitate the proof of soundness of the system. However, inspection of the rules will reveal that evaluation does not depend upon types or predicates, provided the expressions are well-formed. Therefore we can type-check a program and then erase the types before executing it.

4.2 The Security Automaton Signature

In order to specialize the generic language, we construct a typed interface or *signature* for the constants in the language. The signature for a security automaton $\mathcal{A}$, shown in Figure 7, has three parts: the type

$$v(v_1,\ldots,v_n) \longmapsto.\ e_m[v,v_1,\ldots,v_n/g,x_1,\ldots,x_n]$$

if $v = v'\,\phi_1\cdots\phi_m$
and $v' = \mathtt{fix}\,g[\theta_1,\ldots,\theta_m].(x_1{:}\tau_1,\ldots,x_n{:}\tau_n).e_0$
and for $1 \le i \le m$,
$\phi_i = [\cdot]$ and $\theta_i = P_i$ and $e_i = e_{i-1}$, or
$\phi_i = [P'_i]$ and $\theta_i = \rho_i{:}\kappa_i$ and $e_i = e_{i-1}[P'_i/\rho_i]$

$$\mathtt{let}\,\rho,x = \mathtt{unpack}\,(\mathtt{pack}[P,v]\ \mathit{as}\ \tau)\,\mathtt{in}\,e \longmapsto.\ e[P,v/\rho,x]$$

$$\mathtt{if}\,q'\,(q \to e_1 \mid \_ \to e_2) \longmapsto.\ e_1 \qquad \text{if } q' = q$$

$$\mathtt{if}\,q'\,(q \to e_1 \mid \_ \to e_2) \longmapsto.\ e_2 \qquad \text{if } q' \neq q$$

Figure 6: Operational Semantics for $\lambda_{\mathcal{A}}$

$\boxed{\text{Type and Value Signature}}$

$$\begin{array}{lcl}
\mathcal{I}(\hat{a}) & = & \mathtt{Val} \quad \text{for } a \in \mathrm{A} \\
\mathcal{I}(\hat{q}) & = & \mathtt{State} \quad \text{for } q \in Q \\
\mathcal{I}(\delta_f) & = & (\mathtt{State},\mathtt{State},\overbrace{\mathtt{Val},\ldots,\mathtt{Val}}^{n}) \to \mathcal{B} \\
& & \quad \text{if } \mathcal{C}_{insecure}(f) = (b_1,\ldots,b_n,(b)\to\mathsf{0})\to\mathsf{0} \\
\mathcal{C}_{target}(a) & = & b(\hat{a}) \quad \text{if } \mathcal{C}_{insecure}(a) = b \\
\mathcal{C}_{target}(q) & = & \mathtt{S}(\hat{q}) \quad \text{if } q \in Q \\
\mathcal{C}_{target}(\delta_f) & = & \forall[\varrho_1{:}\mathtt{State},\rho_1{:}\mathtt{Val},\ldots,\rho_n{:}\mathtt{Val},\neq(\varrho_1,\mathit{bad})].(\varrho_1,S(\varrho_1),b_1(\rho_1),\ldots,b_n(\rho_n), \\
& & \quad \forall[\varrho_2{:}\mathtt{State},\delta_f(\varrho_1,\varrho_2,\rho_1,\ldots,\rho_n)](\varrho_1,S(\varrho_2))\to\mathsf{0})\to\mathsf{0} \\
& & \text{if } \mathcal{C}_{insecure}(f) = (b_1,\ldots,b_n,(b)\to\mathsf{0})\to\mathsf{0} \\
\mathcal{C}_{target}(f) & = & \forall[\varrho_1{:}\mathtt{State},\varrho_2{:}\mathtt{State},\rho_1{:}\mathtt{Val},\ldots,\rho_n{:}\mathtt{Val},\neq(\varrho_2,\mathit{bad}),\delta_f(\varrho_1,\varrho_2,\rho_1,\ldots,\rho_n)]. \\
& & \quad (\varrho_1,b_1(\rho_1),\ldots,b_n(\rho_n),\forall[].(\varrho_2,\exists\rho{:}\mathtt{Val}.b(\rho))\to\mathsf{0})\to\mathsf{0} \\
& & \text{if } \mathcal{C}_{insecure}(f) = (b_1,\ldots,b_n,(b)\to\mathsf{0})\to\mathsf{0}
\end{array}$$

$\boxed{\text{Operational Signature}}$

$$\delta_f[\hat{q}_1][\hat{a}_1]\cdots[\hat{a}_n][\cdot](q_1,a_1,\ldots,a_n,v_{cont}) \longmapsto.\ v_{cont}[\hat{q}_2][\cdot](q_2)$$

if $\delta(f)(q_1,a_1,\ldots,a_n) = q_2$
and for $1 \le i \le n, \mathcal{C}_{insecure}(a_i) = b_i$
and $\mathcal{C}_{insecure}(f) = (b_1,\ldots,b_n,(b)\to\mathsf{0})\to\mathsf{0}$

$$f[\hat{q}_1][\hat{q}_2][\hat{a}_1]\cdots[\hat{a}_n][\cdot][\cdot](a_1,\ldots,a_n,v_{cont}) \longmapsto_{f(a_1,\ldots,a_n)} v_{cont}(\mathtt{pack}[\hat{a},a]\ \mathit{as}\ \exists\rho{:}\mathtt{Val}.b(\rho))$$
(for some a such that $\mathcal{C}_{insecure}(a) = b$)

if for $1 \le i \le n, \mathcal{C}_{insecure}(a_i) = b_i$
and $\mathcal{C}_{insecure}(f) = (b_1,\ldots,b_n,(b)\to\mathsf{0})\to\mathsf{0}$

Figure 7: Signature for $\lambda_{\mathcal{A}}$

signature $\mathcal{I}$, the value signature $\mathcal{C}$, and the operational signature. These signatures are derived from the insecure language signature $\mathcal{C}_{insecure}$ for constants and from the security automaton specification.

The type signature gives each constant $\hat{a}$ from the insecure language and state $\hat{q}$ from the security automaton kinds `Val` and `State` respectively. Furthermore, for each protected function f, the type signature specifies a predicate δ_f. The invariant the type system ensures is that for all $\hat{q_1}, \hat{q_2}, \hat{a}_1, \ldots, \hat{a}_n$, we will be able to prove the predicate $\delta_f(\hat{q_1}, \hat{q_2}, \hat{a}_1, \ldots, \hat{a}_n)$ only if the corresponding automaton transition holds. In other words, only if:

$$\delta(f)(q_1, a_1, \ldots, a_n) = q_2$$

The value signature specifies that objects a and states q are given the correct singleton types. For each protected function symbol f in the insecure language, the signature gives types to a pair of function values. The function δ_f is supplied by the implementer of the security policy; it is used to dynamically determine the automaton state transition function given the program executes the function f in the state ϱ_1 with arguments identified by $\rho_1, \ldots, \rho_n$. When δ_f has computed the transition, it calls its continuation, passing it the next state ϱ_2 so the continuation can test this state to determine whether it is the *bad* state. The continuation assumes the predicate $\delta_f(\varrho_1, \varrho_2, \rho_1, \ldots, \rho_n)$. Before calling the function f itself, we require that the type-checker be able to prove that f will make a transition to some state other than the bad state. Hence the precondition on f states that we must know the automaton transition we are making ($\delta_f(\varrho_1, \varrho_2, \rho_1, \ldots, \rho_n)$) and moreover that that the new state ϱ_2 is not equal to *bad*.

4.3 Properties of $\lambda_{\mathcal{A}}$

A predicate P is valid with respect to an automaton $\mathcal{A}$, written $\mathcal{A} \models P$, if:

- P is $\neq (\hat{q}, \hat{q}')$ and $q \neq q'$, or
- P is $\delta_f(\hat{q_1}, \hat{q_2}, \hat{a}_1, \ldots, \hat{a}_n)$ and $\mathcal{A}.\delta(f)(q_1, a_1, \ldots, a_n) = q_2$

We say that an expression e is *secure* with respect to a security automaton $\mathcal{A}$ in state q, written $\mathcal{A}; q \vdash e$, if

1. $q \neq bad$

and there exist predicates $P_1, \ldots, P_n$ such that:

2. $\mathcal{A} \models P_i$, for $1 \leq i \leq n$
3. $P_1, \ldots, P_n; \cdot; \hat{q} \vdash e$

If we can prove that an expression e is secure in our deductive system then the expression should not violate the security policy when it executes. The soundness theorem below formalizes this notion. The first part of the theorem, *Type Soundness*, ensures that programs obey a basic level of control-flow safety. More specifically, it ensures that expressions do not get *stuck* during the course of evaluation. An expression e is *stuck* if e is not `halt` and there does not exist an e' such that $e \longmapsto_s e'$. Hence, Type Soundness implies a program will only halt when it executes the halt instruction and not because we have applied a function to too few or the wrong types of arguments. The second part of the theorem, *Security*, ensures programs obey the policy specified by the security automaton $\mathcal{A}$. In other words, the sequence of protected operations executed by the program must form a string in the language $\mathcal{L}(\mathcal{A})$. In this second statement, we use the notation $|s_1, \ldots, s_n|$ to denote the subsequence of the symbols $s_1, \ldots, s_n$ with all occurences of $\cdot$ removed.

Theorem 1 (Soundness)
If $\mathcal{A}; q_0 \vdash e_1$ then

1. (Type Soundness) *For all evaluation sequences $e_1 \longmapsto_{s_1} e_2 \longmapsto_{s_2} \cdots \longmapsto_{s_n} e_{n+1}$, the expression e_{n+1} is not* stuck.
2. (Security) *If $e_1 \longmapsto_{s_1} e_2 \longmapsto_{s_2} \cdots \longmapsto_{s_n} e_{n+1}$ then $|s_1, s_2, \ldots, s_n| \in \mathcal{L}(\mathcal{A})$*

Soundness can be proven syntactically in the style of Wright and Felleisen [23] using the following two lemmas. The proof appears in Appendix B.

Lemma 2 (Progress) *If $\mathcal{A}; q \vdash e$ then either:*

1. $e \longmapsto_s e'$ *or*
2. $e = \texttt{halt}$

Lemma 3 (Subject Reduction) *If $\mathcal{A}; q \vdash e$ and $e \longmapsto_s e'$ then*

1. *if $s = \cdot$ then $\mathcal{A}; q \vdash e'$*
2. *and if $s = f(a_1, \ldots, a_n)$ then $\mathcal{A}; q' \vdash e'$ where $\delta(f)(q, a_1, \ldots, a_n) = q'$*

Finally, inspection of the typing rules will reveal that for any expression or value, there is exactly one typing rule that applies and that the preconditions for the rules only depend upon subcomponents of the terms or values (with possibly a predicate substitution). Judgements for the well-formedness of types and predicates are also well-founded so the type system is decidable:

Theorem 4 *It is decidable whether or not $\Phi \vdash e$.*

4.4 Language Extensions

If security policies depend upon higher-order functions or immutable data structures such as tuples and records, we will have to track the values of these data structures in the type system using singleton types as we did with values of base type. The simplest way to handle this extension is to use an allocation semantics [10, 11]. In this setting, when a function closure $\texttt{fix}\, g[\Delta](\cdots).e$ is allocated, it is bound to a new address (ℓ). Instead of substituting the closure through the rest of the code as we do now, we would substitute the address (ℓ) through the code and give it the singleton type $\tau(\hat{\ell})$ where τ is $\forall[\Delta](\cdots) \to \texttt{0}$. All the other mechanisms remain unchanged. We decided not to present this style of semantics in this paper because it adds extra mechanism but gives little

$$
\begin{array}{lcl}
|b| & = & \exists \rho{:}\texttt{Val}.b(\rho) \\
|(\tau_1,\ldots,\tau_n) \to 0| & = & \forall[\varrho{:}\texttt{State}, \neq(\varrho, bad)].(\varrho, \texttt{S}(\varrho), |\tau_1|,\ldots,|\tau_n|) \to 0 \\
\\
|x| & = & x \\
|a| & = & \texttt{pack}[\hat{a}, a] \ \mathit{as} \ \exists \rho{:}\texttt{Val}.b(\rho) \quad \text{if } \mathcal{C}_{insecure}(a) = b \\
|\texttt{fix}\, g(x_1{:}\tau_1,\ldots,x_n{:}\tau_n).e| & = & \texttt{fix}\, g[\varrho{:}\texttt{State}, \neq(\varrho, bad)].(\varrho, x{:}S(\varrho), x_1{:}|\tau_1|,\ldots,x_n{:}|\tau_n|).|e|_{\varrho,x} \\
|f| & = & \texttt{fix}\,\_[\varrho_1{:}\texttt{State}, \neq(\varrho_1, bad)](\varrho_1, x_0{:}S(\varrho_1), x_1{:}|b_1|,\ldots,x_n{:}|b_n|, x_{n+1}{:}|(b) \to 0|). \\
& & \quad \texttt{let}\, \rho_1, x_1' = \texttt{unpack}\, x_1\, \texttt{in} \\
& & \quad \ldots \\
& & \quad \texttt{let}\, \rho_n, x_n' = \texttt{unpack}\, x_n\, \texttt{in} \\
& & \quad \texttt{let}\, \varrho_2, \delta_f(\varrho_1, \varrho_2, \rho_1,\ldots,\rho_n), x_{\varrho_2} = \delta_f[\varrho_1][\rho_1]\cdots[\rho_n][\cdot](x_0, x_1',\ldots,x_n')\, \texttt{in} \\
& & \quad \texttt{if}\, x_{\varrho_2}(\\
& & \qquad bad \to \texttt{halt} \\
& & \qquad \mid \_ \to \texttt{let}\, x = f[\varrho_1][\varrho_2][\rho_1]\cdots[\rho_n][\cdot][\cdot](x_1',\ldots,x_n')\, \texttt{in} \\
& & \qquad\quad x_{n+1}[\varrho_2][\cdot](x_{\varrho_2}, x)) \\
& & \text{if } \mathcal{C}_{insecure}(f) = (b_1,\ldots,b_n,(b) \to 0) \to 0 \\
\\
|v_0(v_1,\ldots,v_n)|_{P,v} & = & |v_0|[P][\cdot](v, |v_1|,\ldots,|v_n|) \\
|\texttt{halt}\,|_{P,v} & = & \texttt{halt}
\end{array}
$$

Figure 8: Program Instrumentation

additional insight into the solution of the problem. Despite the extra complexity, an allocation semantics is common in low-level typed languages such as Morrisett's Typed Assembly Language [12] that must reason about memory structure.

There are a number of possibilities for handling mutable data structures. The main principle is that if security-sensitive operations depend upon mutable data then the state of that data must be encoded in the state of the automaton. The assignment operator must be designated as a protected operation that changes the state.

5 Program Instrumentation

It is straightforward to design a translation that instruments our insecure source language with security checks (see Figure 8) now that we have set up the appropriate type-theoretic machinery in the secure target language.

The interesting portion of the type translation involves the translation of function types. The static semantics maintains the invariant that programs never enter the bad state and we naturally express this fact as a precondition to every function call. Hence the type translation of the function type $(\tau_1,\ldots,\tau_n) \to 0$ is:

$$\forall[\varrho{:}\texttt{State}, \neq(\varrho, bad)].(\varrho, \texttt{S}(\varrho), |\tau_1|,\ldots,|\tau_n|) \to 0$$

In general, we may not know the current state statically so we quantify over all states ϱ, provided $\varrho \neq bad$. In order to determine state transfers do not go wrong, we will also have to thread a representation of the state ($\texttt{S}(\varrho)$) through the computation.

Most of the work in the value translation is accomplished during the translation of the protected operations. We unpack the arguments so the individual values can be tracked through the type system and then use δ_f to determine the state transition that will occur if we execute f on these arguments. After checking to ensure we do not enter the bad state, we execute f itself passing it a continuation that executes in state ϱ_2. In this translation, we use the abbreviation:

$$\begin{array}{l}\texttt{let}\,\Delta, x_1,\ldots,x_n = v(v_1,\ldots,v_n)\,\texttt{in}\, e \equiv \\ v(v_1,\ldots,v_n, \texttt{fix}\,\_[\Delta].(x_1{:}\tau_1,\ldots,x_n{:}\tau_n).e)\end{array}$$

and we assume predicate and value variables bound by `let` are fresh.

Instrumented programs type check and thus they are *secure* in the sense made precise in the last section:

Theorem 5 *If* $\vdash_{insecure} e$ *then* $\mathcal{A}; q_0 \vdash |e|_{\hat{q_0},q_0}$.

This property can be proven using a straightforward induction on the typing derivation of the source term.

5.1 Instrumenting The Taxation Applet

Figure 9 presents the results of instrumentating the taxation applet from Section 2 with checks from the security automaton of Section 3. We have simplified the output of the formal translation slightly to make it more readable. In particular, we have inlined the functions that the translation wraps around each protected function symbol.

The translation does not assume that the taxation applet is invoked in the initial automaton state and consequently the resulting function abstracts the input state ϱ_1. Also, as specified by the translation, objects of base type, like the file *secret* become existentials. The main point of interest in this example is that before each of the protected operations *send* and *read*, the corresponding automaton function determines the next state. Then the `if` construct checks that these states are not *bad*. If the dynamic check succeeds, the type checker introduces information into the context that allows it to infer that executing the *read* and *send* operations is safe. When reading the code calling *send* or *read*, notice

```
fix _[ϱ1:State, ≠ (ϱ1, bad)]
    (ϱ1, x_ϱ1:S(ϱ1), secret:∃ρ:Val.file(ρ),
    x_cont:τ_cont).
  let ϱ2, δ_send(ϱ1, ϱ2), x_ϱ2 = δ_send[ϱ1][·](x) in
  if x_ϱ2 (
    bad → halt
    | _→
      let _ = send[ϱ1][ϱ2][·][·]() in
      let ρ, secret′ = unpack secret in
      let ϱ3, δ_read(ϱ2, ϱ3, ρ), x_ϱ3 =
        δ_read[ϱ2][·](x_ϱ2, secret′) in
      if x_ϱ3 (
        bad → halt
        | _→
          let salary = read[ϱ2][ϱ3][·][·](secret′) in
          let taxes = salary in
          x_cont[ϱ3][·](x_ϱ3, taxes)))

where τ_cont =
  ∀[ϱ_cont, ≠ (ϱ_cont, bad)].
    (ϱ_cont, S(ϱ_cont), ∃ρ′:Val.int(ρ′)) → 0
```

Figure 9: Instrumenting the Taxation Applet

that the instantiation of predicate variables indicates the state transition that occurs. For example, execution of the expression $send[\varrho_1][\varrho_2][\cdot][\cdot]()$ causes the automaton to make a transition from ϱ_1 to ϱ_2.

5.2 Optimization

Many security automata exhibit special structure that allows us to optimize secure programs by eliminating checks that are inserted by the naive program instrumentation procedure [21]. One common case is an operation f that always succeeds in a given state q and transfers control to a new state q' regardless of its arguments. In this situation, we can make the following axiom available to the type checker:

$$\frac{}{\Phi \vdash \delta_f(q, q', P_1, \ldots, P_n)} \quad \text{(for all } P_1, \ldots, P_n\text{)}$$

If we know we are in state q, we can use the axiom above and the fact that $q \neq bad$ to satisfy the precondition on f; there is no need to perform a run-time check.

The *send* operation in the security automaton in Section 3 has this property. When invoked in the initial state, *send* always succeeds and execution continues in the initial state. Therefore, we can safely add the axiom:

$$\frac{}{\Phi \vdash \delta_{send}(start, start)}$$

Now, if we know our taxation function is only invoked in the *start* state, we can rewrite it, eliminating one of the run-time checks:

```
% Optimization 1:
fix _[](start, x_start:S(start), secret:∃ρ:Val.file(ρ),
        x_cont:τ_cont).
  let _ = send[start][start][·][·]() in
  ...
```

The type checker can prove *send* is executed in the *start* state and that the predicates $\delta_{send}(start, start)$ and $\neq (start, bad)$ are valid. Therefore, the optimized applet continues to type-check.

A second important way to optimize $\lambda_{\mathcal{A}}$ programs is to perform a control-flow analysis that propagates provable predicates statically through the program text. Using this technique, we can further optimize the taxation applet. Assume the calling context can prove the predicate $\delta_{read}(start, has\_read, \rho)$ (perhaps a run-time check was performed at some earlier time) where ρ is the value predicate corresponding to the file *secret*. In this case, the caller can invoke a tax applet with a stronger precondition that includes the predicate $\delta_{read}(start, has\_read, \rho)$. Moreover, with this additional information, an optimizer can eliminate the redundant check surrounding the file read operation:

```
% Optimization 2:
fix _[ρ:Val, δ_read(start, has_read, ρ)](start,
        secret:file(ρ),
        x_cont:∀[].(has_read, ∃ρ:Val.int(ρ)) → 0).
  let _ = send[start][start][·][·]() in
  let salary = read[start][has_read][·][·](secret) in
  let taxes = salary in
  x_cont(taxes)
```

In the code above, the optimizer rewrites the applet precondition with the necessary information. The caller is now obligated to prove the additional precondition before the applet can be invoked. The caller also unpacks the secret file before making the call so that the type checker can make the connection between the arguments to the δ_{read} predicate and this particular file. Finally, because the automaton state transitions are statically known throughout this program, we do not need to thread the state representation through the program. We assumed an optimizer was able to detect this unused argument and eliminate it. After performing all these optimizations, the resulting code is operationally equivalent to the original taxation applet from section 2, but provably secure.

The flexibility in the type system is particularly useful when a program repeatedly performs the same restricted operations. A more sophisticated tax applet might need to make a series of reads from the secret file (for charitable donations, number of dependents, etc.). If we assume the recursive function *read_a_lot* performs these additional reads, we need no additional security checks:

```
fix read_a_lot
    [ϱ:State, ρ:Val, δ_read(ϱ, has_read, ρ), ≠ (ϱ, bad)]
    (ϱ, secret:file(ρ),
    x_cont:∀[](has_read, ∃ρ:Val.int(ρ)) → 0).
  % In unknown state ϱ
  let info = read[ϱ][has_read][·][·]() in
  % In known state has_read
  ...
  % Must prove δ(has_read, has_read, ρ)
  read_a_lot[has_read][ρ][·][·](secret, x_cont)
```

The *read_a_lot* function can be invoked in a good state ϱ (*i.e.* either *start* or *has_read*) when we can prove $\delta_{read}(\varrho, has\_read, \rho)$. Using the δ_{read} predicate in the function precondition, the type checker infers that the read operation transfers control from the ϱ state to the *has_read* state. Before the recursive call, the type checker has the obligation to prove $\delta_{read}(has\_read, has\_read, \rho)$ but it cannot do so because it only knows that $\delta_{read}(\varrho, has\_read, \rho)$! Fortunately, we can remedy this problem by adding another policy-specific axiom to the type-checker:

$$\frac{\begin{array}{c}\Phi \vdash \neq (P, bad) \\ \Phi \vdash \delta_{read}(P, has\_read, P_f)\end{array}}{\Phi \vdash \delta_{read}(P', has\_read, P_f)} \quad \text{(for all } P, P', P_f\text{)}$$

This axiom states that if we can read a file P_f in one state (P), then we can read it in any state (except the *bad* one) and we always move to the *has_read* state. This condition is easily decidable.

In practice, Erlingsson and Schneider's untyped optimizer analyzes security automaton structure and performs optimizations similar to the ones discussed above [21]. Once the optimizer has obtained the information necessary for a particular transformation, this information can also be used to automatically generate the policy-specific axioms that we have discussed.

One significant optimization that is used in practice that we cannot encode in this typed framework is inlining. The run-time security checks, δ_f, are abstract constants in our framework because we have made a decision to trust the implementor of the security policy. When the policy writer implements the functions δ_f, he does not have to supply a formal proof that the functions imply some semantic property; the implementer merely asserts that some abstract predicate $\delta_f(q_1, q_2, \rho_1, \ldots, \rho_n)$ has been satisfied. If we inline δ_f into untrusted code, the corresponding assertion will also be inlined *into untrusted code*. In the current framework, there is no way to verify that these assertions are placed correctly by untrusted code. If performance is critical, the user will have to rely on a trusted just-in-time compiler to inline checks when they cannot be proven redundant statically.

6 Related and Future Work

The design of $\lambda_{\mathcal{A}}$ was inspired, in part, by Xi and Pfenning's Dependent ML (DML) [25, 24]. As in DML, we track the identity of values using dependent refinement types and singleton types. However, rather than applying the technology to array bounds check elimination and dead-code elimination, we have applied it to the problem of expressing security policies.

The secure language $\lambda_{\mathcal{A}}$ also bears resemblance to monadic systems [7, 17]. Both paradigms can be used to thread state through a computation. Normally, monadic types do not express the details of the state transformation. In contrast, the $\lambda_{\mathcal{A}}$ types describe the effect of the translation precisely: A function precondition specifies the input state and its continuation specifies the output state.

Leroy and Rouaix [6] also consider security in the context of strongly-typed languages. Their main concern is proving that standard strongly-typed languages provide certain security properties. For example, they show that a program written in a typed lambda calculus augmented with references cannot modify unreachable (in the sense of tracing garbage collection) locations. They also show that they can wrap dynamic checks around writes to sensitive locations and ensure "bad" values are not written to these locations. They did not investigate mechanisms for mechanically checking that their instrumented programs are safe, nor did they study the broader range of security properties that can be enforced using security automata.

The inability to inline security checks suggests an interesting direction for future research. The central problem is that the inlined code is trusted and has more priviledges than code written by an outsider. In particular, the inlined code has the priviledge to assert a state change in the security automaton. In order to inline we need a way to distinguish trusted and untrusted segments of code and to ensure that the trusted segments have not been tampered with. Recent work by Zdancewic *et al.* [26] describes how to distinguish between code segments with varying privileges. It may be possible to augment their system with some syntactic rules for reasoning about code equivalence. Using these techniques, inlining and then further optimization such as constant folding may be possible.

In order to test the practical consequences of our language design, we are interested in porting Erlingsson and Schneider's untyped SASI compiler [21] to the type-preserving Popcorn compiler [8] and using Typed Assembly Language as the secure target language. The current TAL implementation contains many of the typing constructs that we will need including polymorphic function types, existentials, and singleton types. We are also currently augmenting the system with a first-order predicate logic that interprets integer inequalities. We believe we will be able extend this framework with the additional constructs necessary to generate a secure, yet flexible and efficient Typed Assembly Language.

Acknowledgements

I greatly appreciate the time Úlfar Erlingsson and Fred Schneider have spent explaining their security automaton model and SASI system to me. Karl Crary, Úlfar Erlingsson, Neal Glew, Dan Grossman, and Steve Zdancewic have made extremely helpful suggestions on previous drafts of this work. I thank my advisor Greg Morrisett for the energy and enthusiasm he has dedicated to my education.

References

[1] Bowen Alpern and Fred Schneider. Recognizing safety and liveness. *Distributed Computing*, 2:117–126, 1987.

[2] Karl Crary, Stephanie Weirich, and Greg Morrisett. Intensional polymorphism in type-erasure semantics. In *ACM SIGPLAN International Conference on Functional Programming*, pages 301–312, Baltimore, September 1998.

[3] Robert Harper and Greg Morrisett. Compiling polymorphism using intensional type analysis. In *Twenty-Second ACM Symposium on Principles of Programming Languages*, pages 130–141, San Francisco, January 1995.

[4] John E. Hopcroft and Jeffrey D. Ullman. *Introduction to Automata Theory, Languages, and Computation*. Addison-Wesley, 1979.

[5] Dexter Kozen. Efficient code certification. Technical Report TR98-1661, Cornell University, January 1998.

[6] Xavier Leroy and Francois Rouaix. Security properties of typed applets. In *Twenty-Fifth ACM Symposium on Principles of Programming Languages*, pages 391–403, San Diego, January 1998.

[7] Eugenio Moggi. Notions of computation and monads. *Information and Computation*, 93:55–92, 1991.

[8] Greg Morrisett, Karl Crary, Neal Glew, Dan Grossman, Richard Samuels, Frederick Smith, David Walker, Stephanie Weirich, and Steve Zdancewic. TALx86: A realistic typed assembly language. Submitted to the 1999 ACM SIGPLAN Workshop on Compiler Support for System Software, 1999.

[9] Greg Morrisett, Karl Crary, Neal Glew, and David Walker. Stack-based typed assembly language. In *Second International Workshop on Types in Compilation*, pages 95–117, Kyoto, March 1998. Published in Xavier Leroy and Atsushi Ohori, editors, *Lecture Notes in Computer Science*, volume 1473, pages 28-52. Springer-Verlag, 1998.

[10] Greg Morrisett, Matthias Felleisen, and Robert Harper. Abstract models of memory management. In *ACM Conference on Functional Programming and Computer Architecture*, pages 66–77, La Jolla, June 1995.

[11] Greg Morrisett and Robert Harper. Semantics of memory management for polymorphic languages. In A.D. Gordon and A.M. Pitts, editors, *Higher Order Operational Techniques in Semantics*, Publications of the Newton Institute. Cambridge University Press, 1997.

[12] Greg Morrisett, David Walker, Karl Crary, and Neal Glew. From System F to Typed Assembly Language. In *Twenty-Fifth ACM Symposium on Principles of Programming Languages*, pages 85–97, San Diego, January 1998.

[13] George Necula. Proof-carrying code. In *Twenty-Fourth ACM Symposium on Principles of Programming Languages*, pages 106–119, Paris, 1997.

[14] George Necula and Peter Lee. Safe kernel extensions without run-time checking. In *Proceedings of Operating System Design and Implementation*, pages 229–243, Seattle, October 1996.

[15] George Necula and Peter Lee. The design and implementation of a certifying compiler. In *ACM SIGPLAN Conference on Programming Language Design and Implementation*, pages 333 – 344, Montreal, June 1998.

[16] George C. Necula and Peter Lee. Safe, untrusted agents using proof-carrying code. *LNCS 1419: Special Issue on Mobile Agent Security*, October 1997.

[17] Simon L. Peyton Jones and Philip Wadler. Imperative functional programming. In *Twentieth ACM Symposium on Principles of Programming Languages*, Charleston, South Carolina, January 1993.

[18] John C. Reynolds. Definitional interpreters for higher-order programming languages. In *Conference Record of the 25th National ACM Conference*, pages 717–740, Boston, August 1972.

[19] Fred Schneider. Enforceable security policies. Technical Report TR98-1664, Cornell University, January 1998.

[20] Chris Small. MiSFIT: A tool for constructing safe extensible C++ systems. In *Proceedings of the Third USENIX Conference on Object-Oriented Technologies*, Portland, OR, June 1997.

[21] Úlfar Erlingsson and Fred B. Schneider. SASI security. Unpublished manuscript produced at Cornell University, March 1998
See also http://sasi.cs.cornell.edu/javademo/.

[22] Robert Wahbe, Steven Lucco, Thomas Anderson, and Susan Graham. Efficient software-based fault isolation. In *Fourteenth ACM Symposium on Operating Systems Principles*, pages 203–216, Asheville, December 1993.

[23] Andrew K. Wright and Matthias Felleisen. A syntactic approach to type soundness. *Information and Computation*, 115(1):38–94, 1994.

[24] Hongwei Xi. *Dependent Types in Practical Programming*. PhD thesis, Carnegie Mellon University, 1999.

[25] Hongwei Xi and Frank Pfenning. Dependent types in practical programming. In *ACM Symposium on Principles of Programming LanguagesTwenty-Sixth*, pages 214–227, San Antonio, TX, January 1999.

[26] Steve Zdancewic, Dan Grossman, and Greg Morrisett. Principals in programming languages. Submitted to the Fourth ACM SIGPLAN International Conference on Functional Programming, 1999.

A Type Formation for $\lambda_{\mathcal{A}}$

$Dom(\Delta)$	$=$	The set of variables ρ such that $\rho{:}\kappa$ appears in Δ
Γ		A finite map of value vars to types
$\Gamma, x{:}\tau$		Update Γ so that $\Gamma(x) = \tau$
Φ	$::=$	$\Delta; \Gamma; P$
$(\Delta; \Gamma; P), \rho{:}\kappa$	$=$	$\Delta, \rho{:}\kappa; \Gamma; P$ if $\rho \notin \Delta$
$(\Delta; \Gamma; P), x{:}\tau$	$=$	$\Delta; \Gamma, x{:}\tau; P$ if $x \notin \Gamma$
$(\Delta; \Gamma; P), P'$	$=$	$\Delta, P'; \Gamma; P$
$(\Delta; \Gamma; P) \leftarrow P'$	$=$	$\Delta; \Gamma; P'$

Figure 10: Type Checking Contexts

$\boxed{\Phi \vdash P : \kappa}$

$$\frac{}{\Delta; \Gamma; P \vdash \rho : \kappa} \; (\Delta(\rho) = \kappa) \tag{16}$$

$$\frac{}{\Delta; \Gamma; P \vdash \iota : \kappa} \; (\mathcal{I}(\iota) = \kappa) \tag{17}$$

$$\frac{\begin{array}{c}\Phi \vdash \iota : (\kappa_1, \ldots, \kappa_n) \to \kappa \\ \Phi \vdash P_1 : \kappa_1 \quad \cdots \quad \Phi \vdash P_n : \kappa_n\end{array}}{\Phi \vdash \iota(P_1, \ldots, P_n) : \kappa} \tag{18}$$

$\boxed{\Phi \vdash \Delta}$

$$\frac{}{\Phi \vdash \cdot} \tag{19}$$

$$\frac{\Delta; \Gamma; P \vdash \Delta'}{\Delta; \Gamma; P \vdash \Delta', \rho{:}\kappa} \; (\rho \notin Dom(\Delta) \cup Dom(\Delta')) \tag{20}$$

$$\frac{\Phi \vdash \Delta \quad \Phi, \Delta \vdash P : \mathcal{B}}{\Phi \vdash \Delta, P} \tag{21}$$

$\boxed{\Phi \vdash \tau}$

$$\frac{\Phi \vdash P : \texttt{Val}}{\Phi \vdash b(P)} \; (b \neq \texttt{S}) \tag{22}$$

$$\frac{\Phi \vdash P : \texttt{State}}{\Phi \vdash \texttt{S}(P)} \tag{23}$$

$$\frac{\begin{array}{c}\Phi \vdash \Delta \qquad \Phi, \Delta \vdash P : \texttt{State} \\ \Phi, \Delta \vdash \tau_1 \qquad \cdots \qquad \Phi, \Delta \vdash \tau_n\end{array}}{\Phi \vdash \forall[\Delta].(P, \tau_1, \ldots, \tau_n) \to \texttt{0}} \tag{24}$$

$$\frac{\Phi, \rho{:}\kappa \vdash \tau}{\Phi \vdash \exists \rho{:}\kappa.\tau} \tag{25}$$

Figure 11: Contexts, Types, and Predicates

B Proof of Soundness

The following proof is a standard application of the syntactic proof techniques popularized by Wright and Felleisen [23]. The central lemmas are Subject Reduction and Progress, but before we can prove these results, we require a number of supporting lemmas, which are generally proven by induction on the structure of terms or on the typing derivations.

Inspection of the typing rules reveals that Δ is the only relevant part of the context Φ in the judgements:

1. $\Phi \vdash P : \kappa$
2. $\Phi \vdash P$
3. $\Phi \vdash \Delta$
4. $\Phi \vdash \tau$

Furthermore, Δ and Γ are not relevant in the judgement $\Phi \vdash P$ in_state and P is not relevant in the judgement $\Phi \vdash v : \tau$. When a portion of the context is not relevant, we will often write _ in its place.

It will be useful to define a couple of auxiliary judgements that state the well-formedness of contexts:

Definition 2 **($\Delta \vdash \Gamma$)** *If* $\Delta; \_; \_ \vdash \tau$ *for all* $x \in Dom(\Gamma)$ *such that* $\Gamma(x) = \tau$ *then* $\Delta \vdash \Gamma$.

Definition 3 **($\vdash \Phi$)** *If* $\vdash \Delta$ *and* $\Delta \vdash \Gamma$ *and* $\Delta; \_; \_ \vdash P : \texttt{State}$ *then* $\vdash \Delta; \Gamma; P$.

We will use the notation $\mathrm{FV}(X)$ to denote the free variables of an object X. In a predicate context $\Delta, \rho{:}\kappa, \Delta'$, the variable ρ is considered bound in Δ'.

Lemma 6 *If* $\vdash \Phi$ *and* $\Phi = \Delta; \Gamma; P$ *then*

1. *If* $\Phi \vdash \Delta'$ *then* $\mathrm{FV}(\Delta') \subseteq Dom(\Delta)$
2. *If* $\Delta \vdash \Gamma$ *then* $\mathrm{FV}(\Gamma) \subseteq Dom(\Delta)$
3. *If* $\Phi \vdash P : \kappa$ *then* $\mathrm{FV}(P) \subseteq Dom(\Delta)$
4. *If* $\Phi \vdash \tau$ *then* $\mathrm{FV}(\tau) \subseteq Dom(\Delta)$
5. *If* $\Phi \vdash v : \tau$ *then* $\mathrm{FV}(v) \subseteq Dom(\Delta) \cup Dom(\Gamma)$
6. *If* $\Phi \vdash e$ *then* $\mathrm{FV}(e) \subseteq Dom(\Delta) \cup Dom(\Gamma)$

Proof: By induction on the height of the respective typing derivations. □

Clearly, extending the context will not reduce the number of things that we can type. Hence, we can prove the following weakening lemma.

Lemma 7

1. *If* $\Phi \vdash \Delta$ *then* $\Phi, P \vdash \Delta$
2. *If* $\Delta \vdash \Gamma$ *then* $\Delta, P \vdash \Gamma$
3. *If* $\Phi \vdash P : \kappa$ *then* $\Phi, P \vdash P : \kappa$
4. *If* $\Phi \vdash \tau$ *then* $\Phi, P \vdash \tau$
5. *If* $\Phi \vdash P'$ *then* $\Phi, P \vdash P'$
6. *If* $\Phi \vdash P'$ in_state *then* $\Phi, P \vdash P'$ in_state
7. *If* $\Phi \vdash v : \tau$ *then* $\Phi, P \vdash v : \tau$
8. *If* $\Phi \vdash e$ *then* $\Phi, P \vdash e$

Proof: By induction on the height of the typing derivation. □

Subject Reduction relies upon the fact that validity is preserved by the provability relation. In other words, given a list of valid predicates, if you can prove another predicate using the syntactic proof rules, then it had better be valid. Following this lemma are number of substitution lemmas that are also critical to Subject Reduction.

Lemma 8 *Given predicates* $P_1, \ldots, P_n$, *if for* $1 \leq i \leq n$, $\mathcal{A} \models P_i$ *and* $P_1, \ldots, P_n; \_; \_ \vdash P$ *then* $\mathcal{A} \vdash P$.

Proof: Trivial by inspection of the proof rules. $\square$

Lemma 9 (Type Substitution I) *If* $\cdot; \_; \_ \vdash \Delta, \rho{:}\kappa, \Delta'$ *and* $\Delta, \rho{:}\kappa, \Delta'; \_; \_ \vdash P : \kappa$ *then*

1. *If* $\Delta, \rho{:}\kappa, \Delta'; \_; \_ \vdash \tau$ *then* $\Delta, (\Delta'[P/\rho]); \_; \_ \vdash \tau[P/\rho]$
2. *If* $\Delta, \rho{:}\kappa, \Delta'; \_; \_ \vdash P''$ *then* $\Delta, (\Delta'[P/\rho]); \_; \_ \vdash P''[P/\rho]$
3. *If* $\Delta, \rho{:}\kappa, \Delta'; \_; \_ \vdash \Delta$ *then* $\Delta, (\Delta'[P/\rho]); \_; \_ \vdash \tau[P/\rho]$
4. *If* $\Delta, \rho{:}\kappa, \Delta'; \_; \_ \vdash P'$ *then* $\Delta, (\Delta'[P/\rho]); \_; \_ \vdash P'[P/\rho]$
5. *If* $\Delta, \rho{:}\kappa, \Delta'; \_; P'' \vdash P'$ in_state *then* $\Delta, (\Delta'[P/\rho]); \_; P''[P/\rho] \vdash P'[P/\rho]$ in_state

Proof:

The proof proceeds by induction on the height of the typing derivation. Most cases follow straightforwardly from the induction hypothesis. Rule (16) for predicate variables uses the assumption $\Delta, \rho{:}\kappa, \Delta'; \_; \_ \vdash P : \kappa$.

$\square$

Lemma 10 (Type Substitution II) *Let* Φ *be* $\Delta, \rho{:}\kappa, \Delta'; \Gamma; P$ *and let* Φ' *be* $\Delta, (\Delta'[P'/\rho]); \Gamma[P'/\rho]; P[P'/\rho]$. *If* $\Phi \vdash P' : \kappa$ *then*

1. *if* $\Phi \vdash v : \tau$ *then* $\Phi' \vdash v[P'/\rho] : \tau[P'/\rho]$
2. *if* $\Phi \vdash e$ *then* $\Phi' \vdash e[P'/\rho]$

Proof:

The proof is by induction on the height of the typing derivation. Rule (5) follows because τ is closed. Rules (6) through (12) follow by Type Substitution I and the induction hypothesis. Rules (14) and (15) follow by the induction hypothesis. Rule (13) is slightly more interesting. The precondition for the rule includes the judgement $\Phi \vdash v : \mathtt{S}(\rho')$. If $\rho' \neq \rho$ then the result follows by induction so assume that $\rho' = \rho$. In this case, the judgement has the form:

$$\frac{\begin{array}{cc}\Phi \vdash v : \mathtt{S}(\rho) & \Phi \vdash q : \mathtt{S}(\hat{q}) \\ \Phi' \vdash e_1[\hat{q}/\rho] & \Phi, \neq (\rho, \hat{q}) \vdash e_2\end{array}}{\Phi \vdash \mathtt{if}\, v\,(q \rightarrow e_1 \mid \_ \rightarrow e_2)}$$

There are three cases to consider:

1. $P' = \rho''$ for some other predicate variable ρ''. In this case, the result also follows by a straightforward application of the induction hypothesis.
2. $P' = \hat{q}$. In this case, we have $\Phi' \vdash v[\hat{q}/\rho] : \mathtt{S}(\hat{q})$ and $\Phi' \vdash q : \mathtt{S}(\hat{q})$ by induction. Using this fact and the judgement, $\Phi' \vdash e_1[\hat{q}/\rho]$ from the typing judgement above, we can use rule (14) to conclude:

   $$\Phi \vdash \mathtt{if}\, v[\hat{q}/\rho](q \rightarrow e_1[\hat{q}/\rho] \mid \_ \rightarrow e_2[\hat{q}/\rho])$$

   and therefore that:

   $$\Phi \vdash \mathtt{if}\, v(q \rightarrow e_1 \mid \_ \rightarrow e_2)[\hat{q}/\rho]$$

3. $P' \neq \hat{q}$ and $P' \neq \rho''$ for any other predicate variable ρ''. In this case, we have $\Phi' \vdash v[P'/\rho] : \mathtt{S}(P')$ and $\Phi' \vdash q : \mathtt{S}(\hat{q})$ and $\Phi', \neq (\rho, \hat{q})[P'/\rho] \vdash e_2[\hat{q}/\rho]$ by induction. Now, we can use rule (15) to conclude:

   $$\Phi \vdash \mathtt{if}\, v[P'/\rho](q \rightarrow e_1[P'/\rho] \mid \_ \rightarrow e_2[P'/\rho])$$

   and therefore that:

   $$\Phi \vdash \mathtt{if}\, v(q \rightarrow e_1 \mid \_ \rightarrow e_2)[P'/\rho]$$

$\square$

For the Value Substitution lemma below, we will use some additional notation. Let $\Gamma \backslash x$ be the restriction of map Γ to domain $Dom(\Gamma) - \{x\}$.

Lemma 11 (Value Substitution) *If* $\Gamma(x) = \tau$ *and* $\vdash \Delta; \Gamma; P$ *and* $\Delta; \Gamma \backslash x; P \vdash v : \tau$ *then*

1. *If* $\Delta; \Gamma; P \vdash v' : \tau'$ *then* $\Delta; \Gamma \backslash x; P \vdash v'[v/x] : \tau'$
2. *If* $\Delta; \Gamma; P \vdash e$ *then* $\Delta; \Gamma \backslash x; P \vdash e[v/x]$

Proof:

The proof follows by induction on the height of the typing derivations for values and expressions.

□

Lemma 12 *If $\Phi \vdash P : \kappa$ and $\rho \notin \Phi$ then*

1. *If $\Phi \vdash P'[P/\rho] : \kappa'$ then $\Phi, \rho{:}\kappa \vdash P' : \kappa'$*
2. *If $\Phi \vdash \Delta[P/\rho]$ then $\Phi, \rho{:}\kappa \vdash \Delta[P/\rho]$*
3. *If $\Phi \vdash \tau[P/\rho]$ then $\Phi, \rho{:}\kappa \vdash \tau[P/\rho]$*

Proof: By induction on the structure of P', Δ, and τ. □

Before we can tackle the Progress and Subject Reduction lemmas, we need two additional lemmas. The first lemma states that well-formed values are always given well-formed types. The second is the Canonical Forms lemma. Intuitively, Canonical Forms captures the invariants that the type system ensures for all values of each type τ. One of the most useful invariants is some indication of the shape of the value. For example, Canonical Forms states that a value with existential type must have the form $\texttt{pack}[P, v']$ *as* $\exists\rho{:}\texttt{Val}.\tau'$.

Lemma 13 *If $\vdash \Phi$ and $\Phi \vdash v : \tau$ then $\Phi \vdash \tau$.*

Proof:

The proof is by induction on the typing derivation for values. Rule (4) follows from the definition $\vdash \Phi$ and rule (5) follows from the fact that the codomain of $\mathcal{C}$ only contains well-formed, closed types. Rule (6) is immediate. Rule (7) follows from the induction hypothesis and Type Substitution I. Rule (8) follows by induction. Rule (9) follows from the induction hypothesis and Lemma 12.

□

Lemma 14 (Canonical Forms) *If $\cdot;\cdot;\_ \vdash v : \tau$ then*

1. *If $\tau = b(\hat{a})$ then $v = a$*
2. *If $\tau = \texttt{S}(P)$ then $P = \hat{q}$ and $v = q$ for some state q*
3. *If $\tau = \forall[\,].(P, \tau_1, \ldots, \tau_n) \to \texttt{0}$ then either:*

   (a) $v = v'\phi_1 \cdots \phi_m$ *where* $v' = \texttt{fix}\, g[\theta_1, \ldots, \theta_m].(P', x_1{:}\tau_1', \ldots, x_n{:}\tau_n').e_0$
   Moreover, let S_0 be the empty substitution and let SS' denote the composition of substitutions S and S'. Now, for $1 \leq i \leq m$,
   $\phi_i = [\cdot]$ *and* $\theta_i = P_i'$ *and* $\Phi \vdash P_i'$ *and* $S_i = S_{i-1}$ *or*
   $\phi_i = [P_i']$ *and* $\theta_i = \rho_i{:}\kappa_i$ *and* $\Phi \vdash P_i : \kappa_i$ *and* $S_i = S_{i-1}[P_i'/\rho_i]$
   Furthermore:

   - $P = S_m(P')$
   - $\tau_1 = S_m(\tau_1')$

     $\vdots$
   - $\tau_m = S_m(\tau_m')$

   and finally, $\Phi \vdash v' : \forall[\theta_1, \ldots, \theta_m].(P', \tau_1', \ldots, \tau_n') \to \texttt{0}$

   OR:

   (b) *for some* $q_1, a_2, \ldots, a_{n-1}$, $v = \delta_f[\hat{q_1}][\hat{a_2}]\cdots[\hat{a_{n-1}}][\cdot]$, *and*

   - $P = \hat{q_1}$
   - $\tau_1 = \texttt{S}(\hat{q_1})$
   - $\tau_2 = b_2(\hat{a_2})$

     $\vdots$
   - $\tau_{n-1} = b_{n-1}(\hat{a_{n-1}})$
   - $\tau_n = \forall[\varrho_2{:}\texttt{State}, \delta_f(\hat{q_1}, \varrho_2, a_2, \ldots, a_{n-1})](\hat{q_1}, \texttt{S}(\varrho_2)) \to \texttt{0}$

   where for $2 \leq i \leq n-1$, $b_i = \mathcal{C}_{insecure}(a_i)$
   and $\mathcal{C}_{insecure}(f) = (b_1, \ldots, b_n, (b) \to \texttt{0}) \to \texttt{0}$ *for some base type* b

   OR:

(c) *for some* $q_1, q_2, a_1, \ldots, a_{n-1}$, $v = f[\hat{q_1}][\hat{q_2}][\hat{a_1}] \cdots [\hat{a_n}][\cdot][\cdot]$ *and*

- $\Phi \vdash \neq (\hat{q_2}, bad)$
- $\Phi \vdash \delta_f(\hat{q_1}, \hat{q_2}, \hat{a_1}, \ldots, \hat{a_n})$
- $\tau_1 = b_1(\hat{a_1})$

  $\vdots$
- $\tau_{n-1} = b_{n-1}(\widehat{a_{n-1}})$
- $\tau_n = \forall[\,].(\hat{q_2}, \exists\rho{:}\mathtt{Val}.b(\rho)) \rightarrow \mathtt{0}$

where for $1 \leq i \leq n-1$, $b_i = \mathcal{C}_{insecure}(a_i)$
and $\mathcal{C}_{insecure}(f) = (b_1, \ldots, b_n, (b) \rightarrow \mathtt{0}) \rightarrow \mathtt{0}$

4. *If* $\tau = \exists\rho{:}\mathtt{Val}.\tau'$ *then* $v = \mathtt{pack}[P, v']$ *as* $\exists\rho{:}\mathtt{Val}.\tau'$

Proof:

For parts 1 and 2, the proof follows by inspection of the typing rules and the value signature $\mathcal{C}$. Part 3 (a) follows by an induction on the derivation $\cdot;\cdot;\_ \vdash v : \forall[\Delta].(P, \tau_1, \ldots, \tau_n) \rightarrow \mathtt{0}$ and inspection of static semantics rules (7) and (8). Parts 3 (b) and 3 (c) follow by a similar induction and inspection of the signature $\mathcal{C}$. Part 4 follows by inspection of the typing rules.

□

Lemma 15 (Progress) *If* $\mathcal{A}; q \vdash e$ *then either:*

1. $e \longmapsto_s e'$ *or*
2. $e = \mathtt{halt}$

Proof:

The assumption $\mathcal{A}; q \vdash e$ implies there exist valid predicates $P_1, \ldots, P_n$ such that $\Phi \vdash e$ where Φ is the context $P_1, \ldots, P_n; \cdot; \hat{q}$. Let TD be the typing derivation for this judgement. The proof proceeds by cases on the syntax of the expression e:

case: $v_0(v_1, \ldots, v_n)$ The typing derivation TD has the form:

$$\frac{\Phi \vdash v_0 : \forall[\,].(P, \tau_1, \ldots, \tau_n) \rightarrow \mathtt{0} \quad \Phi \vdash v_1 : \tau_1 \quad \cdots \quad \Phi \vdash v_n : \tau_n \quad \Phi \vdash P \text{ in\_state}}{\Phi \vdash v_0(v_1, \ldots, v_n)}$$

By Canonical Forms, part 3, v_0 has one of three forms:

1. $v_0 = v'\phi_1 \cdots \phi_m$ and $v' = \mathtt{fix}\, g[\theta_1, \ldots, \theta_m].(P', x_1{:}\tau_1', \ldots, x_n{:}\tau_n').e_0$. Canonical Forms 3 (a) satisfies the requirements on the operational rule for $\mathtt{fix}$ immediately and consequently:
   $e \longmapsto. (S_m(e))[v', v_1, \ldots, v_n/g, x_1, \ldots, x_n]$ where S_m is the substitution by the same name defined in Canonical Forms 3 (a).
2. $v = \delta_f[\hat{q}_1][\hat{a}_2] \cdots [\hat{a}_{n-1}][\cdot]$ for some $q_1, a_1, \ldots, a_{n-1}$
   By Canonical Forms, part 3 (b),
   - $\tau_1 = \mathtt{S}(\hat{q}_1)$ and
   - $\tau_2 = b_2(\hat{a}_2)$ and

     $\vdots$
   - $\tau_{n-1} = b_{n-1}(\hat{a}_{n-1})$

   where for $1 \leq i \leq n$, $b_i = \mathcal{C}_{insecure}(a_i) = b_i$
   and $\mathcal{C}_{insecure}(f) = (b_1, \ldots, b_n, (b) \rightarrow \mathtt{0}) \rightarrow \mathtt{0}$ for some base type b
   By Canonical Forms, parts 1 and 2, $v_1 = q_1$ and for $2 \leq i \leq n-1$, $v_i = a_i$. Because δ is total, there exists some q_2 such that $\delta(q_1)(a_2, \ldots, a_{n-1}) = q_2$. Thus, $e \longmapsto. v_n[\hat{q}_2][\cdot](q_2)$
3. $v = f[\hat{q_1}][\hat{q_2}][\hat{a_1}] \cdots [\hat{a_n}][\cdot][\cdot]$ for some $q_1, q_2, a_1, \ldots, a_{n-1}$
   By Canonical Forms, part 3 (c):
   - $\tau_1 = b_1(\hat{a_1})$

     $\vdots$
   - $\tau_{n-1} = b_{n-1}(\hat{a}_{n-1})$
   - $\tau_n = \forall[\,].(\hat{q_2}, \exists\rho{:}\mathtt{Val}.b(\rho)) \rightarrow \mathtt{0}$

where for $1 \le i \le n-1$, $b_i = \mathcal{C}_{insecure}(a_i)$
and $\mathcal{C}_{insecure}(f) = (b_1, \ldots, b_n, (b) \to \mathtt{0}) \to \mathtt{0}$
By Canonical Forms, part 1, for $1 \le i \le n-1$, $v_i = a_i$
Thus, $e \longmapsto_{f(a_1,\ldots,a_n)} v_n(\mathtt{pack}[\hat{a}, a]\ as\ \exists\rho{:}\mathtt{Val}.b(\rho))$ for some a such that $\mathcal{C}_{insecure}(a) = b$

case: `halt` Trivial.

case: `unpack` The expression e has the form $\mathtt{let}\,\rho, x = \mathtt{unpack}\,v\,\mathtt{in}\,e''$. The typing derivation TD has the form:

$$\frac{\Phi \vdash v : \exists\rho{:}\mathtt{Val}.\tau \quad \Phi, \rho{:}\mathtt{Val}, x{:}\tau \vdash e}{\Phi \vdash \mathtt{let}\,\rho, x = \mathtt{unpack}\,v\,\mathtt{in}\,e}$$

By Canonical Forms, part 4, v has the form $\mathtt{pack}[P, v']\ as\ \exists\rho{:}\mathtt{Val}.\tau$. Consequently, $e \longmapsto_s e''[P, v'/\rho, x]$.

case: $\mathtt{if}\,v$ The expression e has the form $\mathtt{if}\,v\,(q \to e_1 \mid \_ \to e_2)$. There are three cases corresponding to static semantics rules (13), (14), and (15). By Lemmas 13 and 6, the type of the value v cannot contain a free variable. Thus, we can rule out the possibility of the last rule in the derivation TD being (13). The typing derivation TD for the other two rules both contain a precondition of the form $\Phi \vdash v : \mathtt{S}(P)$. By Canonical Forms, part 2, $P = \hat{q}'$ and $v = q'$ for some state q'. If $q' = q$ then $e \longmapsto_\cdot e_1$ and otherwise $e \longmapsto_\cdot e_2$.

□

Lemma 16 (Subject Reduction) *If $\mathcal{A}; q \vdash e$ and $e \longmapsto_s e'$ then*

1. *if $s = \cdot$ then $\mathcal{A}; q \vdash e'$*
2. *and if $s = f(a_1, \ldots, a_n)$ then $\mathcal{A}; q' \vdash e'$ where $\delta(f)(q, a_1, \ldots, a_n) = q'$*

Proof:

The assumption $\mathcal{A}; q \vdash e$ implies

1. $q \neq bad$

and there exists predicates $P_1, \ldots, P_k$ such that:

2. $\mathcal{A} \models P_i$, for $1 \le i \le k$
3. $\Phi \vdash e$ where $\Phi = P_1, \ldots, P_k; \cdot; \hat{q}$

Let TD be the typing derivation proving item 3. We now proceed by cases on the operational semantics $e \longmapsto_s e'$:

case: `fix` The expression e has the form $v'\phi_1 \cdots \phi_m(v_1, \ldots, v_n)$
where $v' = \mathtt{fix}\,g[\theta_1, \ldots, \theta_m].(P, x_1{:}\tau_1, \ldots, x_n{:}\tau_n).e_0$
and for $1 \le i \le m$, when S_0 is the empty substitution,
$\phi_i = [\cdot]$ and $\theta_i = P_i'$ and $S_i = S_{i-1}$ or
$\phi_i = [P_i']$ and $\theta_i = \rho_i{:}\kappa_i$ and $S_i = S_{i-1}[P_i'/\rho_i]$
The expression e' has the form $S_m(e_0)[v', v_1, \ldots, v_n/g, x_1, \ldots, x_n]$
and $e \longmapsto_\cdot e'$
The typing derivation TD for e is:

$$\frac{\Phi \vdash v'\phi_1 \cdots \phi_m : \forall[\,].(P', \tau_1', \ldots, \tau_n') \to \mathtt{0} \qquad \begin{array}{c}\Phi \vdash v_1 : \tau_1' \\ \vdots \\ \Phi \vdash v_n : \tau_n'\end{array} \qquad \Phi \vdash P'\ \mathrm{in\_state}}{\Phi \vdash v'\phi_1 \cdots \phi_m(v_1, \ldots, v_n)}$$

By Canonical Forms, part 3 (a), $\Phi \vdash v' : \tau_g$ where $\tau_g = \forall[\theta_1, \ldots, \theta_m].(P, \tau_1, \ldots, \tau_n) \to \mathtt{0}$.
Inspection of the typing rules, indicates that only the rule 6 could have applied. The precondition for this rule states that: $(\Phi, \theta_1, \ldots, \theta_m, g{:}\tau_g, x_1{:}\tau_1, \ldots, x_n{:}\tau_n) \leftarrow P \vdash e_0$.
By Canonical Forms, part 3 (a), we also know that $1 \le i \le m$, if $\phi_i = [P_i']$ then $\Phi \vdash P_i' : \kappa_i$. Consequently, we can use the Type Substitution II and Value Substitution lemmas, to show that:

$$(\Phi, \theta_1', \ldots, \theta_m') \leftarrow S_m(P) \vdash S_m(e_0)[v', v_1, \ldots, v_n/g, x_1, \ldots, x_n]$$

where for $1 \le i \le m$,

- if $\theta_i = \rho{:}\kappa$ then $\theta_i = \cdot$ ($i.e.$ does not appear in the list)
- if $\theta_i = P$ then $\theta_i' = S_i(\theta_i)$

This fact establishes part 3 of the proof obligation $\mathcal{A}; q \vdash e'$.
Now we must establish the validity of the predicates $\theta'_1, \ldots, \theta'_m$. By Canonical Forms, part 3 (a), $\Phi \vdash \theta_i$ for $1 \leq i \leq m$ when θ_i is of the form P'_i. By Type Substitution I, we can conclude that $\Phi \vdash \theta'_i$. By assumption, for $1 \leq j \leq k$, each predicate P_j in the initial context Φ is valid. Therefore, by Lemma 8, since $\Phi \vdash \theta'_i$, we know that each θ'_i is valid. This fact establishes part 2 of the proof obligation.
Finally, $\Phi \vdash P'$ in_state. By inspection of the judgement in_state and the fact that $\Phi = P_1, \ldots, P_n; \cdot; q$ and $q \neq bad$, we have $P' = q \neq bad$. This fact establishes part 1. Therefore, $\mathcal{A}; q \vdash e'$. Since $s = \cdot$, we are done.

case: δ_f The expression e has the form $\delta_f[\hat{q_1}][\hat{a_1}] \cdots [\hat{a_n}][\cdot](q_1, a_1, \ldots, a_n, v_{cont})$ and e' has the form $v_{cont}[\hat{q_2}][\cdot](q_2)$. By inspection of the operational semantics, we also know that $\delta(f)(q_1, a_1, \ldots, a_n) = q_2$. Using the definition of validity, we have:

4. $\mathcal{A} \models \delta_f(q_1, q_2, a_1, \ldots, a_n)$

The typing derivation TD is the following where the context Φ is $P_1, \ldots, P_n; \cdot; \hat{q}$:

$$\frac{\Phi \vdash \delta_f[\hat{q_1}][\hat{a_1}] \cdots [\hat{a_n}][\cdot] : \forall[\,].(P', \tau_0, \tau_1, \ldots, \tau_n, \tau_{n+1}) \to \mathtt{0} \qquad \begin{array}{c} \Phi \vdash q_1 : \tau_0 \\ \Phi \vdash a_1 : \tau_1 \\ \vdots \\ \Phi \vdash a_n : \tau_n \\ \Phi \vdash v_{cont} : \tau_{n+1} \end{array} \qquad \Phi \vdash P' \text{ in\_state}}{\Phi \vdash \delta_f[\hat{q_1}][\hat{a_1}] \cdots [\hat{a_n}][\cdot](q_1, a_1, \ldots, a_n, v_{cont})}$$

From Canonical Forms, we have:

5. $\tau_0 = \mathtt{S}(\hat{q_1})$
6. for $1 \leq i \leq n$, $\tau_i = b_i(\hat{a_i})$
7. $\tau_{n+1} = \forall[\varrho_2{:}\mathtt{State}, \delta_f(\hat{q_1}, \varrho_2, \hat{a_1}, \ldots, \hat{a_n})](\hat{q_1}, \mathtt{S}(\varrho_2)) \to \mathtt{0}$
8. $P' = \hat{q_1}$

By inspection of the rule for $\Phi \vdash P'$ in_state, we also know that (9.) $\hat{q} = P' = \hat{q_1}$.
Now, we can show:

10. $\hat{q} \neq bad$ (which we know by 1.)
11. Predicates $P_1, \ldots, P_n, \delta_f(q_1, q_2, a_1, \ldots, a_n)$ are valid. (which we know by 2. and 4.)
12. $\Phi' \vdash v_{cont}[\hat{q_2}][\cdot](q_2)$ where $\Phi' = P_1, \ldots, P_n, \delta(f)(q_1, q_2, a_1, \ldots, a_n); \cdot; \hat{q}$ using the following derivation:

$$\frac{\dfrac{(13a) \quad (13b)}{\Phi' \vdash v_{cont}[\hat{q_2}][\cdot] : \forall[\,].(\hat{q_1}, \mathtt{S}(\hat{q_2})) \to \mathtt{0}}\,(13) \qquad \dfrac{}{\Phi' \vdash q_2 : \mathtt{S}(\hat{q_2})}\,(14) \qquad \dfrac{}{\Phi' \vdash \hat{q_1} \text{ in\_state}}\,(15)}{\Phi' \vdash v_{cont}[\hat{q_2}][\cdot](q_2)}$$

We can show judgement 14 using static semantics rule (5) and the definition of the signature $\mathcal{C}$. We can show judgement 15 using rule (3) because by 9, $\hat{q_1} = \hat{q}$. We can show 13 using the static semantics rule (8) and the sub-derivations 13a:

$$\frac{\dfrac{}{\Phi' \vdash v_{cont} : \forall[\varrho_2{:}\mathtt{State}, P_{\delta_f}].(\hat{q_1}, \mathtt{S}(\hat{q_2})) \to \mathtt{0}}\,(16) \qquad \dfrac{}{\Phi' \vdash \hat{q_2} : \mathtt{State}}\,(17)}{\Phi' \vdash v_{cont}[\hat{q_2}] : \forall[P_{\delta_f}[\hat{q_2}/\varrho_2]].(\hat{q_1}, \mathtt{S}(\hat{q_2})) \to \mathtt{0}}\,(18)$$

and 13b:

$$\frac{}{\Phi' \vdash P_{\delta_f}[\hat{q_2}/\varrho_2]}$$

where $P_{\delta_f} = \delta_f(q_1, \varrho_2, a_1, \ldots, a_n)$
By Lemma 7 and TD, we can show $\Phi' \vdash v_{cont} : \tau_{n+1}$ and therefore, we have 16. Judgement 17 follows by inspection of the signature $\mathcal{I}$ and static semantics rule (17). Consequently, 18 follows using static semantics rule (7) and deduction 7 above. Therefore, we have 13a. 13b follows by rule (1) because $\delta_f(q_1, q_2, a_1, \ldots, a_n)$ appears in Φ'.

case: f The expression e has the form $v(a_1, \ldots, a_n, v_{cont})$ where $v = f[\hat{q_1}][\hat{q_2}][\hat{a_1}] \cdots [\hat{a_n}][\cdot][\cdot]$.
By inspection of the operational semantics, we have: $e \longmapsto_{f(a_1, \ldots, a_n)} e'$
where e' has the form $v_{cont}(\mathtt{pack}[\hat{a}, a]\ as\ \exists\rho{:}\mathtt{Val}.b(\rho))$ for some a such that $\mathcal{C}_{insecure}(a) = b$. Moreover:

– for $1 \leq i \leq n, \mathcal{C}_{insecure}(a_i) = b_i$
– and $\mathcal{C}_{insecure}(f) = (b_1, \ldots, b_n, (b) \to \mathtt{0}) \to \mathtt{0}$

The typing derivation TD has the form:

$$\frac{\Phi \vdash : \forall[\,].(P, \tau_1, \ldots, \tau_n, \tau_{cont}) \to \mathtt{0} \qquad \begin{array}{c}\Phi \vdash a_1 : \tau_1 \\ \vdots \\ \Phi \vdash a_n : \tau_n\end{array} \qquad \Phi \vdash v_{cont} : \tau_{cont} \qquad \Phi \vdash P \text{ in\_state}}{\Phi \vdash v(a_1, \ldots, a_n, v_{cont})}$$

By Canonical Forms, $\tau_{cont} = \forall[\,].(\hat{q}_2, \exists\rho{:}\mathtt{Val}.b(\rho)) \to \mathtt{0}$
We can now use static semantics rule (10) and the following derivation where $\Phi' = \Phi \leftarrow q_2$ to deduce that the epression e' is well-formed:

$$\frac{\dfrac{}{\Phi' \vdash v_{cont} : \tau_{cont}}\ (4) \qquad \dfrac{\dfrac{}{\Phi' \vdash a : \mathtt{S}(\hat{a})}\ (7)}{\Phi' \vdash \mathtt{pack}[\hat{a}, a]\ as\ \exists\rho{:}\mathtt{Val}.b(\rho)}\ (5) \qquad \dfrac{}{\Phi' \vdash q_2 \text{ in\_state}}\ (6)}{\Phi' \vdash v_{cont}(\mathtt{pack}[\hat{a}, a]\ as\ \exists\rho{:}\mathtt{Val}.b(\rho))}$$

Judgement 4 follows from $\Phi \vdash v_{cont} : \tau_{cont}$ and the fact that the state portion of the ctxt is irrelevant to this judgement. Judgement 7 follows using rule (5) and the fact that $\mathcal{C}(a) = \hat{a}$. Judgement 5 follows from 7 using static semantic rule (9) for existentials. Judgement 6 follows by rule (3).
By TD and Canonical Forms, part 3 (c), we have $\Phi \vdash \neq (q_2, bad)$ and $\Phi \vdash \delta_f(q_1, q_2, a_1, \ldots, a_n)$. Since, by assumption 2, $P_1, \ldots, P_k$ are valid, we know that $\neq (q_2, bad)$ and $\delta_f(q_1, q_2, a_1, \ldots, a_n)$ are also valid using Lemma 8. By the definition of validity, $q_2 \neq bad$ and therefore $\mathcal{A}; q_2 \vdash e'$. Moreover, $\delta(f)(q_1, a_1, \ldots, a_n) = q_2$. Therefore, we have our result.

case: `unpack` The expression e has the form $\mathtt{let}\, \rho, x = \mathtt{unpack}(\mathtt{pack}[P', v']\ as\ \tau)\, \mathtt{in}\, e''$ where τ has the form $\exists\rho{:}\mathtt{Val}.\tau'$. Also, $e \longmapsto_{\cdot} e'$ where e' has the form $e''[P', v'/\rho, x]$ The typing derivation TD has the form:

$$\frac{\dfrac{\Phi \vdash P' : \mathtt{Val} \quad \Phi \vdash v' : \tau'[P'/\rho]}{\Phi \vdash \mathtt{pack}[P', v']\ as\ \tau : \tau} \qquad \Phi, \rho{:}\mathtt{Val}, x{:}\tau' \vdash e''}{\Phi \vdash \mathtt{let}\, \rho, x = \mathtt{unpack}(\mathtt{pack}[P', v']\ as\ \tau)\, \mathtt{in}\, e''}$$

Using Type Substitution II and Value Substitution, we have $\Phi \vdash e''[P', v'/\rho, x]$. By assumptions 1 and 2, the predicates $P_1, \ldots, P_n$ in Φ are valid and the context state P is not bad. Consequently, $\mathcal{A}; q \vdash e''[P', v'/\rho, x]$ and we are done.

case: `if` The expression e has the form $\mathtt{if}\, v\, (q' \to e_1 \mid \_ \to e_2)$. There are two cases:

– Assume $v = q'$. In this case, $e \longmapsto_{\cdot} e_1$. Also, $\mathcal{C}(q') = \hat{q'}$. By inspection of the static semantics rules, we can deduce that the typing derivation TD must end in the rule (14):

$$\frac{\Phi \vdash v : \mathtt{S}(\hat{q'}) \quad \Phi \vdash q' : \mathtt{S}(\hat{q'}) \quad \Phi \vdash e_1}{\Phi \vdash \mathtt{if}\, v\, (q' \to e_1 \mid \_ \to e_2)}$$

By assumptions 1 and 2, the predicates $P_1, \ldots, P_n$ in the context Φ are valid and the current state q is not bad. Consequently, $\mathcal{A}; q \vdash e_1$. Therefore, we have our result.

– Assume $v \neq q'$ but is equal to some other state q''. In this case, $e \longmapsto_{\cdot} e_2$. Also, $\mathcal{C}(q') = \hat{q'}$. By inspection of the static semantics rules, we can deduce that the typing derivation TD must end in the rule (15):

$$\frac{\Phi \vdash v : \mathtt{S}(\hat{q''}) \quad \Phi \vdash q' : \mathtt{S}(\hat{q'}) \quad \Phi, \neq (\hat{q''}, \hat{q'}) \vdash e_2}{\Phi \vdash \mathtt{if}\, v\, (q' \to e_1 \mid \_ \to e_2)} \left(\begin{array}{c} q'' \neq \rho \\ q'' \neq q' \end{array}\right)$$

By assumptions 1 and 2, the predicates $P_1, \ldots, P_n$ in the context Φ are valid and the current state q is not bad. By the definition of validity, $\neq (q'', q')$ is valid since $q'' \neq q'$. Consequently, $\mathcal{A}; q \vdash e_2$ and we have our result.

□

Now that we have Subject Reduction and Progress, we can proceed to show the main result, Soundness. First, we need to define what it means for a program to be *stuck*:

Definition 4 (Stuck) *An expression e is* stuck *if e is not* `halt` *and there does not exist an expression e' such that $e \longmapsto_s e'$.*

Theorem 17 (Soundness)
If $\mathcal{A}; q_0 \vdash e_1$ *then*

1. (Type Soundness) *For all evaluation sequences,* $e_1 \longmapsto_{s_1} e_2 \longmapsto_{s_2} \cdots \longmapsto_{s_n} e_{n+1}$*, the expression* e_{n+1} *is not* stuck.
2. (Security) *If* $e_1 \longmapsto_{s_1} e_2 \longmapsto_{s_2} \cdots \longmapsto_{s_n} e_{n+1}$ *then* $|s_1, s_2, \ldots, s_n| \in \mathcal{L}(\mathcal{A})$

Proof:

The proof of part 1 proceeds by induction on the length of the evaluation sequence:

$n = 1$ By Progress, e_1 is not stuck.

$n > 1$ By Subject Reduction, $\mathcal{A}; q' \vdash e_2$ for some q'. By induction e_n is not stuck.

The proof of part 2 also proceeds by induction on the length of the evaluation sequence. The induction hypothesis is:

If $\mathcal{A}; q \vdash e_1$ and $e_1 \longmapsto_{s_1} e_2 \longmapsto_{s_2} \cdots \longmapsto_{s_n} e_{n+1}$ then $Accept(q, |s_1 s_2 \cdots s_n|)$

$n = 1$ From $\mathcal{A}; q \vdash e_1$, we know $q \neq bad$. The sequence of symbols emitted by the operational semantics is empty. Therefore, by the definition of *Accept*, we have $Accept(q, \cdot)$.

$n > 1$ If $s_1 = \cdot$ then by Subject Reduction, we have $\mathcal{A}; q \vdash e_2$. By induction we also have $Accept(q, |s_2 \cdots s_n|)$ and since $s_1 = \cdot$, $Accept(q, |s_1 s_2 \cdots s_n|)$. If $s_1 = f(a_1, \ldots, a_n)$ then by inspection of the operational semantics, we know $\delta(f)(q, a_1, \ldots, a_n) = q'$. By Subject Reduction, $\mathcal{A}; q' \vdash e_2$. By induction, we have $Accept(q', |s_2 \cdots s_n|)$. By definition of *Accept*, we conclude $Accept(q, |s_1 \cdots s_n|)$.

Consequently, for any evaluation sequence $e_1 \longmapsto_{s_1} e_2 \longmapsto_{s_2} \cdots \longmapsto_{s_n} e_{n+1}$,
if $\mathcal{A}; q_0 \vdash e_1$ then $Accept(q_0, |s_1 s_2 \cdots s_n|)$. By definition, $|s_1 s_2 \cdots s_n| \in \mathcal{L}(\mathcal{A})$.

□